Chapter 1

Safe Features

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

sec-safe-cpp14

Chapter 2

Conditionally Safe Features

ch-conditional

sec-conditional-cpp11 Intro text should be here.

sec-conditional-cpp14

Chapter 3

Unsafe Features

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

Reference-Qualified Member Functions

Ref-qualifiers allow developers to overload a nonstatic member function based on the **value category** of the class object.

Description

Even before standardization, C++ supported decorating nonstatic member functions with cv-qualifiers and allowed overloading on those qualifiers:

```
struct Class1
{
    void member1() const;     // (1) const-qualified member
    void member2();           // (2) no qualifiers
    void member2() volatile;  // (3) volatile-qualified overload of (2)
};

void f1()
{
    Class1          v1;
    const Class1    v2;
    volatile Class1 v3;

    v1.member1();  // Calls function (1).
    v2.member1();  // Calls function (1).

    v1.member2();  // Calls overloaded function (2).
    v3.member2();  // Calls overloaded function (3).

    v3.member1();  // Error, no member1 overload matches a volatile object.
    v2.member2();  // Error,  " member2     "       "    " const       "
}
```

The **cv-qualifiers**, **`const`** and/or **`volatile`**, appearing after the parameter list in the member function prototype apply to the object on which the member is called and allow us to overload on the cv-qualification of that object. Overload resolution will select the closest match whose cv-qualifiers are the same as, or more restrictive than, the object's cv-qualification; hence, `v1.member1()` calls a **`const`**-qualified member even though `v1` is not **`const`**. A qualifier cannot be dropped during overload resolution, however, so `v3.member1()` and `v2.member2()` are ill formed.

C++11 introduced a similar feature, adding optional qualifiers that indicate the valid value categories for the expression a member function may be invoked on. Declaring a member function overload specifically for *rvalue* expressions, for example, allows library writers to make better use of move semantics. Note that readers of this feature are presumed to be familiar with value categories (see **??**) and, in particular, the distinction between *lvalue* and *rvalue* references (see Section 2.1."**??**" on page **??**:

```
struct Class2
```

```
{
    void member() &;   // (1)
    void member() &&;  // (2)
};
```

Each member function with a trailing `&` or `&&` is said to be **ref-qualified**; the trailing `&` or `&&` symbols are called ref-qualifiers. The `&` ref-qualifier on overload (1), above, restricts that overload to *lvalue* expressions. The `&&` ref-qualifier on overload (2) restricts that overload to *rvalue* expressions:

```
void f2()
{
    Class2 v;
    v.member();         // Calls overloaded function (1)
    Class2().member();  // Calls overloaded function (2)
}
```

The expression, `v`, is an *lvalue*, so `v.member()` calls the *lvalue*-qualified overload of `member`, whereas the expression, `Class2()`, is an *rvalue*, so calling `member` on it chooses the *rvalue*-qualified overload.

At the heart of understanding both cv-qualifiers and ref-qualifiers on member functions is recognizing the existence of an implicit parameter by which the class object is passed to the function:

```
class Class3
{
    // ...
public:
    void mf(int) &;         // Two parameters: [Class3&       ], int
    void mf(int) &&;        //  "       "    : [Class3&&      ],   "
    void mf(int) const &;   //  "       "    : [const Class3& ],   "
    void mf(int) const &&;  //  "       "    : [const Class3&&],   "
};
```

In each of the four overloads of `mf`, there is a hidden reference parameter (shown in square brackets in the comments) in addition to the explicitly declared **`int`** parameter. The qualifiers at the end of the declarator, i.e., after the parameter list, specify the cv-qualifiers and ref-qualifier for this implicit reference. The **`this`** pointer holds the address of the object passed for this implicit parameter:

```
void Class3::mf(/* [Class3& __self,] */ int i) &
{
    // Implicit Class3 *const this = &__self
    // ...
}

void Class3::mf(/* [Class3&& __self,] */ int i) &&
{
    // Implicit Class3 *const this = &__self
    // ...
}
```

```
void Class3::mf(/* [const Class3& __self,] */ int i) const &
{
    // Implicit const Class3 *const this = &__self
    // ...
}

void Class3::mf(/* [const Class3&& __self,] */ int i) const &&
{
    // Implicit const Class3 *const this = &__self
    // ...
}
```

For descriptive purposes, we will refer to the implicit reference parameter as `__self` throughout this section; in reality, it has no name and cannot be accessed from code. The **`this`** pointer within the function is, therefore, the address of `__self`. Note that the type of **`this`** does not reflect whether `__self` is an *lvalue* reference or an *rvalue* reference; pointer types do not convey the value category of the pointed-to object.

When a member function of an object is called, overload resolution finds the best match for the value category and cv-qualification of all of its arguments, including the implicit `__self` argument:

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

Class3 makeObj();

void f3()
{
    Class3          obj1;
    const Class3    obj2;
    volatile Class3 obj3;
    const Class3&   r1 = obj1;
    Class3&&        r2 = std::move(obj1);  // Note: r2 is an lvalue

    obj1.mf(0);  // Call mf(int) &
    obj2.mf(0);  // Call mf(int) const &
    obj3.mf(0);  // Error, no overload, mf(int) volatile &
    r1.mf(0);    // Call mf(int) const &
    r2.mf(0);    // Call mf(int) &

    makeObj().mf(0);        // Call mf(int) &&
    std::move(obj1).mf(0);  // Call mf(int) &&
    std::move(obj2).mf(0);  // Call mf(int) const &&
}
```

The three objects, `obj1`, `obj2`, and `obj3`, are *lvalues*, so calls to `mf` will match only the *lvalue* reference overloads, i.e., those with a `&` ref-qualifier. As always, overload resolution will pick the version of `mf` that best matches the cv-qualification of the object, without dropping any qualifiers. Thus, the call to `obj2.mf(0)` selects the **`const`** overload whereas the call to `obj3.mf(0)` fails because all candidate functions would require dropping the **`volatile`** qualifier. The **const** *lvalue* reference `r1` matches the **const** *lvalue*-qualified `__self` even though the object to which `r1` is bound is not **const**. Though declared as an *rvalue* reference, a named reference such as `r2` is always an *lvalue* when used in an expression; hence, `r2.mf(0)` calls the non**const** *lvalue*-qualified overload of `mf`.

The function `makeObj` returns an *rvalue* of type `Class3`. When `mf` is called on that *rvalue*, the non**const** *rvalue* reference overload is selected. The expression `std::move(obj1)` also binds to an *rvalue* reference and thus selects the same overload. An *rvalue* reference to **const** occurs relatively rarely in real code, but when it happens, it is usually the result of calling `std::move` on a **const** object (e.g., `obj2`), especially in generic code. Note, however, that a **const** *lvalue* reference can be bound to an *rvalue*; thus, if a matching *rvalue* reference overload is not found, and a **const** *lvalue* reference overload exists, then the latter will match an *rvalue* reference to the class object:

```
class Class4
{
    // ...
public:
    void mf1() &;
    void mf1() const &;
    void mf1() &&;
    // No void mf1() const && overload.

    void mf2() &;
    void mf2() const &;
    // No void mf2() && overload.
    // No void mf2() const && overload.
};

void f4()
{
    const Class4 obj1;
    Class4       obj2;

    std::move(obj1).mf1();  // Calls mf1() const &
    std::move(obj2).mf2();  // Calls mf2() const &
}
```

Syntax and restrictions

syntax-and-restrictions

A ref-qualifier is an optional part of a nonstatic member function declaration. If present, it must come after any cv-qualifiers and before any exception specification. A constructor or destructor may not have a ref-qualifier:

```
void f1() &;  // Error, ref-qualifier on a nonmember function

class Class1
{
    // ...
public:
```

```
    Class1() &&;          // Error, ref-qualifier on constructor.
    ~Class1() &;          // Error, ref-qualifier on destructor.
    void mf() & const;    // Error, ref-qualifier before cv-qualification
    void mf() noexcept&;  // Error, ref-qualifier after exception specification
    void mf() & &&;       // Error, two ref-qualifiers
    static void smf() &;  // Error, ref-qualifier on static member

    void mf(int) const && noexcept;  // OK, ref-qualifier correctly placed.
};
```

A member function that does not have a ref-qualifier can be called for *either* an *lvalue* or an *rvalue*. Thus, C++03 code continues to compile and work as before:

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

class Class2
{
    // ...
public:
    void mf();
    void mf() const;
};

void f2()
{
    Class2       obj1;
    const Class2 obj2;

    obj1.mf();             // Calls mf()
    obj2.mf();             // Calls mf() const
    std::move(obj1).mf();  // Calls mf()
    std::move(obj2).mf();  // Calls mf() const
}
```

For a set of overloads having the same name and the same parameter types, ref-qualifiers must be provided for *all* members in that set or for *none* of them:

```
class Class3
{
    // ...
public:
    void mf1(int*);
    int  mf1(int*) const &;  // Error, prior mf1(int*) is not ref-qualified

    int  mf2(int) const;
    void mf2(int);           // OK, neither mf2(int) is ref-qualified

    int&       mf3() &;
    const int& mf3() const &;
    int&&      mf3() &&;    // OK, all mf3() overloads are ref-qualified
```

```
    void mf4(int);
    void mf4(char*) &&;      // OK, mf4(int) and mf4(char*) are different.

    int mf5(int) &;          // OK, not overloaded

    int&& mf6() &&;          // OK, not overloaded
};
```

Note that the overload of **mf1** is ill formed even though the unqualified and ref-qualified versions have different return types and different cv-qualifiers.

Member function templates may also have ref-qualifiers:

```
class Class4
{
    // ...
public:
    template <typename T> Class4&  mf(const T&) &;
    template <typename T> Class4&& mf(const T&) &&;
};
```

Within a member function's body, regardless of whether the member has a `&` ref-qualifier, a `&&` ref-qualifier, or no ref-qualifier at all, uses of `*this` and of any nonstatic data members yield *lvalues*. Although arguably counterintuitive, this behavior is identical to the way that other reference parameters work:

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

template <typename T> bool isLvalue(T&)  { return true; }
template <typename T> bool isLvalue(T&&) { return false; }

class Class5
{
    int d_data;
public:
    void mf(int&& arg) &&
    {
        assert(isLvalue(arg));     // OK, named reference is an lvalue
        assert(isLvalue(*this));   // OK, pointer dereference is an lvalue
        assert(isLvalue(d_data));  // OK, member of an lvalue is an lvalue
    }

    void mf(int& arg) &
    {
        assert(isLvalue(arg));     // OK, named reference is an lvalue
        assert(isLvalue(*this));   // OK, pointer dereference is an lvalue
        assert(isLvalue(d_data));  // OK, member of an lvalue is an lvalue
    }
};
```

If a member function calls another member function on the same object, only the *lvalue*-qualified overloads are considered:

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

struct Class6
{
    bool mf1() &  { return false; }  // Return false if called on lvalue
    bool mf1() && { return true; }   //    "   true   "    "    " rvalue

    void mf2() && { assert(false == mf1()); }  // Calls lvalue overload
};
```

Function `mf2`, although *rvalue*-qualified, nevertheless calls the *lvalue*-qualified overload of `mf1` because `*this` is an *lvalue*. If the desired behavior is to propagate the value category of the object on which it was called, `std::move` (or another reference cast) must be used:

```
class Class7
{
    int d_data;

public:
    bool mf1() &  { return false; }  // Return false if called on lvalue
    bool mf1() && { return true; }   //    "   true   "    "    " rvalue

    void mf2(int&& arg) &&
    {
        assert(! isLvalue(std::move(arg)));
        assert(! isLvalue(std::move(*this)));
        assert(! isLvalue(std::move(d_data)));

        assert(std::move(*this).mf1());
    }
};
```

In this example, each call to `std::move` *reconstitutes* the value category of the original object. Note that we must mention `*this` explicitly in order to call the *rvalue*-qualified overload of `mf1`.

The ref-qualifier, if any, is part of a member function's signature and is, therefore, part of its type and the type of a corresponding pointer to member function:

```
struct Class8
{
    void mf1(int) &;   // (1)
    void mf1(int) &&;  // (2)

    void mf2(int);     // (3)
};

using Plqf = void (Class8::*)(int)&;   // pointer to lvalue-qualified function
using Prqf = void (Class8::*)(int)&&;  //    "      " rvalue-qualified     "
using Puqf = void (Class8::*)(int);    //    "      " unqualified           "

void f8()
```

```
{
    Plqf lq = &Class8::mf1;  // OK, pointer to member function (1)
    Prqf rq = &Class8::mf1;  // OK,     "     "    "       "    (2)
    Puqf xq = &Class8::mf1;  // Error, mf1 is ref-qualified but Puqf is not.

    Puqf uq = &Class8::mf2;  // OK, pointer to member function (3)
    Plqf yq = &Class8::mf2;  // Error, mf2 is not ref-qualified but Plqf is.

    Class8 v;
    (v.*lq)(0);         // Call v.mf1(int), overload (1)
    (Class8{}.*rq)(1);  // Call Class8{}.mf1(int), overload (2)
    (v.*rq)(2);         // Error, rq expects an rvalue object.
}
```

Note that `Plqf`, `Prqf`, and `Puqf` are three different, mutually-incompatible, types that reflect the ref-qualifier of the member function they each point to.

use-cases-refqualifier

Use Cases

-subobject-of-an-rvalue

Returning a subobject of an *rvalue*

Many classes provide accessors that return a reference to a member of the class. We can gain performance benefits if those accessors returned *rvalue* references when called on an *rvalue* object:

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

class RedString
{
   std::string d_value;

public:
    RedString(const char* s = "") : d_value("Red: ") { d_value += s; }

    std::string&        value() &       { return d_value; }
    std::string const&  value() const & { return d_value; }
    std::string&&       value() &&      { return std::move(d_value); }

    // ...
};

void f2()
{
    RedString       rs1("hello");
    const RedString rs2("world");

    std::string h1 = rs1.value();                    // "Red: hello"
    std::string h2 = rs2.value();                    // "Red: world"
    std::string h3 = RedString("goodbye").value();   // "Red: goodbye"
```

```
    std::string h4 = std::move(rs1).value();        // "Red: hello"
    std::string h5 = rs1.value();                   // Bug, unspecifiedvalue

    std::string h6 = std::move(rs2).value();        // "Red: world"
    std::string h7 = rs2.value();                   // OK, "Red: world"
}
```

The `RedString` class provides three ref-qualified overloads of `value`. When `value` is called on `rs1` and `rs2`, the non**`const`** and **`const`** *lvalue*-qualified overloads, respectively, are selected. Both return an *lvalue* reference to `std::string`, so `h1` and `h2` are constructed using the copy constructor, as usual. In the case of the temporary variable created by `RedString("goodbye")`, however, the *rvalue*-qualified overload of `value` is selected. It returns an *rvalue* reference, so `h3` is invoked using the move constructor, which is more efficient.

As in the case of most such code, it is assumed that an *rvalue* reference refers to an object whose state no longer matters after evaluation of the expression. When that assumption doesn't hold, unexpected results may occur, as in the case of `h5`, which is initialized from a moved-from string, yielding a valid but unspecified string value.

The `value` method is not overloaded for a **`const`** *rvalue*-qualified object. Invoking it for such a (rarely encountered) type selects the **`const`** *lvalue*-qualified overload, as *rvalues* can always be bound to **`const`** *lvalue* references. As a result, `h6` is initialized from a **`const`** `std::string&`, invoking the copy constructor and leaving `rs2` unmodified.

One danger of this design is that the reference returned from the *rvalue*-qualified overload could outlive the `RedString` object:

```
void f3()
{
    std::string&& s = RedString("goodbye").value();
    char c = s[0];  // Bug, s refers to a deleted string
}
```

The temporary variable created by the expression `RedString("goodbye")` is destroyed at the end of the statement; **lifetime extension** does not come into play because `s` is not bound to the temporary object itself, but to a reference returned by the `value` method. This situation can be avoided by returning by *value* rather than by reference:

```
class BlueString
{
   std::string d_value;

public:
    BlueString(const char* s = "") : d_value("Blue: ") { d_value += s; }

    std::string&        value() &       { return d_value; }
    std::string const&  value() const & { return d_value; }
    std::string         value() &&      { return std::move(d_value); }

    // ...
};

void f4()
```

```
{
    std::string h = BlueString("hello").value();

    std::string&& s = BlueString("goodbye").value();
    char c = s[0];  // OK, lifetime of s has been extended
}
```

The expression `BlueString("hello").value()` yields a temporary `std::string` initialized via move-construction from the member variable `d_value`. The variable `h` is, in turn, move-constructed from the temporary variable. Compared to the `RedString` version of `value` that returned an *rvalue* reference, this sequence logically has one extra move operation (two move-constructor calls instead of one). This extra move does not pose a problem in practice because (a) move construction of `std::string` objects is cheap and (b) most C++11 and C++14 compilers will *elide* the extra move anyway, yielding equivalent code to the `RedString` case.[1]

Similarly, the expression `BlueString("goodbye").value()` yields a temporary `std::string`, but in this case the temporary variable is bound to the reference, `s`, which causes its lifetime to be extended until `s` goes out of scope. Thus, `s[0]` safely indexes a string that is still live.

Note one more, rather subtle, difference between the behavior of `value` for `RedString` versus `BlueString`:

```
void f5()
{
    RedString  rs("hello");
    BlueString bs("hello");

    std::move(rs).value();  // rs.d_value is unchanged
    std::move(bs).value();  // bs.d_value is moved from
}
```

Calling `value` on an *rvalue* of type `RedString` doesn't actually change the value of `d_value`; it is not until the returned *rvalue* reference is actually used (e.g., in a move constructor) that `d_value` is changed. Thus, if the return value is ignored, nothing happens. Conversely, for `BlueString`, the return of `value` is always move constructed, causing `d_value` to end up in a moved-from state, even if the return value is ultimately ignored. This difference in behavior is seldom important in practice, as reasonable code will assume nothing about the value of a variable after it was used as the argument to `std::move`.

Forbidding modifying operations on *rvalues*

g-operations-on-rvalues

Modifying an *rvalue* means modifying a temporary object that is about to go out of scope. A common example of a bug resulting from this behavior is accidental assignment to a temporary object. Consider a simple `Employee` class with a `name` accessor and a function that attempts to set the name:

[1] Beginning with C++17, the description of the way return values are initialized changed so as to no longer *materialize* a temporary variable in this situation. This change is sometimes referred to as *mandatory copy/move elision* because, in addition to defining a more consistent and portable semantic, it effectively legislates the optimization that was previously optional.

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

class Employee
{
public:
    // ...
    std::string name() const;
    // ...
};

void f1(Employee& e)
{
    e.name() = "Fred";
}
```

The programmer probably thought that the assignment to `e.name()` would result in updating the name of the `Employee` object referenced by `e`. Instead, it modifies the temporary string returned by `e.name()`, having no effect whatsoever except, possibly, to consume CPU resources.

One way to prevent these sorts of accidents is to design a class interface with ref-qualified modifiers that are callable only for non**`const`** *lvalues*:

```
class Name
{
    std::string d_value;

public:
    Name() = default;
    Name(const char* s) : d_value(s) {}
    Name(const Name&) = default;
    Name(Name&&) = default;

    Name& operator=(const Name&) & = default;
    Name& operator=(Name&&) &      = default;
    // ...
};
```

Note that both the copy- and move-assignment operators for `Name` are ref-qualified for *lvalues* only. Overload resolution will not find an appropriate match for assignment to an *rvalue* of type `Name`:

```
class Employee2
{
    Name d_name;

public:
    // ...
    Name name() const { return d_name; }
    // ...
};
```

```
void f2(Employee2& e)
{
    e.name() = "Fred";  // Error, cannot assign to rvalue of type Name
}
```

Now, assignment to the temporary returned by `e.name()` fails to find a matching assignment operator, so the accidental assignment is avoided by an error message. The same approach can be used to avoid many other accidental modifications on *rvalues*, including inserting elements, erasing elements, etc. Note, however, that modifying a temporary is not always a bug; see **??** — *Forbidding modifications to rvalues breaks legitimate use cases* on page 22.

Forbidding operations on *lvalues*

g-operations-on-lvalues

If an instance of a class is intended to exist only for the duration of a single expression, then it might be desirable to disable most operations on *lvalues* of that type. For example, an object of type `LockableStream`, below, works like an `std::ostream` except that it acquires a mutex for the duration of a single streaming expression. It does this by creating a proxy object, `LockedStream`, that is useful only as an *rvalue*:

```
#include <iostream>  // std::ostream, std::cout, std::endl
#include <mutex>     // std::mutex
#include <cassert>   // standard C assert macro

class LockableStream;  // forward reference

class LockedStream
{
    friend class LockableStream;

    LockableStream* d_lockable;

    explicit LockedStream(LockableStream* ls) : d_lockable(ls) { }

public:
    LockedStream(const LockedStream&) = delete;
    LockedStream(LockedStream&& other) : d_lockable(other.d_lockable)
    {
        other.d_lockable = nullptr;
    }

    LockedStream& operator=(const LockedStream&) = delete;
    LockedStream& operator=(LockedStream&&) = delete;

    ~LockedStream();

    template <typename T> LockedStream operator<<(const T& v) &&;
};
```

A `LockedStream` object holds a pointer to a `LockableStream`. It is move-only (not copyable) and not assignable. Its move constructor transfers ownership of the `LockableStream`, which, as we shall see, implicitly transfers ownership of the mutex. The streaming operator, **operator**<<, can be invoked only on an *rvalue*. Note that the only way to construct a LockedStream is through a private constructor that is invoked by the **friend** class, LockableStream:

```
class LockableStream
{
    std::mutex    d_mutex;
    std::ostream& d_os;

    friend class LockedStream;

public:
    LockableStream(std::ostream& os) : d_os(os) { }

    void lock()   { d_mutex.lock(); }
    void unlock() { d_mutex.unlock(); }

    template <typename T> LockedStream operator<<(const T& v)
    {
        lock();
        return LockedStream(this) << v;
    }
};
```

A LockableStream holds an std::mutex and a reference to an std::ostream object. The streaming operator, **operator**<<, locks the mutex, constructs a LockedStream object, and delegates the actual streaming operation to the LockedStream. The return value of **operator**<< is an *rvalue* of type LockedStream.

Each invocation of **operator**<< on a LockedStream outputs to the stored std::ostream, then returns ***this** by move construction:

```
template <typename T>
LockedStream LockedStream::operator<<(const T& v) &&
{
    assert(d_lockable);  // assert *this is not in moved-from state
    d_lockable->d_os << v;
    return std::move(*this);
}
```

When the last invocation of **operator**<< in a chain of such invocations completes, the returned LockedStream has ownership of the LockableStream. Its destructor then unlocks the mutex:

```
LockedStream::~LockedStream()
{
    if (d_lockable)
    {
        d_lockable->unlock();
    }
}
```

Note that `d_lockable` will be **`nullptr`** if the `LockedStream` is in the moved-from state, as most of the temporary `LockedStream` objects in the chain will be. Finally, we can create a `LockableStream` and print to it:

```
LockableStream lockableCout(std::cout);

void f5()
{
    lockableCout << "Hello, " << 2021 << "\n";

    LockedStream ls(lockableCout << "Hello, ");
    ls << 2021;  // Error, can't stream to lvalue
}
```

Similar code in other threads can print to `lockableCout` concurrently; the locking protocol will prevent them from creating a race condition. The first statement in `f5` acquires the lock, prints `"Hello, 2021"` followed by a newline, then releases the lock automatically. An attempt to break this sequence into multiple statements fails because an *lvalue* of type `LockedStream` cannot be used for streaming.

Note that this idiom is intended to protect the user from casual errors only. If the user is intent on doing so, they can cast an *lvalue* of `LockedStream` to an *rvalue* using `std::move`, and they can prevent a `LockedStream` from going out of scope by using lifetime extension. These workarounds can be used safely, if applied consciously, so it is not necessary to protect against such usage.

Optimizing immutable types and builder classes

pes-and-builder-classes

An **immutable type** is a type that has no modifying operations except for assignment. Among other benefits, the representation of an immutable type can be shared by all objects that have the same value, including in concurrent threads. Every object that logically "modifies" an object of immutable type does so by returning a new object having the modified value; the original object remains unchanged. An `ImmutableString` class, for example, might have an `insert` method that takes a second string argument and returns a copy of the original string with the second string inserted in the specified location:

```
#include <memory>    // std::shared_ptr
#include <string>    // std::string
#include <iostream>  // std::ostream, std::cout, std::endl

class ImmutableString
{
    std::shared_ptr<std::string> d_dataPtr;

    static const std::string s_emptyString;

public:
    using size_type = std::string::size_type;

    ImmutableString() {}
```

```
    ImmutableString(const char* s)
        : d_dataPtr(std::make_shared<std::string>(s)) { }

    ImmutableString(std::string s)
        : d_dataPtr(std::make_shared<std::string>(std::move(s))) { }

    ImmutableString insert(size_type pos, const ImmutableString& s) const
    {
        std::string dataCopy(asStdString());    // Copy string from this object.
        dataCopy.insert(pos, s.asStdString());  // Do insert.
        return std::move(dataCopy);             // Move into return value.
    }

    const std::string& asStdString() const
    {
        return d_dataPtr ? *d_dataPtr : s_emptyString;
    }

    friend std::ostream& operator<<(std::ostream& os, const ImmutableString& s)
    {
        return os << s.asStdString();
    }
    // ...
};

const std::string ImmutableString::s_emptyString{};
```

The internal representation of an `ImmutableString` is an `std::string` object allocated on the heap and accessed via an instantiation of the C++ Standard reference-counted smart pointer, `std::shared_ptr`. The copy and move constructors and assignment operators are defaulted; when an `ImmutableString` is copied or moved, only the smart pointer member is affected. Thus, even large string values can be copied in constant time.

Note that the `insert` method begins by making a copy of the *internal representation* of the immutable string. The copy is modified then returned; the representation in the original `ImmutableString` is not modified:

```
void f1()
{
    ImmutableString is("hello world");
    std::cout << is << std::endl;                 // Print "hello world"
    std::cout << is.insert(5, ",") << std::endl;  // Print "hello, world"
    std::cout << is << std::endl;                 // Print "hello world"
}
```

Immutable types are often paired with *builder* classes — mutable types that are used to "build up" a value, which is then "frozen" into an object of the immutable type. Let's define a `StringBuilder` class with mutating `append` and `erase` methods that modify its internal state, and a conversion operator that returns an `ImmutableString` containing the built-up value:

```
class StringBuilder
{
    std::string d_string;

public:
    using size_type = std::string::size_type;

    StringBuilder&  append(const char* s) &  { d_string += s; return *this; }
    StringBuilder&& append(const char* s) && { return std::move(append(s)); }

    StringBuilder&  erase(size_type pos, size_type n) &
    {
        d_string.erase(pos, n);
        return *this;
    }
    StringBuilder&& erase(size_type pos, size_type n) &&
    {
        return std::move(erase(pos, n));
    }

    operator ImmutableString() && { return std::move(d_string); }
};
```

The `append` and `erase` methods are each ref-qualified and overloaded for both *lvalues* and *rvalues*. The only difference between the overloads is that the *lvalue* overloads each return an *lvalue* reference and the *rvalue* overloads each return an *rvalue* reference. In fact, in each case, the *rvalue* overload simply calls the corresponding *lvalue* overload, then calls `std::move` on the result. This technique works, and does not cause infinite recursion within the *rvalue* overload, because `*this` is always an *lvalue*, just as a parameter of *rvalue* reference is always an *lvalue* within a function.

The operator to convert from `StringBuilder` to `ImmutableString` is *destructive* in that it moves the built-up value out of the builder into the returned string. It is ref-qualified for *rvalue* references only — if the builder is not an *rvalue*, then the user must deliberately call `std::move`. This protocol acts a signal to the future maintainer that the builder object is in a moved-from state after the conversion and cannot be reused after its value has been extracted:

```
void f2()
{
    StringBuilder builder;
    builder.append("apples, pears, bananas");
    builder.erase(8, 7);
    ImmutableString s1 = builder;            // Error, can't convert lvalue
    ImmutableString s2 = std::move(builder); // OK, convert rvalue reference
    std::cout << s2 << std::endl;            // print "apples, bananas"

    ImmutableString s3 = StringBuilder()     // modify builder rvalue
        .append("apples, pears, bananas")
        .erase(8, 7);                        // OK, convert pure rvalue
```

```
    std::cout << s3 << std::endl;             // print "apples, bananas"
}
```

The `builder` object is an *lvalue* and is intended to be modified several times before yielding a built-up `ImmutableString` value. After it is modified using `append` and `erase` — selecting the *lvalue* overloads in both cases — attempting to convert it directly to `ImmutableString` fails because there is no such conversion from an *lvalue* `builder`. The initialization of `s2`, conversely, succeeds, *moving* the value from the `StringBuilder` into the result.

The expression `StringBuilder()` constructs an *rvalue*, which is then modified by a chain of calls to `append` and `erase`. The *rvalue* overload of `append` is selected, which returns an *rvalue* reference that, in turn, drives the selection of the *rvalue* overload of `erase`. Because the result of the chain of modifiers is an *rvalue* reference, **`operator`** `ImmutableString` can be invoked without calling `std::move`. This usage is safe because the temporary `StringBuilder` object is destroyed immediately afterward, so there is no opportunity for improperly reusing the builder object.

-pitfalls-refqualifer

Potential Pitfalls

-legitimate-use-cases

Forbidding modifications to *rvalues* breaks legitimate use cases

An earlier use case, *Use Cases — Forbidding modifying operations on rvalues* on page 15, is also the subject of a potential pitfall. Consider a string class with a `toLower` modifier method:

```
class String
{
public:
    // ...
    String& toLower();
        // Convert all uppercase letters to lowercase, then return modified
        // *this object.
};

void test()
{
    String x{ /*...*/ };  // Variable of type String
    String f();           // Function returning String

    String& a = x.toLower();    // OK, a refers to x
    f().toLower();              // Bug (1), modifies temporary variable; no-op
    String& b = f().toLower();  // Bug (2), b is a dangling reference
}
```

Bug (1) is a statement that modifies a temporary variable and, hence, has no effect. Bug (2) illustrates how `toLower` unintentionally acts as an *rvalue*-to-*lvalue* reference cast because it returns an *lvalue* reference to a (possibly *rvalue*) object. The *lvalue* reference, `b`, is bound to the modified temporary `String` returned by `f()`, after it is modified by `toLower`. At the end of the statement, the temporary object is destroyed, causing `b` to become a **dangling reference**.

Given these issues, it is tempting to add a ref-qualifier to `toLower` so that it can be called only on *lvalues*:

```
class String
{
public:
    // ...
    String& toLower() &;
};
```

Although this ref-qualification prevents do-nothing modifications to a temporary `String`, it also prevents legitimate uses of `toLower` on an *rvalue*:

```
String c = f().toLower();  // Error, toLower cannot be called on an rvalue
```

Here, the return value of `toLower` would be used to initialize `c` to a copy of the modified `String`. Unfortunately, we've prohibited calling `toLower` with an *rvalue*, so the call is ill formed. This pitfall might manifest any time we suppress modification of *rvalues* for a method that returns a value or has a side effect.

We could, of course, create ref-qualified overloads for *both lvalue* and *rvalue* objects, returning by *lvalue*-reference or *by value*, respectively, as we saw in the `BlueString` class in *Use Cases — Returning a subobject of an rvalue* on page 13 Returning a subobject of an rvaluexref, but doing so ubiquitously can become a maintenance burden; see *Annoyances — Providing ref-qualified overloads may be a maintenance burden* on page 23.

annoyances-refqualifier

Annoyances

be-a-maintenance-burden

Providing ref-qualified overloads may be a maintenance burden

Having two or more ref-qualified overloads of a member function can confer expressiveness and safety to a class. The trade-off is that these overloads expand the class interface and usually require code duplication, which can become a maintenance burden:

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

class Thing
{
    std::string      d_name;
    std::vector<int> d_data;
    // ...

public:
    // ...

    std::string const& name() const & { return d_name; }
    std::string        name() &&      { return std::move(d_name); }

    std::vector<int>      &  data()       &  { return d_data; }
    std::vector<int> const&  data() const &  { return d_data; }
    std::vector<int>      && data()       && { return std::move(d_data); }
```

```
    std::vector<int> const&& data() const && { return std::move(d_data); }

    Thing& rename(const std::string& n) &  { d_name = n; return *this; }
    Thing& rename(std::string&& n) &  { d_name = std::move(n); return *this; }
    Thing  rename(const std::string& n) &&
    {
        d_name = n;
        return *this;  // Bug, should be return std::move(*this).
    }
    Thing  rename(std::string&& n) &&
    {
        return std::move(rename(std::move(n)));  // Delegate to lvalue overload
    }
};
```

The `name` member is a classic accessor. Overloading it based on ref-qualification provides an optimization so that the `d_name` string can be moved instead of copied when the `Thing` object is expiring. Because it is an accessor, only the **`const`** *lvalue* and non**`const`** *rvalue* overloads are needed; other cv-qualifications do not make sense.

A modifiable `Thing` object can be modified via the return type of its `data` member function, but a **`const`** `Thing` cannot. We are used to overloading based on **`const`**, but adding ref-qualification doubles the number of overload combinations.

Finally, the `rename` member illustrates a different kind of combinatorial overload set. This member is overloaded on the value category of both the `Thing` argument and the `n` argument. In addition to the total number of overloads for a single function, this example illustrates a potential performance bug that occurs easily when cutting-and-pasting numerous similar function bodies: by returning `*`**`this`** instead of `std::move(*`**`this`**`)` in the first *rvalue*-qualified overload, the return value is copy constructed instead of move constructed.

One way to mitigate the maintenance burden of having many overloads is for the *rvalue*-qualified overloads to delegate to the the *lvalue*-qualified ones, as seen in the last *rvalue*-qualified overload of `rename`. Note that `*`**`this`** is *always* an *lvalue*, even within the *rvalue*-qualified overloads, so the call to `rename` within the *rvalue*-qualified version does not result in a recursive call to itself but instead results in a call to the *lvalue*-qualified version.

See Also

Further Reading

sec-unsafe-cpp14