# Chapter 1

## Safe Features

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

# Chapter 1 Safe Features

sec-safe-cpp14

# Chapter 2

## Conditionally Safe Features

ch-conditional

sec-conditional-cpp11 Intro text should be here.

# Move Semantics and *Rvalue* References (&&)

and-Rvalue-References

The introduction of ***rvalue* references** in C++11 provides a consistent mechanism to identify circumstances under which an object's state may be safely *moved* to a new object, a.k.a. **move semantics**. Operations having move semantics often afford a more runtime-efficient alternative to conventional copy operations. Moreover, *rvalue* references enable the creation of **move-only types**, which can be used to represent unique, transferable ownership of a given resource.

## Description

description-rvalueref

*Rvalue* references are perhaps the defining language feature of modern C++. To enable the introduction of move semantics, the C++ language evolved the notion of *lvalues* and non*lvalues* to a number of value categories, allowing for a mostly smooth ability to capture the moment when an object's value is no longer needed and can, consequently, can have its internal state taken instead of copied. For a discussion of the history and motivation of the evolution of value categories as well as the foundation of their meaning prior to C++11, see *Appendix — The evolution of value categories* on page 87.

### Introduction to *rvalue* references

-to-rvalue-references

Prior to C++11, the only kind of reference type in C++ was the ***lvalue* reference**. For any type `T`, the type `T&` is an *lvalue* reference to `T`, and entities with this reference type act as alternate names for the objects they refer to:

```
int  c;
int& cr  = c;   // lvalue reference to c
int* p   = &c;
int& cpr = *p;  // also lvalue reference to c
```

Binding a **`const`** *lvalue* reference to an expression that can be used to initialize a temporary object, such as a literal value or a call to a function that returns by value (which are examples of *prvalues*; see *Prvalues in C++11/14* on page 11), creates a temporary object whose lifetime will be extended to that of the *lvalue* reference; see *Appendix — Lifetime extension of a temporary bound to a reference* on page 91:

```
const int& d = 5;    // lvalue reference to temporary with value 5

int v();             // returns an int by value
const int& e = v();  // lvalue reference to temporary returned by v
```

Non**`const`** *lvalue* references cannot, however, bind to temporaries:

```
int& f = 7;    // Error, cannot bind temporary to nonconst lvalue reference
int& g = v();  // Error,    "     "      "      "  "    "       "       "
```

Being unable to bind non**`const`** *lvalue* references to temporaries protects programmers from inadvertently storing information in an object that is clearly at the end of its lifetime. On the other hand, this restriction also prevents programmers from taking advantage of the

imminent destruction of the referent by assuming ownership of any resources that it might control.

To support creating references to objects that are at the end of their lifetimes and to enable modification of such objects, *rvalue* references were added in C++11. For any type `T`, the type `T&&` is an *rvalue* reference to `T`. The primary distinction between an *lvalue* reference and an *rvalue* reference is that non**const** *rvalue* references can bind to temporary objects:

```
int&& rf = 7;    // OK, rvalue reference to temporary with value 7
int&& rg = v();  // OK, rvalue reference to temporary returned by v
```

An *rvalue* reference will not, however, bind to *lvalues*:

```
int   h;
int&& rvh1 = h;    // Error, cannot bind rvalue reference to lvalue
int*  hp   = &h;
int&& rvh2 = *hp;  // Error, cannot bind rvalue reference to lvalue
```

Importantly, *lvalues* can be explicitly cast to *rvalue* references using a **`static_cast`**:

```
int&& rvh3 = static_cast<int&&>(h);    // OK
int&& rvh4 = static_cast<int&&>(*hp);  // OK
```

These restrictions prevent creating *rvalue* references to objects where there has been no implicit or explicit indication that the object's value is no longer going to be needed.

introduction-to-xvalues

## Introduction to *xvalues*

The quality of having a clear distinction for a reference to an object whose value is no longer needed is essential to understanding the purpose and use of *rvalue* references. Objects whose lifetime has already been started can only be bound to an rvalue reference when they are identified by an expression with the *xvalue* **value category**, which is new to C++11. As we saw above, *rvalue* references can also bind to *prvalues*; in that case, a temporary is created to which the reference is bound.

The C++11 *xvalue* value category can arise in two primary ways. Any expression that produces a temporary value will have the *xvalue* value category, enabling an *rvalue* reference to bind to the temporary and make use of its contents:

```
struct S { };
S f();          // OK, function that returns by value
S&& rs1 = f();  // OK, rvalue reference binds to temporary S object
```

The other way to have an expression with the *xvalue* value category is to explicitly form an expression with a type that is an *rvalue* reference, either by invoking a function that returns such a type or by using a cast to convert an *lvalue* to an *rvalue* reference:

```
S&& g();
S&& rs2 = g();                  // OK
S s;
S&& rs3 = static_cast<S&&>(s);  // OK
```

Of particular note is the Standard Library utility `std::move`, which is a function returning an *rvalue* reference, like `g` in the example above, with an implementation like the initializer

for `rs3` above — namely, just a **`static_cast`** to an *rvalue* reference; see *The* `std::move` *utility* on page 20:

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

S&& rs4 = std::move(s);  // OK, same initialization as rs3
```

Casting an *lvalue* into an *xvalue* like this allows for the explicit indication that a **move operation** is allowed on the object being referenced and that its value is no longer going to be needed.

n-overload-resolution

## Introduction to modern overload resolution

During overload resolution, when choosing between a function with a matching *rvalue* reference parameter and a function with a matching **`const`** *lvalue* reference parameter, the *rvalue* reference parameter has higher priority. This prioritization comes into play when the argument is an *xvalue* or a *prvalue*:

```
void f(const int&);  // (1) const lvalue reference
void f(int&&);       // (2) rvalue reference

void test()
{
    int i;
    f(i);             // lvalue, invokes (1)
    f(std::move(i));  // xvalue, invokes (2)
    f(5);             // prvalue, invokes (2)
}
```

move-operations

## Move operations

To take advantage of the new kind of reference type, C++11 also added two new **special member functions** to user-defined class types: the **move constructor** and the **move-assignment operator**. The move constructor parallels the **copy constructor** of a class but instead of taking an *lvalue* reference parameter (which is usually **`const`**), it takes an *rvalue* reference parameter:

```
struct S1
{
    S1(const S1&);  // copy constructor
    S1(S1&&);       // move constructor
};
```

Similarly, the move-assignment operator parallels the **copy-assignment operator** but takes an *rvalue* reference parameter instead:

```
struct S2
{
    S2& operator=(const S2&);  // copy-assignment operator
    S2& operator=(S2&&);       // move-assignment operator
};
```

Both of these new special member functions participate in overload resolution alongside the corresponding copy operations and are eligible and preferred for arguments that are *xvalues* or *prvalues*. These move operations can then do what their name suggests and move the value of the source object, along with any resources that it owns, into the target object without regard for leaving the source object in a useful state, though see *Potential Pitfalls — Inconsistent expectations on moved-from objects* on page 70 for many considerations about what state in which it should be left.

When programming with objects that support move operations, the general assumption is to have no expectations on the state of a moved-from object, although most well-behaved types will support destruction of a moved-from object and assignment of a new value to the object. The unspecified state of a moved-from object can be, if appropriate, the same as its state prior to the move, and from a user perspective, the copy operation on a type generally meets the contract of a move operation. In cases where a type has a copy operation, the move operation often ends up as an optimization that seamlessly reuses no-longer-needed resources from a source object and results in a destination object with the same value that copying would have produced.

motivation

## Motivation

The motivation for the introduction of move semantics was the desire to address common problems involving potentially expensive copying of data in situations where such copies were unnecessary. Consider, for example, the task of swapping the values of two vectors. A simple C++03 implementation would involve at least one allocation and deallocation as well as multiple element-wise copies:

```
void swapVectors(std::vector<int>& v1, std::vector<int>& v2)
{
    std::vector<int> temp = v1;
    v1 = v2;
    v2 = temp;
}
```

These allocations and copies are, however, unnecessary in principle: After execution of `swapVectors`, the program still refers to two heap-allocated buffers containing the values, and simply exchanging the pointers within the vectors being swapped would suffice to achieve the same result. This issue would be further exacerbated if the elements of the vectors were expensive to copy, such as elements of type `std::vector` or `std::string`. Similarly, when growing the buffer within a `std::vector<std::string>` (e.g., to accommodate an additional element), making a copy of each string despite a clear understanding that the old ones will be destroyed immediately after copying them is inefficient.[1] Finally, copying a temporary object that is about to be destroyed is equally wasteful. Move semantics addresses these issues by allowing a potentially more efficient transfer (move) of resource ownership from one object to another in circumstances where a copy would be followed by discarding the original.

[1] Note that, for clarity, we're not involving the strong exception-safety guarantee provided by `std::vector` in this discussion. For more on moves, vectors, and the exception-safety guarantee, see Section 2.1."**??**" on page **??**.

C++11 added *rvalue* references, move operations, and *xvalues* as a new value category. These all work together to enable the use of move semantics as an optimization of copy semantics in a number of scenarios. In addition to the potential for optimization, by providing *only* move operations and *no* copy operations, move-only types can be developed. The prototypical move-only type, `std::unique_ptr`, was introduced to solve the long-standing problem of the many pitfalls of using `std::auto_ptr`. This new type allowed `std::auto_ptr` to be deprecated in C++11.[2]

ategories-in-c++11/14

## Extended value categories in C++11/14

There are three disjoint value categories in modern C++.

1. *lvalues* comprise objects in memory that could conceivably have their address taken but which have not been designated, either implicitly or explicitly, as no longer needing their values and potentially nearing the end of their lifetimes. One important characteristic of *lvalues* is that the address operator `&` requires an *lvalue* as its operand.

2. *xvalues* are also objects in memory but specifically include those that are nearing the end of their lifetime. This includes temporary objects and *lvalues* that have been transformed through various means involving *rvalue* references, such as **`static_cast`**`<T&&>` or the Standard Library `std::move` utility function. Many operations that access subobjects of an *xvalue* expression also yield *xvalues*.

3. *prvalues* are values that have not *yet* been used to populate an object but could be used to do so. This includes literals such as `12`, `7.3e5`, and **`true`** as well as functions that return values that are not references.

Two additional value categories have been defined that capture some of the common properties amongst the value categories: [**AUs: do you want to continue the previous numbering and make these 4 and 5?**]

1. *glvalues*, also called generalized *lvalues*, are all expressions that are either an *lvalue* or an *xvalue*. All *glvalues* represent objects in memory that have an address in memory, although the `&` operator can only be applied to *lvalues*. Many operations, such as applying an operator other than `&` or invoking a member function, require a *glvalue* as an operand or argument. When an expression is a *prvalue* but not a *glvalue* and a *glvalue* is needed, a process called **temporary materialization** initializes a temporary to effectively convert the *prvalue* into an *xvalue* and thus a *glvalue*.

2. *rvalues* are those expressions that are not *lvalues* or, alternatively, expressions that are neither an *xvalue* nor a *prvalue*. All expressions that are *rvalues* can be bound to *rvalue* references; *xvalues* can be bound directly, and *prvalues* can be bound through implicit temporary materialization to produce an *xvalue* to which the reference, in turn, can be bound.[3]

[2] Unlike most other deprecations, `std::auto_ptr` has not lingered in the standard and was completely removed in C++17.

[3] In general, the distinction between a *prvalue* and an *xvalue* is not critical, as they interact with most

lvalues-in-c++11/14

**Lvalues in C++11/14** An ***lvalue*** expression is, by definition, one for which the built-in *address-of* operator, `&`, can be applied to obtain an address:

```
// named lvalue          Taking the address of an lvalue using &
// --------------          ----------------------------------------

double d;                double* dp            = &d;
                         double* dp2           = &(d += 0.5);

double& dr = d;          double*  dp3          = &dr;
                         double** dpp          = &dp3;

const int& cir = 1;      const int*  cip       = &cir;
                         const int** cipp      = &cip;

int f();                 int (*fp)()           = &f;

char a[10];              char (*ap)[10]        = &a;
                         char* cp              = &a[5];

struct S { int x; } s;   S* sp                 = &s;
                         S* sp2                = &(s = s);
                         int* ip               = &s.x;

unsigned& g();           unsigned* up          = &g();

                         const char (*lp)[14]  = &"Hello, World!";
                         const char* lp2       = &"Hello, World!"[5];
```

An *lvalue* expression is never a *temporary*; it identifies, e.g., by name or address, a pre-existing object that will outlive the end of the largest enclosing expression in which it resides. Applying the built-in dereference operator `*` to any valid, nonnull pointer variable (i.e., to any type other than **`void`**), such as `dp` or `fp` in the example above, will produce an unnamed *lvalue* (i.e., `*dp` and `*fp` are *lvalues*) as will invoking a function, such as `g()` in the example above, returning a pre-existing object by *lvalue* reference (i.e., `g()` is an *lvalue*).

Our ability to assign to or otherwise modify an *lvalue* of built-in type is governed entirely by whether it is **`const`**-qualified; for user-defined types, assignment is additionally governed by whether *copy* and/or *move* assignment is supported for that type; see *Appendix — Special member function generation* on page 21. Finally, note that character-string literals (e.g., "Hello, World!"), unlike other literals, are considered *lvalues*, as illustrated by initializing `lp` and `lp2` with extracted addresses, also shown in the example above.

The invocation of assignment or compound assignment operators on built-in types pro-

other features in the same way. In C++17, however, changes are coming that leverage the distinction to cause a *prvalue* returned from a function to be used to initialize the return value directly, with no intervening temporary that might be inconsistently elided. The distinction between the two *rvalue* subcategories facilitates **guaranteed copy elision**, introduced to C++17 through P0135R0 (**?**) since *prvalues* are those expressions that can be used to initialize an object directly instead of indirectly creating a temporary that will then be used to initialize the object through the invocation of copy or move constructors.

duces an *lvalue* as does the invocation of any user-defined function or operator that returns an *lvalue* reference:

```
struct S
{
    S& operator+=(const S&);  // operator returning lvalue reference
};

int& h();   // h() is an lvalue of type int
S& j();     // j() is an lvalue of type S

void testAssignment()
{
    int x = 7;
    x = 5;     // x = 5 is an lvalue of type int.
    x *= 13;   // x *= 13 is an lvalue of type int.
    h() = x;   // h() = x is an lvalue of type int.
    S(s) = s;  // S(s) = s is an lvalue of type S.
    j() = s;   // j() = s is an lvalue of type S.

    S s;
    s += s;    // s += s is an lvalue of type S.
}
```

Just like for *lvalue* references, a pointer cannot be created to point to an *rvalue* reference. Pointers themselves, however, can be of any value category, and a reference of either variety *to* a pointer can be created. However, dereferencing any non**void** pointer, regardless of the value category of the pointer, produces an *lvalue*:

```
int* pf();  // pf() is a prvalue of type int*.
void testDereference()
{
    *pf();  // *pf() is an lvalue of type int.

    int* p = pf();  // p is an lvalue of type int*.
    *p;             // *p is an lvalue of type int.
}
```

Finally, the name of a variable is an *lvalue*, independent of whether that variable is a reference, even if it is itself an *rvalue* reference:

```
void testNames()
{
    int x     = 17;
    int& xr   = x;
    int&& xrv = std::move(x);

    x;    // x   is an lvalue of type int
    xr;   // xr   " "      "     "   "    "
    xrv;  // xrv  " "      "     "   "    "
}
```

prvalues-in-c++11/14

**Prvalues in C++11/14** A ***prvalue***, also called a pure *rvalue*, expression represents a value that is not necessarily associated with an object in memory and definitely cannot have its address taken without a conversion that creates an object in memory. Such conversions create **temporary objects** that are destroyed at the end of the outermost enclosing expression or, if bound to a reference satisfying the rules of **lifetime extension**, when the reference to which they are bound ends its own lifetime; see *Appendix* — **??** on page **??**. [**AUs: there is no subsection called "lifetime extension"; did you mean "Lifetime extension of a temporary bound to a reference"?**] There is no requirement for the compiler to **materialize** a *temporary* underlying object representation unless and until one is needed (e.g., to be mutated, to be moved from, or to be bound to a named reference), and that need is fulfilled by converting the *prvalue* to an *xvalue*, possibly creating a temporary object in the process. One other property particular to a *prvalue* expression is that it must be of **complete type**, i.e., sufficient to determine any underlying object's **size** using the built-in **`sizeof`** operator.

Literals are all *prvalue* expressions:

```
void testLiterals()
{
    5;        // 5    is a literal prvalue of type int.
    1.5;      // 1.5  is a literal prvalue of type double.
    '5';      // '5'  is a literal prvalue of type char.
    true;     // true is a literal prvalue of type bool.
}
```

Enumerators are also *prvalues*:

```
enum E { B };        // B is a named prvalue of type E.
```

The results of numeric operators on built-in types are also all *prvalues*:

```
void testExpressions()
{
    const int x = 3;     // x is a named, nonmodifiable lvalue of type int.
    int y = 4;           // y is a named, modifiable lvalue of type int.

    3 + 2;  // 3 + 2 is a compile-time prvalue of type int.
    x * 2;  // x * 2 is a compile-time prvalue of type int.
    y - 2;  // y - 2 is a prvalue of type int.
    x / y;  // x / y is a prvalue of type int.

    x && y;    // x && y is a prvalue of type bool.
    x == y;    // x == y is a prvalue of type bool.
    &x;        // &x     is a prvalue of type const int*.
    int(x);    // int(x) is a prvalue of type int.
}
```

Functions that return by value and explicit temporaries of user-defined types are also *prvalues*:

```
struct S { } s;  // s is a named, modifiable lvalue of type S.
```

```
int f();         // f is a named, nonmodifiable lvalue of type int(*)().
S g();           // g is a named, nonmodifiable lvalue of type S(*)().

void testCalls()
{
    S();  // S() is a prvalue of type S.
    f();  // f() is a prvalue of type int.
    g();  // g() is a prvalue of type S.
}
```

xvalues-in-c++11/14

**Xvalues in C++11/14** An ***xvalue***, also known as an expiring value, expression is entirely new in C++11. The most salient difference between an *xvalue* and a *prvalue* is that an *xvalue* expression is *guaranteed* to already be represented by an underlying object that resides within the virtual memory space of the running process; no such guarantee exists for a *prvalue* expression, leaving its internal representation an implementation detail of the compiler.

*Xvalue* expressions arise when an *lvalue* is explicitly cast to an *rvalue* reference:

```
void testXvalues()
{
    int x = 9;
    static_cast<int&&>(x);  // static_cast<int&&>(x) is an xvalue of type int.
    const_cast<int&&>(x);   // const_cast<int&&>(x) is an xvalue of type int.
    (int&&) x;              // (int&&) x is an xvalue of type int.
}
```

Functions and operators that return an *rvalue* reference produce *xvalues* when invoked:

```
int&& f();  // f() is an xvalue of type int.
S&& g();    // g() is an xvalue of type S.

S&& operator*(const S&, const S&);  // oddly defined operator

void testOperator()
{
    int i, j;
    i * j;  // i * j is a prvalue of type int.
    S a, b;
    a * b;  // a * b is an xvalue of type S.
}
```

The Standard Library utility function, `std::move`, also produces *xvalues*, as it is nothing more than a function defined to return an *rvalue* reference to the type passed to it; see *The* `std::move` *utility* on page 20.

Finally, expressions that access subobjects of any non*lvalue* are *xvalues*, including non-static data member access, array subscripting, and dereferencing pointers to data members. Note that when any of these operations are applied to a *prvalue*, a temporary needs to be created from that *prvalue* to contain the subobject, so the subobject is an *xvalue*[4]:

[4]The identification of subobjects as *xvalues* rather than *prvalues* or, in some cases, *lvalues*, has been

```
struct C  // C() is a prvalue of type C.
{
    int d_i;
    int d_arr[5];
};

C&& h();                // h() is an xvalue of type C.
int C::* pd = &C::d_i;  // pointer to C::d_i

void testSubobjects()
{
    h().d_i;       // h().d_i is an xvalue of type int.
    C().d_i;       // C().d_i  "  "     "     "   "    "
    h().d_arr;     // h().d_arr is an xvalue of type int[5].
    C().d_arr;     // C().d_arr  "  "     "     "   "     "
    h().d_arr[0];  // h().d_arr[0] is an xvalue of type int.
    C().d_arr[0];  // C().d_arr[0]  "  "     "     "   "    "
    h().*pd;       // h().*pd is an xvalue of type int.
    C().*pd;       // C().*pd  "  "     "     "   "    "
}
```

rvalue-references

## *Rvalue* references

C++11 introduced *rvalue* references, a new reference type that uses a double ampersand, &&, as part of its syntax (e.g., **int**&&):

```
int&& r = 5;  // r is an rvalue reference initialized with a literal int 5.
```

The goal of expanding the type system to include *rvalue* references is to allow function overloading on values that are safe to be moved from, i.e., non*lvalues*. What distinguishes an *rvalue* reference from the familiar *lvalue* reference is that an *rvalue* reference will bind to *only* non*lvalues*. When initialized with an *xvalue*, an *rvalue* reference is bound to the object identified by that *xvalue*. When initialized with a *prvalue*, a temporary is implicitly created that will have the same lifetime as the *rvalue* reference, and the reference is bound to that temporary.

We can now exploit value categories to show exactly what binds to what in the presence of a **const** qualifier[5]:

```
      int        fi();       // returns prvalue of type         int
const int       fci();       // returns prvalue of type const int
      int&    flvri();       // returns  lvalue of type         int
const int&   fclvri();       // returns  lvalue of type const int
      int&&   frvri();       // returns  xvalue of type         int
```

the subject of a number of core issues, all of which were accepted as **defect reports** between C++14 and C++20. Specifically, CWG issue 616 (**?**) and CWG issue 1213 (**?**) both dealt with changes to the value categories of subobject expressions. Also note that compiler implementations of these clarifications took some time, with GCC not fully supporting them until GCC 9.

[5] If we were to include **volatile**, we would double the number of possible qualifier combinations that could, in theory, be applied to each of the three *value categories*; since we know of no practical application for **volatile**, we leave combinations involving it as an exercise for the reader.

```
const int&&  fcrvri();       // returns  xvalue of type const int

int&& r0 =     fi();         // OK
int&& r1 =    fci();         // Error, cannot bind to const prvalue.
int&& r2 =  flvri();         // Error, cannot bind to          lvalue.
int&& r3 = fclvri();         // Error, cannot bind to const  lvalue.
int&& r4 =  frvri();         // OK
int&& r5 = fcrvri();         // Error, cannot bind to const  xvalue.

const int&& cr0 =     fi();  // OK
const int&& cr1 =    fci();  // OK
const int&& cr2 =  flvri();  // Error, cannot bind to          lvalue.
const int&& cr3 = fclvri();  // Error, cannot bind to const  lvalue.
const int&& cr4 =  frvri();  // OK
const int&& cr5 = fcrvri();  // OK
```

An *rvalue* returned from a function can be modified provided (1) it is of user-defined type and (2) there exists a mutating member function; see *Appendix — Modifiable rvalues!* on page 92. Assignment to fundamental types is not permitted for *rvalues*:

```
struct V
{
    int d_i;                                          // public int member
    V(int i) : d_i(i) { }                             // int value constructor
    V& operator=(int rhs) { d_i = rhs; return *this; } // assignment from int
};
      V  fv(int i) { return V(i); }  // Returns nonconst prvalue of type V.
const V fcv(int i) { return V(i); }  // Returns    const prvalue of type V.

void test1()
{
    fv(2).d_i = 5;       // Error, cannot assign to rvalue int
    fv(2).operator=(5);  // OK, member assignment can be invoked on an rvalue.
    fv(2) = 5;           // OK,    "        "      "   "    "     "  "     "

    fcv(2).d_i = 5;      // Error, cannot assign to const rvalue int
    fcv(2).operator=(5); // Error, assignment is a nonconst member function.
    fcv(2) = 5;          // Error,      "      " "  "    "       "       "
}
```

What's more, that modified value will be preserved until the end of the outermost expression containing the *rvalue* subexpression:

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

void test2()
{
    int x = 1 + (V(0) = 2).d_i + 3 + (fv(0) = 4).d_i + 5;
    assert(15 == x);       // 15 == 1 + 2 + 3 + 4 + 5
}
```

The ability to modify an unnamed, temporary *rvalue* directly, however, is distinct from that of modifying a named, nontemporary *rvalue* reference, which is itself an *lvalue*. Binding any *rvalue* — such as a default-constructed object, e.g., `S()`, or a literal, e.g., `1` — to a **`const`** *lvalue* reference was always possible in C++, but subsequently modifying it through that reference was not:

```
void test3()
{
      S&  lvrs = S();      // Error, cannot bind to a nonconst lvalue ref
const S& clvrs = S();      // OK, can bind any value to a const lvalue ref
         clvrs = S();      // Error, cannot modify via a const lvalue ref
const S* pcs1  = &clvrs;   // OK, any const lvalue reference is an lvalue.

      int&  lvri = 5;      // Error, cannot bind to a nonconst lvalue ref
const int& clvri = 5;      // OK, can bind any value to a const lvalue ref
           clvri = 5;      // Error, cannot modify via a const lvalue ref
const int* pci1  = &clvri; // OK any const lvalue reference is an lvalue.
}
```

Notice, however, that each of the named references in the example above, when used in an expression, is itself an *lvalue*, and hence it is possible for the programmer to use the built-in unary address-of operator, `&`, to take the address of the possibly temporary underlying object that initialized it.

*Rvalue* references behave similarly in that, when a *prvalue* is assigned to a named *rvalue* reference, it too becomes an object whose lifetime is the same as the temporary to which the reference is bound. Unlike a non**`const`** *lvalue* reference, however, a non**`const`** *rvalue* reference can be bound to a non**`const`** *rvalue* of the same type, in which case that same underlying object can be subsequently modified via the reference:

```
void test4()
{
      S&&  rvrs = S();    // OK, can bind a nonconst rvalue to an rvalue ref
const S&& crvrs = S();    // OK, can bind any rvalue to a const rvalue ref
           rvrs = S();    // OK, can modify via a named nonconst rvalue ref
          crvrs = S();    // Error, cannot modify via any const rvalue ref
      S* ps2  = &rvrs;    // OK, a named nonconst rvalue ref is an lvalue.
const S* pcs2 = &crvrs;   // OK, a named const rvalue ref is an lvalue.

      int&&  rvri = 5;    // OK, can bind a nonconst rvalue to an rvalue ref
const int&& crvri = 5;    // OK, can bind any rvalue to a const rvalue ref
             rvri = 5;    // OK, can modify via a named nonconst rvalue ref
            crvri = 5;    // Error, cannot modify via any const rvalue ref
      int* pi2  = &rvri;  // OK, a named nonconst rvalue ref is an lvalue.
const int* pci2 = &crvri; // OK, a named const rvalue ref is an lvalue.
}
```

Recall that *rvalue* references were invented to allow library developers to discriminate — using overloaded functions — between *movable* arguments, i.e., those that can be moved from safely, and nonmovable arguments. For this reason, it was necessary to ensure that an *rvalue* reference of a given type never implicitly binds to an *lvalue* of corresponding type,

as allowing such liberal binding would result in move operations cannibalizing the state of objects that had not been identified as ready for such activity. So while a **`const`** *lvalue* reference binds to *all* values, a **`const`** *rvalue* reference is deliberately designed *not* to bind to an *lvalue* of the *same* type but to happily bind to an *rvalue* that is the result of an implicit conversion, including an **integral promotion**, from any other type:

```
double d;                       // d is a named lvalue of type double.
const double cd = 0;            // cd is a named lvalue of type const double.
const double fcd();             // fcd returns a prvalue of type const double.

const double& clr1 = d;         // Initialize with lvalue of type double.
const double& clr2 = cd;        // Initialize with lvalue of type const double.
const double& clr3 = double();  // Initialize with prvalue of type double.
const double& clr4 = fcd();     // Initialize with prvalue of type const double.

const double&& crr1 = d;        // Error, cannot bind to lvalue of same type
const double&& crr2 = cd;       // Error, cannot bind to lvalue of same type
const double&& crr3 = double(); // Initialize with prvalue of type double.
const double&& crr4 = fcd();    // Initialize with prvalue of type const double.

float f;                        // f is a named lvalue of type float.
const float cf = 0.0;           // cf is a named lvalue of type const float.
const float fcf();              // fcf returns a prvalue of type const float.

const double&& cfr1 = f;        // OK, f is converted to rvalue of type double.
const double&& cfr2 = cf;       // OK, cf is converted to rvalue of type double.
const double&& cfr3 = float();  // Initialize with prvalue of type double.
const double&& cfr4 = fcf();    // Initialize with prvalue of type const double.

short z;                        // z is a named lvalue of type short int.
const int&& czr = z;            // OK, z is promoted to an rvalue of type int.
```

If the type of the *rvalue* initializing a non**`const`** *rvalue* reference is *not* the same type as that of the reference, a new underlying object will be materialized; any subsequent modification via the reference will have no effect on the original value. Note that, in the example below, we have used the `std::move` utility to cast the *lvalues* `c` and `u` to their respective *xvalue* —i.e., unnamed *rvalue* reference — counterparts, which are now treated as *rvalues* and are therefore *movable*; see *The* `std::move` *utility* on page 20:

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

void test4()
{
    char          c  = 1;
    unsigned char u  = 2;

    char&& rc = std::move(c);  // rc refers to c.
    char&& ru = std::move(u);  // temporary char created

    assert(&rc == &c);  // rc refers to c.
```

```
    assert(&ru != &u);  // ru refers to a different char.

    rc = 7;  assert(7 == c);  // c modified through rc
    ru = 8;  assert(2 == u);  // u unchanged
}  // Temporary lifetime ends with all other variables here.
```

ding-on-reference-types

**Overloading on reference types** The entire purpose of having a new kind of reference that binds exclusively to *rvalues* is so that existing overload sets employing a **`const`** *lvalue* reference can be augmented so that, when the same named function is invoked with a non*lvalue*, the new overload will be given preference, which can then move from its argument. For example, consider a function, e.g., `g` in the example below, that takes an object of user-defined type `C`, which it *might* need to copy and manipulate internally before returning a value, e.g., an **`int`**[6]:

```
class C { /*...*/ };  // some UDT that might benefit from being "moved"

int g(const C& c);  // (1) [original] takes argument by const lvalue reference.
int g(C&& c);       // (2) [additional] takes argument by nonconst rvalue ref.
```

Let's now consider calling this function `g` on expressions having various kinds of values, e.g., *lvalues* versus *rvalues* and **`const`** versus non**`const`**:

```
      C c;     // c is a named lvalue of type C.
const C cc;    // cc is a named lvalue of type const C.
const C fc();  // fc() is an unnamed rvalue of type const C.

int i1 = g(c);     // OK, invokes overload g(const C&) because c is an lvalue
int i2 = g(C());   // OK, invokes overload g(C&&) because C() is a prvalue
int i3 = g(cc);    // OK, invokes overload g(const C&) because cc is const
int i4 = g(fc());  // OK, invokes overload g(const C&) because fc() is const
```

In this scenario, if an argument to `g` in the code snippet above is a non**`const`** *rvalue* (and therefore known to be movable and potentially a *temporary*), it will bind more strongly to `g(C&&)`. Non*rvalue* and **`const`** arguments to `g` will not consider `g(C&&)` at all but are viable to pass to `g(const C&)`, so that overload will be chosen. Should either overload be chosen with a *prvalue*, a temporary will be created that will then be destroyed at the end of the invoking expression.

Note that adding an *rvalue* reference overload of `g` does not add to the set of usable arguments for `g`. That is, anything that can bind to a `C&&` can also bind to a **`const`** `C&`. A corollary to this observation is that, given any function having an overload set that contains a **`const`** `T&` parameter, one can *safely* introduce a parallel overload having a `T&&` parameter in the corresponding position with the desired effect that, when that function is called with an argument that is safely movable in that position, the newly added *rvalue*-reference overload will be called instead.

In the most general case, there are six — twelve if you consider **`volatile`** — combinations of ways to pass an argument to a function (see Table 1).

[6]Note that, even if we were sometimes inclined to pass user-defined types by value, we would *not* want to do so here for fear we might end up making an expensive copy for no reason; see *Passing movable objects by value* on page 55.

**Table 1: AUs: Pls add a caption.**

rvalueref-table1

| | `nonconst` | `const` |
|---|---|---|
| value | `T` | **`const`** `T` |
| *lvalue* reference | `T&` | **`const`** `T&` |
| *rvalue* reference | `T&&` | **`const`** `T&&` |

One could, in principle, overload a function such as `g` in Table 1 with all six variants, but doing so would lead to ambiguity as passing *by value*, `T`, and passing *by lvalue reference*, `T&`, are equivalently good matches as are passing by value, `T`, and passing by **`const`** value, **`const`** `T`. Hence, in practice, one passes an object either by value (or **`const`** value) or else some subset of the four possible reference variants (see the example below). Having such flexibility, however, is seldom useful.

Consider, for example, a function `h` that is overloaded on a non**`const`** *lvalue* and a non**`const`** *rvalue*:

```
void h(C&  inOut);   // (3) OK, accepts only nonconst lvalues
void h(C&& inOnly);  // (4) OK, accepts only nonconst rvalues
```

As was the explicit intent of the inventors of *rvalue references*, an overload set like `h` provides the ability to distinguish programmatically between non**`const`** objects that are (1) *lvalues*, and therefore are *not* known to be safe to be *moved from*, and (2) non*lvalues*, and therefore *are* presumed to be safe to be *moved from*, yet exclude any object whose type is qualified with **`const`**:

```
void test(C& lv, const C& clv, const C&& crv)
{
    h(lv);              // (5) OK, invokes (3) because lv is an lvalue
    h(C());             // (6) OK, invokes (4) because C() is an rvalue
    h(clv);             // (7) Error, clv is const. No overload of h matches.
    h(std::move(crv));  // (8) Error, crv is const. No overload of h matches.
}
```

But consider that having such an overload set is typically counterindicated. Without the *rvalue* reference overload, invoking `h` on a *temporary* would simply fail to compile, thereby avoiding a runtime defect. With that *rvalue*-reference overload present, the code will in fact compile, but now any output written to that *temporary* will silently disappear along with that temporary.

Although seldom needed, we provide in Table 2 for completeness the relative priority in which pass-by-reference members of an overload set would be selected.

**Table 2: AUs: pls add a caption**

rvalueref-table2

| Value Category | E.g. | g(C&) | g(const C&) | g(C&&) | g(const C&&) |
|---|---|---|---|---|---|
| non**const** *lvalue* | c | 1 | 2 | N/A | N/A |
| **const** *lvalue* | cc | N/A | 1 | N/A | N/A |
| non**const** *rvalue* | C() | N/A | 3 | 1 | 2 |
| **const** *rvalue* | fc() | N/A | 2 | N/A | 1 |

Note that the equivalent function to `fc` for a built-in type, **`const int fi()`**, would return a non**const** *rvalue*, as fundamental types returned by **const** value are treated as if they had been returned by non**const** value. The only way to have a function that returns a **const** *rvalue* of primitive type `T` is to have it return **`const T&&`**, such as **`const int&& fi2()`**.

ferences-in-expressions

**Rvalue references in expressions** Recall from *Lvalues in C++11/14* on page 9 that the name of any variable, including an *rvalue* reference, is itself an *lvalue*:

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

struct S { };

void test()
{
    S   s1;                 // local variable of type S
    S&& s2 = s1;            // Error, s1 is an lvalue.
    S&& s3 = std::move(s1); // OK, std::move(s1) is an xvalue.
    S&& s4 = s3;            // Error, s3 is an lvalue.
    S&& s5 = std::move(s3); // OK, std::move(s3) is an xvalue.
}
```

This important fact helps to ensure that a reference is not accidentally used as an *xvalue* prior to the last time that its value is needed:

```
void f(const S& s);  // only reads the value of s
void f(S&& s);       // consumes the value of s

void test2(S&& s)
{
    f(s);            // invokes int f(constS&)
    f(std::move(s)); // invokes int f(S&&)
}
```

While unintuitive at first glance, it is important to consider how the function `test2`, in the example above, would behave if an *rvalue* reference were itself an *rvalue*. The invocation `f(s)` would invoke **`void f(S&&)`**, which would result in consuming the value of `s` and leaving it in an unspecified state. The subsequent invocation of `f(std::move(s))` would then be attempting to process a moved-from object, which almost certainly would not be the intent of the writer of `test2`.

The final use of an *rvalue* reference in a given context could conceivably be treated as an *xvalue*, but up to this point in the evolution of C++, that has been done only in

the relatively narrow case where variables having **automatic storage duration** are used in **return** statements; see *Implicit moves from lvalues in* **return** *statements* on page 23. Additional implicit moves might be added to the language in the future but only if it the proposals brought to the Standards committee make a convincing case that such additions carry little or no risk of silently doing harm.

the-std::move-utility

**The `std::move` utility** A large part of the original motivation for having a new value category, dubbed *xvalues*, was to intersect the notions of *reachability* and *movability*; see Appendix - Why do we need another value category?. The way we typically do that, e.g., in the case of moving an element in a vector from one position to another, is to cast a pre-existing *lvalue* to an *rvalue* reference:

```
struct S { };  // some UDT that might benefit from being moved

int f(const S&  s);  // (1) takes any kind of S but with lower priority
int f(      S&& s);  // (2) takes only movable kinds of S with high priority

S s;  // s is a named lvalue; (s) is a nonconst lvalue-reference expression.

int i1 = f(s);                  // calls (1); can copy-construct local S from s
int i2 = f(const_cast<S&&>(s)); // calls (2); can move-construct local S from s
```

Note that use of a **const_cast** would allow even a **const** S to be converted to a non**const** *rvalue* reference, while a **static_cast** would enable a conversion from a type convertible to S to be bound to an *rvalue reference*, either of which would typically have unintended consequences. Given that an essential part of the design of *rvalue* references involves a specialized *cast* that must preserve the C++ type and **const**ness, while changing the value category, it was decided that the best approach to performing the conversion was a library utility function template that deduced the C++ type, ignoring reference qualifiers, and then used a **static_cast**, as opposed to a **const_cast**, in its implementation. One plausible implementation — using forwarding references, **constexpr**, and the **noexcept** decorator — is provided here for concreteness; see Section 2.1."**??**" on page **??**, ."**??**" on page **??**, **[AUs: there is no feature called constexpr keyword; we have constexpr function, constexpr variable, and relaxed constexpr]** and Section 3.1."**??**" on page **??**:

```
// utility:
namespace std
{
    template <typename T> struct __RemoveReference      { typedef T type; };
    template <typename T> struct __RemoveReference<T&>  { typedef T type; };
    template <typename T> struct __RemoveReference<T&&> { typedef T type; };

    template <typename T>
    constexpr typename __RemoveReference<T>::type&& move(T&& expression) noexcept
    {
        return static_cast<typename RemoveReference<T>::type&&>(expression);
    }
}
```

As the example implementation above indicates, the name of this standard function template is `move`, it resides in the `std` namespace, and it can be found in the Standard `<utility>` header file. This function is used just as one would use a **`static_cast`** to convert an *lvalue*-reference expression to an unnamed, *rvalue*-reference expression, except that the C++ type is deduced automatically rather than giving the user the opportunity to accidentally mis-specify it:

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

S t1, t2;  // two similar objects that are each movable

int i3 = f(static_cast<S&&>(t1));  // can move-construct a local S from t1
int i4 = f(std::move(t2));         // can move-construct a local S from t2
```

The choice of *move* for the name of this specialized cast to *rvalue* reference can, however, be confusing; see *Annoyances* — `std::move` *does not move* on page 79.

per-function-generation

**Special member function generation** Given the new capability of having additional function overloads for *rvalue* reference parameters, two new special member functions were also added in C++11.

1. A move constructor for a type, `X`, is a constructor that can be invoked with a single *rvalue* reference to `X`. There are two requirements: (1) The first parameter of the constructor must be a **cv-qualified** *rvalue* reference to `X`, i.e., either `X&&`, **`const`** `X&&`, **`volatile`** `X&&`, or **`const volatile`** `X&&`, and (2) the constructor must have exactly one parameter or all parameters after the first must have default values:

```
struct S1 { S1(S1&&); };             // move constructor
struct S2 { S2(const S2&&); };       //  "        "
struct S3 { S3(S3&&, int i = 0); };  //  "        "
struct S4 { S4(S4&&, int i); };      // not a move constructor
struct S5 { S5(int&&); };            //  "  "  "        "
struct S6 { S6(S6&); };              //  "  "  "        "
```

2. A move-assignment operator for a type `X` is a nonstatic, nontemplate member function named **`operator=`** with exactly one parameter that is a cv-qualified *rvalue* reference to `X`, i.e., either `X&&`, **`const`** `X&&`, **`volatile`** `X&&`, or **`const volatile`** `X&&`. Any return type and value is valid for a move-assignment operator, but the common convention is to have a return type of `X&` and to return **`*this`**.

Just as with other special member functions, when not declared explicitly it is possible for the move constructor and move-assignment operator to be implicitly declared for a **`class`**, **`struct`**, or **`union`**.

- An implicitly declared move constructor will be declared if there are no user-declared copy or move constructors, no user-declared copy or move-assignment operators, and no user-declared **destructor**.[7] The implicit move constructor for a type, `X`, will have

[7] For more on the implicit generation of special member functions, see Section 1.1."**??**" on page **??**.

the signature `X::X(X&&)`. The exception specification and triviality of this constructor is determined by the element-wise move constructors of all bases and nonstatic data members of `X`, and the definition will do a memberwise move-construction of all bases and nonstatic data members.

- An implicitly declared move-assignment operator will be declared if there are no user-declared copy or move constructors, no user-declared copy or move-assignment operators, and no user-declared destructor. The implicit move-assignment operator for a type, `X`, will have the signature `X& operator=(X&&)`. The exception specification and triviality of this operator is determined by the element-wise move-assignment operators of all bases and nonstatic data members of `X`, and the definition will do a memberwise move-assignment of all bases and nonstatic data members.

-in-return-statements

**Moves from rvalues in `return` statements** When the expression returned from a function is a *prvalue*, the return value will be initialized via move construction:

```
struct S { };

S f1() { return S{}; }  // returns a prvalue

void test1()
{
    S s = f1();
}
```

The local variable `s` in `test1` above may be initialized in one of two ways. At a minimum, a temporary will be created within the body of `f1` and used as an *rvalue* to copy construct `s` within `test1`. The example is a perfect example for what has come to be known as **return-value optimization (RVO)**. Instead of constructing the object within the function and then copying — or even moving — it to the caller, space for the **footprint** of the object is reserved in the caller's scope, and then the object is constructed just once, **in place**, and is initialized by the returned *prvalue*; see *Potential Pitfalls* — **??** on page **??**. [**AUs: there is no Pitfalls subsection called "RVO and NRVO affect observable behavior"; what did you intend?**] This reduces to zero the number of additional constructor calls beyond the initial call, the best number of operations to invoke from the perspective of the performance-minded.[8]

Return values can be move-constructed when the returned expression is an *xvalue* as well:

```
S f2()
{
    S s;
    return std::move(s);  // returns an xvalue
}
```

In this case, the object initialized by the return value of this function will be constructed with an *rvalue* reference to `s`. It is important to note, however, that explicitly making such

[8] C++17 mandates that there be no additional object created when a function returns a *prvalue*, effectively guaranteeing **copy elision**.

*lvalues* into *xvalues* on return might not be needed and, in some cases, might prevent **NRVO**; see *Implicit moves from lvalues in* **`return`** *statements* on page 23.

Finally, if a *prvalue* expression is to be returned, it is strongly contraindicated to use `std::move` to force a move from that *prvalue*:

```
S f3()
{
    return std::move(S{});  // BAD IDEA
}
```

Applying `std::move` as in `f3` in the example above does more than just cast an expression to a particular type; it requires that there be an object to pass to the utility function. In this case, a temporary will have to be materialized to be the argument to `std::move`, and there will no longer be the possibility of eliding all moves and applying RVO.

es-in-return-statements

**Implicit moves from lvalues in `return` statements** Any time the use of an object coincides with the last time that object can be referenced, it might be beneficial to treat that object as an *xvalue* instead of an *lvalue* so we can reuse the resources owned by that object before they are released by the destructor. As we have seen, temporaries created by an expression will automatically be treated as *xvalues* and are one of the primary means by which *xvalues* are generated by expressions.

In a **`return`** statement, the lifetime of all variables with automatic storage duration — namely, non**`volatile`** local variables and function parameters — ends implicitly after the statement. This enables any **`return`** statement that names such a variable to treat the returned expression as an *xvalue* instead of an *lvalue*, resulting in move construction of the return value instead of copy construction[9]:

```
struct S { };
void g(const S&);
void g(S&&);

S f1()
{
    S s;
    g(s);      // s is not an xvalue and selects g(const S&).
    return s;  // s is an xvalue.
}

S f2(S in)
{
```

[9] C++20 introduced the concept of an **implicitly movable entity** for those automatic storage duration objects that will implicitly be moved when returned in this way. This concept was also extended to include *rvalue* references (to non**`volatile`** objects):

```
S f3(S&& in)
{
    return in;  // in is an lvalue in C++11-17 and an xvalue in C++20.
}
```

```
    g(in);      // in is not an xvalue and selects g(const S&).
    return in;  // in is an xvalue.
}
```

This conversion to *xvalue* broadly overlaps the cases where an automatic variable is eligible for copy elision, otherwise known as the named return-value optimization. When an automatic variable has the same type as the return type of a function and when that variable is named as the expression of a **`return`** statement, the copy implied by the **`return`** statement can be omitted, and instead only one object will be constructed in the footprint of the target of the return value:

```
S* s1Addr = nullptr;

S nrvo1()
{
    S s1;
    s1Addr = &s1;   // store address of automatic variable
    return s1;
}

void callNrvo1()
{
    S s2 = nrvo1();
    assert(s1Addr == &s2);  // implementation dependent, not guaranteed
}
```

This optimization is not guaranteed and might be dependent on optimization levels and properties of the type in question; see *Potential Pitfalls* — **??** on page **??**. [**AUs: there is no Pitfalls subsection called "RVO and NRVO affect observable behavior"; what did you intend?**]

Functions that are eligible for **NRVO** also meet the requirements for move-constructing their return values, even if NRVO does not take effect. This means that, at worst, functions like `f1` and `nrvo1` in the code snippet above will invoke a single move constructor to initialize their return values and, at best, will invoke nothing at all. Compare this to the following function:

```
S f4()
{
    S s;
    return std::move(s);
}
```

Here, by explicitly using `std::move` to move from the local variable to the return value, we have also made this function no longer eligible for NRVO, as the return expression is a function call and not just the name of a local variable. This results in *always* invoking a move constructor to initialize the return value and never eliding the additional object at all.

In general, when an object's lifetime is about to end and it will potentially be treated implicitly as an *xvalue*, it is a reasonable, relatively future-proof, rule of thumb to *not* explicitly use `std::move`. In current code, the move will happen either way, and in future

Standards the extra object might be elided completely as the language evolves.[10]

When the type of a local variable used in a **`return`** statement does *not* exactly match the nonreference C++ type being returned, then the explicit use of `std::move` *is* indicated: A conversion to the return type will always occur in that case. The explicit cast to an *rvalue* reference guarantees that the result of the conversion will be created by *moving from* the object named in the **`return`** statement. Note that some types of *move conversions*, e.g., those resulting from *move construction*, would happen even without the explicit use of `std::move` but there is, however, no situation where having an explicit `std::move` pessimizes the creation of the result object, but see *Potential Pitfalls* — **??** on page **??**. [**AUs: there is no section within Pitfalls called "Pessimizing moves"; what did you intend?**]

When something more complicated than the name of an automatic variable is used as the expression of a **`return`** statement, the explicit use of `std::move` is also indicated, as none of the rules that might enable copy elision are potentially applicable:

```
S f5(bool flag)
{
    S a, b;
    return flag ? std::move(a) : std::move(b);  // std::move needed
}

S f6()
{
    S a;
    return 1, std::move(s);  // std::move needed
}
```

n-to-move-return-values

**When to move return values** What can a poor programmer do to ensure that (1) a copy is always avoided where possible and (2) copy elision is never pessimized by an explicit cast?

Following six simple rules of thumb will achieve both of these goals now, and the resulting code should remain future-proof in perpetuity.

1. First, do no harm. If there is a possibility that a value might subsequently be referenced from some other part of the program, then an explicit cast is *not* indicated:

```
S f1(S* s)
{
    return *s;  // explicit cast not indicated
}

S s1;  // externally reachable variable

S g1()
{
```

[10] Proposals such as **?** which makes NRVO apply more broadly and be guaranteed in a fashion similar to RVO in C++17, **?**, which added implicitly movable entities to C++20, and **?** which improves the handling of implicitly movable entities, all lead to improvements to functions that do not explicitly move, while making no changes or improvements to functions that explicitly move.

```
    return s1;  // explicit cast not indicated
}
```

Moving from such an object might result in unintended and possibly incorrect behavior.

2. If the expression being returned is a *prvalue* (e.g., the anonymous construction of an object or a call to a function returning an object by value), then an explicit cast is *not* indicated:

```
S f2()
{
    return S(/*...*/);  // explicit cast not indicated
}

S g2()
{
    return f2(/*...*/);  // explicit cast not indicated
}
```

It is precisely these cases in which the unnamed, return-value optimization (RVO) is likely to kick in, thereby eliding even move construction; providing an explicit cast would effectively disable RVO, forcing two objects to be created instead of just one.

3. If the expression in a **`return`** statement consists of just the name of a locally declared nonreference variable, including a by-value parameter or one in a local **`catch`** clause, and exactly matches the **`return`** type of the function, an explicit cast is *not* indicated:

```
S f3()
{
    S s;
    return s;  // explicit cast not indicated
}

S g3(S s)
{
    return s;  // explicit cast not indicated
}

S h3()
{
    try
    {
        // ...
    }
    catch (S s)
    {
        return s;  // explicit cast not indicated
    }
}
```

In some cases, NRVO will cause local objects to be constructed directly in the result; otherwise, an implicit cast is guaranteed so the object will move if the type supports move construction. Although the argument of a **`catch`**-clause parameter will never be optimized away, it too will be moved implicitly, and attempting to explicitly cast it unnecessarily might even trigger compiler warnings.

4. If the expression in a **`return`** statement consists of the name of a locally declared nonreference variable that does not exactly match the **`return`** type of the function, an explicit cast *is* indicated:

```
T f4()  // S is convertible to T.
{
    S s;
    return std::move(s);  // explicit cast is indicated
}
```

Not all move conversions are guaranteed to occur implicitly; since copy elision doesn't apply when the types do not match exactly, having an explicit cast cannot pessimize object creation but might enable a *move* instead of a *copy* operation.

5. If the expression being returned is any kind of reference and it is clear that it is appropriate to move the object being referenced, then an explicit cast *is* indicated:

```
S f5()
{
    S s;
    S& r = s;
    return std::move(r);  // Explicit cast is indicated.
}

S g5(S&& s)  // Being passed as an rvalue reference indicates s should be moved.
{
    return std::move(s);  // Explicit cast is indicated.
}
```

There are no implicit conversions from references; since copy elision doesn't apply either, having an explicit cast cannot pessimize object creation but might enable a move instead of a copy.[11]

6. Additionally, when an expression resides in a **`throw`** statement, an explicit cast *is* indicated but only when it is *safe* to do so:

```
void f6(S s)
{
    throw std::move(s);  // Explicit cast is indicated.
}
```

[11] While the *rvalue* reference parameter of `g5` will be implicitly moved in C++20, there is also no apparent downside here to explicitly requesting the move and avoiding changes in meaning with different language standards.

```
void g6()
{
    S s;
    S& r = s;
    throw std::move(r);  // Explicit cast is indicated.
}

void h6()
{
    S s;  // used below by f6

    try
    {
        throw s;  // Explicit cast is not indicated.
    }
    catch (...) { }

    f6(s);  // uses s after being thrown
}
```

An exception object will always be created, and, without an explicit cast, it is not guaranteed that a viable move constructor will be invoked, especially when the thrown expression is of reference type. There is no situation where using `std::move` pessimizes the creation of the exception object, whereas not using it might result in its being *copied* instead of *moved*. If, however, the thrown object were to be caught within the same function and later used, then an explicit cast would *not* be indicated.

## Summary

summary

The introduction of the extended value categories in C++ was a fundamental part of introducing move semantics into the language. While an understanding of the concepts of *lvalues* and *rvalues* is usually sufficient for most programming tasks, it is, in fact, a simplified model of the C++ value categories. Knowing the complete story can help programmers to understand move semantics in a deeper way, which can enable developers to write and understand more sophisticated code.

There are now two main categories, *glvalue* and *rvalue*, with three disjoint subcategories, *lvalue*, *xvalue*, and *prvalue*. One way to think of the new value categories is to note that C++ expressions have two independent properties: identity, or reachability, and movability. Objects are reachable if we can take their address. Objects are movable if they cannot be referenced or if we explicitly tell the compiler that it is safe to move them. *Lvalue* expressions have identity, we can take their address, and they are reachable. *Prvalues* have no identity and can be moved from. *Xvalues* have identity and can be moved from.

*Rvalue* references as function parameters in overload sets enable functions and, most importantly, constructors and assignment operators to distinguish between cases where they are passed an *lvalue* or an *rvalue* and decide based on which they received whether they can cannibalize the contents of their arguments to avoid expensive copy operations.

use-cases-rvalueref

## Use Cases

ptimizations-of-copying

### Move operations as optimizations of copying

With the introduction of *rvalue* references, we are now able to discriminate between a reference bound to an object that is eligible to be moved from, i.e., an *rvalue* reference, from that which is bound to an object that must be copied, i.e., an *lvalue* reference. That is, we are now able to **overload** our constructors and assignment operators for both *rvalues* and *lvalues*.

Constructing an object from an *lvalue* expression of the same type is called *copy construction*. Constructing an object from an *rvalue* expression of the same type is called *move construction*. The term *move* colloquially expresses the idea of transferring **owned resources**, typically dynamically allocated memory blocks, from one object to another as opposed to that of somehow duplicating, e.g., allocating and copying, them.

In most practical applications, *moving* can be reasonably interpreted and properly implemented as an *optimization* of *copying* in that the requirements on the target object are the same, whereas the requirements on the source object are relaxed such that it need not retain the same state — or **value**, where applicable — after the operation completes. Importantly, knowing that the semantics of *move* operations can be fully defined in terms of their corresponding *copy* operations makes them easy to understand. What's more, these newly added move-semantic operations can be tested using unit tests that are modified only slightly from their existing **copy-semantic** counterparts.

Repurposing internal resources from an object that no longer needs them can lead to faster *copy*-like operations, especially when dynamic allocation and deallocation of memory is involved. Countervailing considerations, such as **locality of reference**, can, however, suggest that preferring *move* operations to *copy* operations might, in some circumstances, be contraindicated for overall optimal runtime performance, especially at scale.[12]

Properly implementing *move* operations that, for **expiring** objects, act like optimized **copy operations** will depend on the specifics of how we chose to implement a type, e.g., an object that (1) is written to manage its own resources explicitly (see *Creating a low-level value-semantic type (VST)* on page 29) or (2) delegates resource management to its subobjects (see **??** on page **??**). [**AUs: there is no section called "UDTs having implicitly generated move operations"; what did you intend?**]

lue-semantic-type-(vst)

**Creating a low-level value-semantic type (VST)** Often, we want to create a **user-defined type (UDT)** that represents a value. When implemented properly, we refer to such a type generally as a **value-semantic type (VST)**. Although there are some cases where a VST might be implemented as a simple **aggregate type** (see UDTs having implicitly generated *move* operations), [**AUs: there is no section called "UDTs having implicitly generated *move* operations"; what did you intend?**] there are other cases where a VST might, instead, manage its internal resources directly. The latter explicit implementation of a VST is the subject of this subsection.

For illustration purposes, consider a simple VST, **`class`** `String`, that maintains, as an **object invariant**, a null-*terminated* string value; i.e., this string class explicitly does not support having a *null* pointer value. In addition to a *value* constructor and a single **`const`**

[12] **?**

member function to access the value of the object, each of the four C++03 special member functions — default constructor, copy constructor, assignment operator, and destructor — are **user provided**, i.e., defined explicitly by the programmer. To keep this example focused, however, we will *not* store separately the length of the string, and we'll omit the notion of *excess* capacity altogether, leaving just a single non**static const char*** data member, `d_str_p`, to hold the address of the dynamically allocated memory:

```
class String { const char *d_str_p; /*...*/ };  // null-terminated-string manager
```

One practical aspect that we preserve is that default-constructed container types — going back to C++98 — are well advised, on purely performance grounds, never to pre-allocate resources, lest creating a large array of such empty containers be impracticably runtime-intensive to construct:

```
String s;  // no memory is allocated
```

To avoid checking, internally, whether a given string representation is a **null pointer value** on each access, we instead create a common **static** *empty* string, `s_empty`, nested within our `String` class, and install its address during default construction or when we would otherwise want to represent an empty string. This address serves as a sentinel whose requisite runtime checking is be properly relegated to more costly and/or presumably less frequent operations, such as copy construction, assignment and destruction. Thus, a second object invariant is that a string value whose representation is dynamically allocated is never empty.

Finally, to provide better factoring, the definition of our **value-semantic** `String` class declares a private **static** member function, `dupStr`, that dynamically allocates and populates a new block of memory exactly sized to hold a supplied, nonempty **null-terminated-string** value:

```
// my_string.h:
// ...

class String  // greatly simplified null-terminated-string manager
{
    const char* d_str_p;  // immutable value, often allocated dynamically

    static const char s_empty[1];  // empty, used as sentinel indicating null

    static const char* dupStr(const char* str);  // allocate/return copy of str

public:
    // C++03
    String();                               // default constructor
    String(const char* value);              // value constructor
    String(const String& original);         // copy constructor
    ~String();                              // destructor
    String& operator=(const String& rhs);  // copy-assignment operator
    const char* str() const;

    /* C++11 (to be added later)
    String(String&& expiring) noexcept;    // move constructor
```

```
    String& operator=(String&& expiring);  // move-assignment operator
    */
};
```

Perhaps the best way to unpack the C++03 class definition of `String` in the code snippet above is to define its members in order of declaration in a corresponding `.cpp` file, realizing, of course, that all but the definition of the **`static`** data member, `s_empty`, would most likely be moved to the `.h` file as **`inline`** functions:

```
// my_string.cpp:
#include <my_string.h>  // Component header is routinely included first.
#include <cstddef>      // std::size_t
#include <cstring>      // std::memcpy, std::strlen
#include <cassert>      // standard C assert macro

/* C++11 (to be added later)
#include <utility>   // std::move
*/

const char String::s_empty[1] = {'\0'};  // const needs explicit initialization.
```

The static helper, `dupStr`, provides a factored implementation to create a copy of a *nonempty* null-terminated string. Note that, while this function would work on an empty string, our object invariant is that all empty strings are represented by the *sentinel*, `s_empty`, thus avoiding any gratuitous memory allocation:

```
const char* String::dupStr(const char *str)
{
    assert(str && *str);                    // strs are expected to be nonempty.
    std::size_t capacity = strlen(str) + 1; // Calculate capacity.
    char* tmp = new char[capacity];         // Allocate memory.
    memcpy(tmp, str, capacity);             // Copy all chars thru final '\0'.
    return tmp;                             // Return duplicate of str.
}
```

Our deliberate choice to avoid coupling the clean implementation in the example above with the `s_empty` sentinel — and thereby further reducing the amount of repetitive source code — allows us to express explicitly and more precisely detailed design and coding decisions that are generally applicable. For example, calling `dupStr` on either a null pointer value, e.g., `0`, or an empty string, e.g., `""`, is **out of contract** and, in a debug build, would be flagged as an error. Given a defect-free implementation of `String`, however, this assertion can never be triggered as it is a *manifestly* **defensive check**.

With these *independent* primitive utilities in hand, we are now ready to implement the client-facing interface. The simplest and most straightforward member function is the default constructor, which simply installs the sentinel, `s_empty`, as its string value:

```
String::String() : d_str_p(s_empty) { }  // set d_str_p to null value
```

We say that the object is effectively set to a "null" value internally because `s_empty`, although not literally a null pointer value, consumes no dynamic-memory resources; externally, however, it represents efficiently an empty string value (`""`).

Next let's consider the value constructor, which allocates memory only when the supplied string buffer is neither a null pointer value (`0`) nor empty (`""`):

```
String::String(const char* value)
    // Value constructor allocates only if nonempty and then exactly as needed.
{
    if (!value || !*value)        // if value is null or empty
    {
        d_str_p = s_empty;        // no dynamic memory allocation
    }
    else                          // not empty
    {
        d_str_p = dupStr(value);  // allocated copy
    }
}
```

If the supplied `value` is either a null pointer value or an empty string, the sentinel is stored; otherwise, our trusty `dupStr` function will dynamically allocate and populate an appropriately sized **`char`** array to be managed by the `String` object going forward.

Next, let's consider what it means to copy construct a `String` object from another. In this case, we know that the internal representation of the other string cannot be a null address or even a dynamically allocated empty string; `d_str_p` holds either the sentinel, `s_empty`, or else a nonempty dynamically allocated string:

```
String::String(const String& original)
    // Copy constructor allocates exactly what's needed.
{
    if (s_empty == original.d_str_p)         // if original is null
    {
        d_str_p = s_empty;                   // no dynamic allocation
    }
    else                                     // not empty
    {
        d_str_p = dupStr(original.d_str_p);  // allocated copy
    }
}
```

If the original object is *null*, then so too will be a copy; otherwise, a new resource will be allocated via `dupStr` to be managed by this object.

The destructor needs to deallocate when the object being managed isn't *null*:

```
String::~String()
    // Destructor deallocates only if needed.
{
    if (s_empty != d_str_p)  // if not null
    {
        delete[] d_str_p;    // deallocate dynamic memory
    }
}
```

Thus, an empty `String` object, or an array thereof, requires no allocation on construction and no corresponding deallocation on destruction.

By far the most complex of the special member functions to implement explicitly is the copy-assignment operator. First, we must guard against classic aliasing in which the source and the destination of the assignment refer to the same object, or, if it is a possibility with the type in question, overlapping parts of the same object; in such cases, no state change is indicated. Second, we must secure any needed resources; that way, if an exception is thrown (e.g., due to being out of available memory), the destination will be unaffected. Third, we can now proceed to deallocate any resource currently managed by the object. Fourth, we install the new resource into the destination object. Fifth, we must remember to return a reference to the destination object:

```
String& String::operator=(const String& rhs)
     // Copy-assignment operator deallocates and allocates if and as needed.
{
    if (&rhs != this)                   // (1) Avoid assignment to self.
    {
        const char* tmp;                // (2) Hold preallocated resource.

        if (s_empty == rhs.d_str_p)     // If the rhs string is null,
        {
            tmp = s_empty;              // make this string null too.
        }
        else                            // rhs string is not empty.
        {
            tmp = dupStr(rhs.d_str_p);  // allocated copy
        }

        if (s_empty != d_str_p)         // (3) If this object isn't null,
        {
            delete[] d_str_p;           // deallocate storage.
        }

        d_str_p = tmp;                  // (4) Assign the resource.
    }

    return *this;                       // (5) lvalue reference
}
```

The comparatively greater complexity of copy assignment derives, in part, from there being two objects involved, either of which may be managing dynamically allocated memory resources.[13] Again, to ensure exception safety (in this case, the **strong guarantee**), we must *always* remember to allocate any new resource before modifying the state of the destination object.

One last function needed to complete the example is the `str` accessor, which affords direct, efficient access to a *null*-terminated string representing the value managed by a `String` object:

```
const char* String::str() const { return d_str_p; }
```

[13]Implementing assignment operations becomes even more challenging in the presence of locally supplied memory allocators; see **?.This is the Meeting C++17 talk (two parts)**.

```
    // Value accessor returns null-terminated string with maximal efficiency.
```

Note that much of the prep work that was done already was in anticipation of making the implementation of this accessor function as streamlined and runtime efficient as possible, e.g., no conditional branching on each access.

The C++03 example above was written in a style that is consistent with sound C++03 design. We can, for example, create a large array of empty strings without having to separately allocate dynamic memory for each element:

```
String a[10000] = {};  // allocates no dynamic memory

static_assert(sizeof a == sizeof(char*) * 10000);
```

For a fixed-size array, we're in good shape; we'll allocate memory if and when we want to install a nonempty string. For a dynamically growing container, however, substantial inefficiencies can occur. For example, suppose we create an empty `std::vector<String>` object, `vs`, and then append, using `push_back`, 5 nonempty `String` objects using values `"a"`, `"bb"`, `"ccc"`, `"dddd"`, and `"eeeee"`. Assuming a geometric growth strategy for the capacity of the vector starting with the sequence 0, 1, 2, 4, 8 .., we might expect a total of 17 dynamic allocations due to constructing `String` objects on top of the 4 dynamic allocations needed to resize the `vector` from 0 to 1 to 2 to 4 to 8 elements, along with all the deallocations associated with those allocations; note that this illustration assumes 8-byte **`char`** pointers, which dictates the footprint size of our `String` object:

```
#include <my_string.h>
#include <vector>  // std::vector

void test1() // using C++03's vector::push_back with C++03 String
{
    std::vector<String> vs;  // no dynamic allocation or deallocation
    vs.push_back("a");       // 2 8 2 -2
    vs.push_back("bb");      // 3 16 3 2 -2 -8 -3
    vs.push_back("ccc");     // 4 32 4 2 3 -2 -3 -16 -4
    vs.push_back("dddd")     // 5 5 -5
    vs.push_back("eeeee");   // 6 64 6 2 3 4 5 -2 -3 -4 -5 -32 -6
}                            // -2 -3 -4 -5 -6 -64
```

In the `test1` function above, an empty `std::vector`, `vs`, is created without any memory being allocated dynamically. The first time we "push back" a value, `"a"`, a temporary `String` object is constructed, requiring an allocation of 2 bytes. The vector `vs` is then resized from a capacity of 0 to 1, allocating 8 bytes on a 64-bit platform, and then the temporary `String` is copy constructed into this dynamic array of 1 element, requiring another 2 bytes to be allocated. When the original temporary is destroyed, those two bytes are reclaimed (-2 bytes).

Adding a second string, `"bb"`, to `vs` requires a reallocation by `std::vector` to a capacity of two `String` objects (16 bytes). So 3 bytes for a temporary to hold `"bb"`, 16 bytes for the new capacity, 3 bytes to copy construct the temporary into the array, and 2 bytes to copy over the `String` representing `"a"`. After that, the original string holding `"a"` in the array of smaller capacity is destroyed (-2 bytes), the old capacity is deallocated (-8 bytes), and the original temporary used to "push back" the `"bb"` value is destroyed (-3 bytes).

Adding `"ccc"` is again similar. However, adding `"dddd"` is different in that, for the first time, there is sufficient capacity in the array *not* to have to resize. The temporary `String` needed to hold `"dddd"` is created (5 bytes), it is copied into the vector (5 bytes), and the temporary is then destroyed (-5 bytes). Adding `"eeeee"` is analogous to adding `"ccc"`.

Finally, when `vs` goes out of scope, there are 5 calls to the `String` destructor, each requiring a deallocation, plus a call to the destructor of `std::vector`, deallocating the 64-byte-capacity buffer stored within.

The excessive cost due to copying results from two distinct causes. 1. Having to construct a temporary string object before copying it into the vector 2. Having to copy each string from the old-capacity buffer to the new one each time the vector is resized

As of C++11, the Standard Library version of `std::vector` provides a more efficient member function, `emplace_back`, that makes use of **forwarding references** (see Section 2.1."**??**" on page **??**) to construct the string only once *in place*, thus reducing the number of allocations resulting from constructing `String` objects from 17 to 12:

```
void test2() // using C++11's vector::emplace_back with C++03 String
{
    std::vector<String> vs;    // no dynamic allocation or deallocation
    vs.emplace_back("a");      // 8 2
    vs.emplace_back("bb");     // 16 3 2 -2 -8
    vs.emplace_back("ccc");    // 32 4 2 3 -2 -3 -16
    vs.emplace_back("dddd")    // 5
    vs.emplace_back("eeeee");  // 64 6 2 3 4 5 -2 -3 -4 -5 -32
}                              // -2 -3 -4 -5 -6 -64
```

But the real win comes from adding the two missing move operations to the C++03 version of class `String` without changing a single character in the existing class definition. Note the *essential* use of **`noexcept`** following the parameter list in the move constructor, which enables `std::vector` to choose to *move* rather than *copy*; see Section 3.1."**??**" on page **??**:

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

String::String(String&& expiring) noexcept
    // Move constructor never allocates or deallocates.
: d_str_p(expiring.d_str_p)      // Assign address of expiring's resource.
{
    expiring.d_str_p = s_empty;  // expiring is now null but valid/empty.
}
```

The *move* constructor in the example above never allocates or deallocates memory; instead, it simply propagates whatever resource was employed by the *expiring* source to the destination. Then, to establish unique ownership by the destination, the address of the sentinel value, `s_empty`, is used to overwrite the previous address in the expiring source.

The move-assignment operator, not necessary for this demonstration, is again somewhat more involved in that it must (1) guard against move assignment to self, (2) ensure that the resource is deallocated if the destination is managing a nonempty string, (3) assign the `expiring` object's resource to the destination object, (4) unconditionally overwrite the previous resource with the empty-string sentinel, and (5) always return an *lvalue*, not an *rvalue*, reference to the destination object:

```
String& String::operator=(String&& expiring)
    // Move-assignment operator never allocates; deallocates if necessary.
{
    if (&expiring != this)           // (1) Avoid assignment to self.
    {
        if (s_empty != d_str_p)      // (2) If this object isn't null,
        {
            delete[] d_str_p;        // deallocate dynamic storage.
        }

        d_str_p = expiring.d_str_p;  // (3) Assign address of expiring's resource.

        expiring.d_str_p = s_empty;  // (4) expiring is now null but valid/empty
    }

    return *this;                    // (5) lvalue reference
}
```

Note that we have chosen *not* to provide the **noexcept** specifier here because, for the *move*-assignment operator, the **noexcept** specifier is typically not needed and not always appropriate; see Section 3.1."**??**" on page **??**.

Now that the `String` class has been properly augmented with the two missing C++11 move operations (**move construction**, in particular), the number of allocations resulting from constructing `String` objects under the same `test2` goes from 12 to 5, where the 5 allocations result from constructing each unique object from its literal string value exactly once. Importantly, allocation and deallocation resulting from copying over elements from the old to the new storage in the `std::vector<String>` is eliminated entirely:

```
void test2() // using C++11's vector::emplace_back with C++11 String
{
    std::vector<String> vs;    // no dynamic allocation or deallocation
    vs.emplace_back("a");      // 8 2
    vs.emplace_back("bb");     // 16 3 -8
    vs.emplace_back("ccc");    // 32 4 -16
    vs.emplace_back("dddd")    // 5
    vs.emplace_back("eeeee");  // 64 6 -32
}                              // -2 -3 -4 -5 -6 -64
```

The use of the **noexcept** specifier in the implementation of the move constructor is essential because of the **strong exception-safety guarantee** promised by the Standard when inserting or emplacing an element at the end of a `std::vector`; see Section 2.1."**??**" on page **??**. It will turn out that, once our `String` class is outfitted with a *move* constructor, the *rvalue* reference of `vector::push_back` will avoid the dynamic allocation and deallocation of a copy, substituting just a single additional *move* operation; `vector::emplace_back` avoids even that tiny overhead.

Finally, the design of a class that supports *move* operations follows from the general idea that construction of an empty container should be inexpensive (i.e., not require memory allocation), as was done here. One could imagine making other trade-offs that would simplify the implementation at the expense of necessitating that every *valid* object maintains

allocated resources, which would undermine a primary advantage of *move* operations, i.e., faster copy operations. For move operations to be both efficient and generally applicable, it is essential that the object has at least one valid state that does not require it to be managing external resources; see *Potential Pitfalls — Requiring owned resources to be valid* on page 77.

eating-a-high-level-vst

**Creating a high-level VST** Implementing move operations for higher-level types, such as **aggregates**, that merely comprise other types that support move operations, is fairly straightforward when compared with lower-level types that manage resources explicitly. Because the resources are managed independently by the move-operation-enabled base class and/or member subobjects themselves, all that's required is to ensure that those move operations are employed.

1. No special member functions are user provided. As an example, suppose we want to create a class that comprises two independent strings, `firstName` and `lastName`, describing a person. For this application, there is no special constraint between the two string data members, and we will want to provide *write* as well as *read* access to each member unless the object of that class is itself **`const`** or accessed via a **`const`** pointer or reference. We will therefore elect to implement our class as a **`struct`** having two public data members of a **regular** value-semantic type, such as `std::string`, that manages its own resources. For concreteness of exposition, however, we will use the `String` class developed in *Creating a low-level value-semantic type (VST)* on page 29:

```
struct Person  // C++03 and C++11 aggregate type
{
    String firstName;
    String lastName;
};
```

The `Person` class above does not declare any explicit constructors of its own, yet it can be **default constructed** and **braced initialized** (see Section 2.1."??" on page ??):

```
Person p0;                                  // p0 holds { "", "" }
Person p1 = {};                             // p1 holds { "", "" }
Person p2 = { "Slava" };                    // p2 holds { "Slava", "" }
Person p3 = { "Alisdair", "Meredith" };  // p3 holds { Alisdair, "Meredith"
}
```

Note that, in the code sample above, `p0` is **default initialized** by an implicitly generated default constructor, whereas `p1`, `p2`, and `p3` follow the rules of **aggregate initialization**; see Section 2.1."??" on page ??.

Once constructed, the individual members of a `Person` object can be manipulated directly:

```
void test0()
{
    p0.firstName = "Vittorio";  // OK, p0 holds { "Vittorio", "" }.
    p0.lastName = "Romeo";      // OK, p0 holds { "Vittorio", "Romeo" }.
}
```

Moreover, a `Person` object, as a whole, can be both copy constructed and **copy assigned** via its corresponding, implicitly generated special member functions:

```
void test1()
{
    Person x(p3);  // x holds { "Alisdair", "Meredith" }; p3 is unchanged.
    x = p0;        // x now holds { "Vittorio", "Romeo" }; p0 is unchanged.
}                  // x destroys both of its member objects as it leaves scope.
```

What's more, when a `Person` object, e.g., `x` in the code snippet above, leaves scope, its respective members are destroyed by its implicitly generated destructor. That is, a C++03 aggregate that comprises objects that individually support all of the needed special member functions will tacitly make the corresponding operations available to the overall aggregate type.

To update the `Person` object to support move operations, there is literally nothing to do! Provided that each member (and base) subobject supports the desired move operations for itself (as we know to be the case for `String` objects), the compiler generates them implicitly for the `Person` **`struct`** as well:

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

void test2()
{
    Person x(p3);            // x holds { "Alisdair", "Meredith" }.
                             // p3 still holds { "Alisdair", "Meredith" }.

    Person y(std::move(x));  // y holds { "Alisdair", "Meredith" }.
                             // x might now hold { "", "" }.

    x = std::move(y);        // x now holds { "Alisdair", "Meredith" }.
                             // y might now  hold { "", "" }.
}
```

The example above demonstrates that moving from an object that has been explicitly designated as expiring enables the `Person` class to, in turn, *allow* its member types to steal resources from the corresponding individual members of the other object during both move construction and move assignment. The result is just a more efficient copy provided that we no longer rely on knowing the value of the moved-from object.

With respect to the specific state of moved-from `String` data members of a `Person`, the example code above — though illustrative of what we know to be the currently implemented behavior of our own `String` — would not necessarily be true of another string type, e.g., the vendor-supplied implementation of `std::string`, or even a future version of `String`; see *Potential Pitfalls — Inconsistent expectations on moved-from objects* on page 70.

Although pure aggregate types often suffice, we might sometimes find that we need to augment such an aggregate type with one or more user-provided special member functions or other constructors; doing so, however, may affect the implicit generation of other such functions and/or strip the type of its aggregate type status.

In what follows, we will proceed to add a **value constructor** (see item 1) followed by each of the six special member functions in turn (see items 2–7). [**AUs: You started your list with item 0; our list style starts with item 1. Please double-check the prev sentence and change as needed: item 1 might need to be item 2, and items 2–7 might need to be items 3–8.**] After adding each function, we will observe the consequences of having added it and then default back whatever was removed. We will repeat this process until all of the missing functionality is restored. The process provides insight into the deeper meaning embedded in the special-member-function table provided in the appendix within Section 1.1."**??**" on page **??**.

2. User-provided value constructor. As it stands now, there is no value constructor that takes the two member values and constructs a `Person` object:

```
Person nonGrata("John", "Lakos");  // Error, no matching constructor
```

Suppose we want to make it possible for a person class, e.g., `Person1`, to be constructed in this way. We will, at a minimum, need to provide a value constructor for that purpose:

```
struct Person1  // We want to add a value constructor.
{
    String firstName;
    String lastName;

    Person1(const char* first, const char* last)  // value constructor
    : firstName(first), lastName(last) { }
};
```

By adding this value constructor in the example above, we necessarily suppress the implicit declaration of the **default constructor**. Moreover, a class having a user-provided constructor of any kind is automatically no longer considered an aggregate; see Section 2.1."**??**" on page **??**:

```
Person1 w1("John", "Lakos");          // OK, invokes value constructor
Person1 x1;                           // Error, no default constructor
Person1 y1 = { "Vittorio" };          // Error, no longer an aggregate
Person1 z1 = { "Vittorio", "Romeo" }; // OK, calls value ctor as of C++11
```

As the example above illustrates, we can now **direct initialize** `w1` via the new, user-provided value constructor, but there is now no implicitly declared default constructor for `x1`. Since `Person1` is no longer considered an aggregate, we cannot use aggregate initialization to initialize (in this case, just some of) the members of `y1`. C++11, however, extends what can be done via braced-list initialization and thus allows this value constructor to be invoked via the rules of **braced-list initialization**; see Section 2.1."**??**" on page **??** and also Section 2.1."**??**" on page **??**.

Still, we would like a way to somehow support the functionality provided by the implicit implementation of all *six* of the C++11 special member functions. Fortunately, C++11 offers a companion feature that allows us to declare and default the compiler-generated

implementation of each special member function explicitly (see Section 1.1."**??**" on page **??**):

```
struct Person1a  // We want to add a value constructor (Revision a).
{
    String firstName;
    String lastName;

    Person1a(const char* first, const char* last)  // value constructor
    : firstName(first), lastName(last) { }

    // New here in Person1a.
    Person1a() = default;                          // default constructor
};
```

Notice that `Person1a` differs from `Person1` in the earlier example only in the addition of a *defaulted* default constructor on the final line of its class definition, thus restoring default construction but not aggregate initialization:

```
Person1a x1a("John", "Lakos");           // OK, invokes value constructor
Person1a y1a;                            // OK, invokes default constructor
Person1a z1a = { "Vittorio" };           // Error, still not an aggregate
Person1a w1a = { "Vittorio", "Romeo" };  // OK, calls value ctor as of C++11
```

Adding *any* nonspecial constructor would have the same effect on the implicit generation of the six special member functions, which in this case was to suppress implicit generation of only the default constructor. This invites the question, What affect does explicitly declaring the default constructor have on other special functions, e.g., move construction? The answer in this specific case is none, but that's the exception; the ultimate answer for each of the other five special member functions is elucidated in items 2–7.[**AUs: You started your list with item 0; our list style starts with item 1. Please double-check the prev sentence and change as needed: items 2–7 might need to be items 3–8.**]

3. User-provided default constructor. We may, at times, need to augment an aggregate with one or more user-provided special member functions, e.g., for debugging or logging purposes, that would *not* alter how we would have the compiler generate the remaining ones.

   Suppose that, for whatever reason, we'd like to add some metrics-gathering code to the default constructor of our original, aggregate `Person` class; we'll call this modified class `Person2`:

```
struct Person2  // We want to employ a user-provided default constructor.
{
    String firstName;
    String lastName;

    Person2() { /* user-provided */ }  // default constructor
};
```

No other **special member function** is affected but, because we have provided a nondefault implementation of a constructor (namely, the default constructor), the class is no longer considered an aggregate:

```
Person2 w2("abc", "def");       // Error, no value constructor is provided.
Person2 x2;                     // OK, invokes user-provided default ctor
Person2 y2 = { "abc" };         // Error, not an aggregate and no 1-arg ctor
Person2 z2 = { "abc", "def" };  // Error, not an aggregate and no value ctor
```

4. User-provided destructor. More typically, augmenting an aggregate with a user-provided special member function will suppress the implicit generation of others. In such cases, we can apply the same `=default` approach used in item 0 to automatically generate the default implementations of those special member functions that were suppressed.

   A user-provided destructor, unlike a default constructor, has far-reaching implications on the implicit generation of both *move* operations that, absent thorough unit testing, might go unnoticed and remain latent defects even after being released to production. Suppose we'd like to add some benign code to the destructor of the aggregate class `Person` and call the new class `Person3`:

```
struct Person3  // We want to employ a user-provided destructor.
{
    String firstName;
    String lastName;

    ~Person3() { /* user-provided */ }  // destructor
};
```

   Declaring the destructor suppresses the declaration and implicit generation of both the move constructor and move-assignment operator but leaves the object of aggregate type. Absent proper unit testing, the object might appear to work as before, though it silently *copies* rather than *moves* its subobjects:

```
Person3 x3 = { "abc", "def" };  // OK, still an aggregate
Person3 y3;                     // OK, default constructor is still available.
Person3 z3(std::move(x3));      // Bug, invokes implicit copy constructor
y3 = std::move(z3);             // Bug, invokes implicit copy assignment
```

   Although we can still aggregate initialize (`x3`) and default construct (`y3`) a `Person3` object, no *move* operations are declared, so when we try to move construct (`z3`) or **move assign** (`y3`) a `Person3`, the corresponding implicitly generated *copy* operations are invoked automatically instead. The result is that each of the three objects — `x3`, `y3`, and `z3` — now manages its own dynamically allocated memory to represent needlessly the same overall person value, namely `{} "abc", "def" }`:

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

void test3()
```

```
{
    assert(strcmp(x3.firstName.str(), ""));     // Error, still holds "abc"
    assert(strcmp(y3.firstName.str(), "abc")); // OK, holds "abc"
    assert(strcmp(z3.firstName.str(), ""));     // Error, still holds "abc"
}
```

Suppose we now decide to default both of these suppressed move operations explicitly in `Person3a`:

```
struct Person3a  // Adding user-provided destructor (Revision a)
{
    String firstName;
    String lastName;

    ~Person3a() { /* user-provided */ }         // destructor

    Person3a(Person3a&&) = default;             // move constructor
    Person3a& operator=(Person3a&&) = default;  // move assignment
};
```

By explicitly declaring the move-assignment operator to be **defaulted**, we restore that capability but implicitly **`delete`** (see Section 1.1."**??**" on page **??**) both the copy constructor and copy-assignment operator. By explicitly defaulting the move constructor, however, not only do we implicitly suppress both copy operations, but we also suppress default construction. However, because no constructor has been user-provided (merely defaulting and/or **deleting** them would not be considered user-provided), the overall object remains of aggregate type:

```
Person3a x3a = {"abc", "def" };  // OK, still an aggregate
Person3a y3a;                    // Error, default constructor is not declared.
Person3a z3a(x3a);               // Error, copy constructor is deleted.
x3a = z3a;                       // Error, copy assignment is deleted.
```

Defaulting both copy operations and the **default destructor**, e.g., `Person3b`, restores the type to its previous functionality as a proper aggregate:

```
struct Person3b  // Adding a user-provided destructor (Revision b)
{
    String firstName;
    String lastName;

    ~Person3b() { /* user-provided */ }             // destructor

    // Already added to Person3a.
    Person3b(Person3b&&) = default;                 // move constructor
    Person3b& operator=(Person3b&&) = default;      // move assignment

    // New here in Person3b.
    Person3b() = default;                           // default constructor
    Person3b(const Person3b&) = default;            // copy constructor
```

```
    Person3b& operator=(const Person3b&) = default;  // copy assignment
};
```

To recap, adding the user-provided destructor in `Person3` suppressed both move operations. Defaulting these move operations in `Person3a`, in turn, suppressed both copy operations and, because of the move constructor specifically, the default constructor as well. Defaulting those three operations in `Person3b` restores everything we had in the original `Person` aggregate yet allows us to add benign metrics to a user-provided destructor as we see fit.

5. User-provided copy constructor. Once we understand the consequences of defining our own destructor (e.g., `Person 3` in the most recent example), the story behind adding a copy constructor to our original aggregate `Person` class will seem remarkably similar. Rather than repeat that laborious discovery process in all its detail, starting with `Person4` (the original aggregate + the copy constructor, not shown), we'll instead provide only the end result, namely `Person4c` with all special member functionality restored:

```
struct Person4c  // adding a user-provided copy constructor
{
    String firstName;
    String lastName;

    Person4c(const Person4c& original)              // copy constructor
    : firstName(original.firstName)
    , lastName(original.lastName) { /* user-provided */ }

    // Already added to Person4a (not shown).
    Person4c() = default;                           // default constructor

    // Already added to Person4b (not shown).
    Person4c(Person4c&&) = default;                 // move constructor
    Person4c& operator=(Person4c&&) = default;      // move assignment

    // New here in Person4c.
    Person4c& operator=(const Person4c&) = default;  // copy assignment
};
```

Though we don't show the revisions to `Person`, `Person 4a`, and `Personb` in the example above, we'll walk through those changes. The class necessarily forfeits its aggregate status in the first revision to `Person`, namely `Person4a`, by user-providing a copy constructor or a constructor of any kind. What's more, explicitly declaring a copy constructor (in any way) suppresses implicit generation of the default constructor. Hence, any unit test expecting to construct the newly augmented person object using either a default constructor or aggregate initialization will not compile.

By defaulting the default constructor in `Person4a`, we regain the ability to compile our unit test driver and can now observe that, by having declared the copy constructor

explicitly, we had suppressed both move construction and **move assignment**, thus introducing a potentially latent performance defect. Defaulting the two move operations in `Person4b` restores their respective move capabilities. Defaulting move assignment in particular, however, now causes **copy assignment** to be implicitly deleted; see Section 1.1."**??**" on page **??**.

Finally, by defaulting the copy-assignment operator in `Person4c` (shown in the example above), we regain all of the regular functionality of the original aggregate class but, because of the user-provided copy constructor, `Person4c` no longer amenable to aggregate initialization. Note that the destructor is *never* suppressed by the explicit declaration of any other function.

6. User-provided copy-assignment operator. Once we understand the consequences of user-providing a copy constructor (e.g., `Person4c` in the example in item 4), there are no surprises here. Again, we'll provide, for reference, only the final, transitive result, `Person5b`:

```
struct Person5b  // adding a user-provided copy-assignment operator
{
    String firstName;
    String lastName;

    Person5b& operator=(const Person5b& rhs)    // copy assignment
    {
        firstName = rhs.firstName;
        lastName  = rhs.lastName;
        return *this;
    }

    // Already added to Person5a (not shown).
    Person5b(Person5b&&) = default;             // move constructor
    Person5b& operator=(Person5b&&) = default;  // move assignment

    // New here in Person5b.
    Person5b() = default;                       // default constructor
    Person5b(const Person5b&) = default;        // copy constructor
};
```

Again, we've omitted `Person5` and `Person5a` from the example above, but we'll walk through those revisions. Providing a user-defined copy-assignment operator in `Person5`, unlike providing a copy constructor, leaves the class of aggregate type but similarly suppresses the declaration of both the move constructor and move-assignment operator. Restoring move assignment in `Person5a` has no further suppressive effects, but restoring move construction in turn suppresses both default construction and copy construction. Class `Person5b`, shown in the code snippet above, provides the same functionality as the original aggregate `Person` class, i.e., including aggregate initialization, along with the ability to add a benign implementation, affecting no other special-member implementations, to the user-provided copy-assignment operator.

7. User-provided move constructor. Instrumenting a move constructor during development, if just to ensure that it is being called when expected, isn't a bad idea. Again, we will provide the final result and an appropriate analysis of how we got here:

```
struct Person6b  // adding a user-provided move constructor
{
    String firstName;
    String lastName;

    Person6b(Person6b&& expiring)                    // move constructor
    : firstName(std::move(expiring.firstName))
    , lastName (std::move(expiring.lastName)) { /* user-provided */ }

    // Already added to Person6a (not shown).
    Person6b() = default;                            // default constructor
    Person6b(const Person6b&) = default;             // copy constructor
    Person6b& operator=(const Person6b&) = default;  // copy assignment

    // New here in Person6b.
    Person6b& operator=(Person6b&&) = default;       // move assignment
};
```

   First note the use of `std::move` in the user-provided implementation of the move constructor in the example above. Recall that a parameter of type *rvalue* reference (`&&`) is an *lvalue*, so `std::move` is *required* to enable a move from such a parameter. Not employing `std::move` would mean that these data members would be individually copied rather than moved. Absent a thorough unit test, such inadvertent, pessimizing omissions might well find their way into widespread use.

   In the original enhanced version (`Person6a`, which is not shown in the example), the user-provided move constructor immediately renders the class to be of nonaggregate type. Moreover, both the copy constructor and the copy-assignment operator are deleted, and the default constructor and the move-assignment operator are not implicitly generated. Since there is no way to create an object and then learn that the move-assignment operator is missing (short of knowing, as we do here), the first step is to get the unit test driver to compile, which is accomplished by defaulting the default constructor, copy constructor, and copy-assignment operator in `Person6a`.

   After that, our thorough unit tests can observe that what should be move assignment is falling back on copy assignment, which needlessly allocates new resources rather than transferring them when the source is expiring. By now also defaulting the move-assignment operator, we arrive at a class that again has all the regular functionality of the original `Person` class, namely `Person6b` (shown in the code snippet above), but absent the ability to be aggregate-initialized.

8. User-provided move-assignment operator. Instrumenting a move-assignment operator during development, just like a move constructor, can be useful:

```
struct Person7b  // adding a user-provided move-assignment operator
{
```

```
    String firstName;
    String lastName;

    Person7b& operator=(Person7b&& expiring)         // move assignment
    {
        firstName = std::move(expiring.firstName);
        lastName  = std::move(expiring.lastName);
        return *this;
    }

    // Previously added to Person7a (not shown).
    Person7b(const Person7b&) = default;             // copy constructor
    Person7b& operator=(const Person7b&) = default;  // copy assignment
    Person7b(Person7b&&) = default;                  // move constructor

    // New here in Person7b.
    Person7b() = default;                            // default constructor
};
```

Again, we've omitted `Person7` and `Person7a`, but we'll discuss those revisions. Importantly, let's again note the use of `std::move` in the implementation of the user-provided move-assignment operator; without it, the members would instead be *copied* undesirably rather than *moved*. In the original version (`Person7`), the user-provided move-assignment operator results in the deletion of both copy operations as well as the suppression of the implicit declaration of the move constructor. However, unlike the user-provided move constructor, the user-provided move-assignment operator doesn't affect the aggregate nature of the overall type nor does it immediately suppress the default constructor.

Since there is neither a copy constructor nor a move constructor available, both omissions would likely show up as compile-time errors in a thorough unit-test suite. By subsequently defaulting all three missing special member functions (`Person7a`), we would discover that the defaulting of the copy and move constructors has, in turn, suppressed the implicit declaration of the default constructor.

Finally, by defaulting the default constructor (`Person7b`), we regain all of the original functionality of the aggregate `Person` class including the ability to aggregate initialize it.

In summary: We can create a higher-level, value-semantic type (VST) quickly and reliably by combining lower-level ones. These higher-level VSTs can be aggregate initialized, default initialized, copied, and moved as a unit, provided the types of the respective, lower-level, base and member subobjects support the needed operations.

Sometimes a user will need to provide a custom implementation of one or more of these special member functions, which may affect implicit generation of other member functions and/or the ability to aggregate initialize the object; see Section 2.1."**??**" on page **??**.

One might reasonably decide to just explicitly default *all* of the remaining special member functions, and that works well in most practical cases. Note, however, that, unlike suppression of the copy operations, when the default constructor or either move operation

is suppressed by an explicit declaration of some other function, it is left **undeclared** rather than being *declared* but deleted, which in turn can have subtle implications.

Explicitly or implicitly defaulting (see Section 1.1."??" on page **??**) or deleting (see Section 1.1."??" on page **??**) a function ensures that it *is* **declared** and, hence, may participate in overload resolution as well as affect the outcome of certain compile-time type traits, such as `std::is_literal_type`, which can be found in the standard header `<type_traits>` (see Section 2.1."??" on page **??**).

Explicitly **defining** a special member function to have the same implementation as would have been generated had that function been declared implicitly will produce the same behavior locally but may affect the outcome of certain type-wide traits such as `std::is_aggregate`, which also can be found in `<type_traits>`.[14]

semantic-container-type

**Creating a generic value-semantic container type** A primary motivation for the introduction of *rvalue* references into C++11 was the desire to provide ubiquitous, uniform support for a pair of special member functions, parallel to the copy constructor and the copy-assignment operator, that distinguish when it is permissible to *steal* resources from the source object.

In particular, for a standard container, such as `std::vector`, to provide the strong exception-safety guarantee for public member functions, e.g., `push_back`, was deemed unacceptably inefficient in C++03 because resizing capacity necessitated copying over all of its element subobjects only to then immediately (and wastefully) destroy the original ones. The gratuitous copying made necessary to preserve the strong guarantee could be sidestepped provided the element-type argument supplied to the `vector` template sports a nonthrowing move constructor.

Proper implementation of `std::vector` requires the ability to detect, at compile time, whether the move constructor of the specified element type is permitted to throw; a thorough discussion of how to implement robust `std::vector`-like operations, such as `push_back` and `emplace_back`, along with the strong guarantee is provided in Section 2.1."??" on page **??**. Here, we will introduce a simplified, fixed-size-array type to illustrate the straightforward benefits of *rvalue* references while sidestepping distractions arising from the strong guarantee, resizing capacity, and local memory allocators.[15]

Let's now consider the class definition for a simple generic container type, `FixedArray<T>`, that provides a minimal set of capabilities to manage a dynamically allocated, "fixed-size" array of elements of type `T`. The implementations of all six special member functions are user-provided, along with an "extra" (**`explicit`**) size constructor, used to set the fixed capacity of the array at construction. Note that only the *copy* and *move* assignment operators may change the size and capacity of an existing array object.

Finally, three more member functions are provided to access array elements by index of a non**`const`** and **`const`** array, along with a **`const`** member function to access the array's size. Note that the behavior is undefined unless the user-supplied element index is less than the array size:

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

[14] Rule C.20 of the *C++ Core Guidelines* advises, "If you can avoid defining default operations, do."; see **?**.

[15] A thorough description of the practical use of **C++11 memory allocators** and, especially, **C++17 pmr allocators** is anticipated in **?**.

```
template <typename T>
class FixedArray
{
    T*          d_ary_p;  // dynamically allocated array of fixed size
    std::size_t d_size;   // number of elements in dynamically allocated array

public:
    FixedArray();                                          // default constructor
    explicit FixedArray(std::size_t size);                 // size constructor
    ~FixedArray();                                         // destructor
    FixedArray(const FixedArray& original);                // copy constructor
    FixedArray& operator=(const FixedArray& rhs);          // copy-assignment
    FixedArray(FixedArray&& expiring) noexcept;            // move constructor
    FixedArray& operator=(FixedArray&& expiring) noexcept;  // move-assignment

    T& operator[](std::size_t index);                  // modifiable element access
    const T& operator[](std::size_t index) const;  // const element access
    std::size_t size() const;                          // number of array elements
};
```

The class definition in the example above — apart from the two C++11 move operations — would be the same in C++03. Moreover, we can instantiate and use a `FixedArray` on any VST that supports at least the four classic special member functions:

```
FixedArray<double> ad;           // empty array of double objects
FixedArray<int>    ai(5);        // array of 5 zero-valued int objects

#include <string>                // std::string
FixedArray<std::string> as(10);  // array of 10 empty std::string objects
```

Note that if the element type supplied for the template parameter, `T`, does not support C++11 move operations, then neither will the `FixedArray<T>` container itself.

Let's now consider the implementation, which — apart from the two move operations — is the same as it would have been in C++03. Notably, creating a default-constructed container ideally allocates no resources (if just for performance reasons), and the **size constructor** is deliberately made explicit to avoid inadvertently creating a fixed array via an unexpected implicit conversion from an **`int`** (see Section 1.1."**??**" on page **??**):

```
template <typename T>  // default constructor
FixedArray<T>::FixedArray() : d_ary_p(0), d_size(0) { }

template <typename T>                    // size constructor
FixedArray<T>::FixedArray(std::size_t size) : d_size(size)
{
    d_ary_p = size ? new T[size]() : 0;  // value initialize each element
}
```

When allocating a resource, we always consistently do so using array **`new`**. Notice that we have deliberately **value initialized** — i.e., **`new`** `T[size]()` — the dynamically allocated

array. For an object that has a default constructor, that constructor would be called regardless; for scalar or user-defined aggregate types, failing to value initialize — i.e., `new T[size]` — could leave the elements, or parts thereof, in an uninitialized state.

It is not uncommon to place defensive checks for object invariants at the top of the body of a destructor as this is the one place every object must pass through before it is destroyed:

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

template <typename T>  // destructor
FixedArray<T>::~FixedArray()
{
     assert(!d_ary_p == !d_size);  // Assert object invariant.
     delete[] d_ary_p;             // resource released as an array of T
}
```

Our object invariant for this `FixedArray` type requires that a memory resource is allocated if and only if the array size is nonzero. Notice also that every allocation is presumed to be allocated by array **new** and never directly, e.g., via **operator new**.

When it comes to copy construction, the implementation is mostly straightforward. If the size of the original object is nonzero, we allocate enough memory to hold the requisite number of elements — nothing more — and then proceed to copy assign them one by one; if the original object was empty, then we make this object's resource handle *null* too:

```
template <typename T>  // copy constructor
FixedArray<T>::FixedArray(const FixedArray& original) : d_size(original.d_size)
{
    if (d_size)  // if original array is not empty
    {
        d_ary_p = new T[d_size];         // default initialize each element

        for (std::size_t i = 0; i < d_size; ++i)  // for each array element
        {
            d_ary_p[i] = original.d_ary_p[i];     // copy-assign value
        }
    }
    else  // else original array is empty
    {
        d_ary_p = 0;  // Make this array null too.
    }
}  // Note that we already set d_size in the initialization list up top.
```

Notice that here we have allocated the array using **default initialization** to minimize unnecessary initializations prior to copy assigning them. In a more efficient implementation, we might consider using **operator new** directly and then copy constructing each object in place, but we would then need to destroy each element in a loop as well where we currently use **operator delete[]**; see Section 2.1."**??**" on page **??**. What's more, we could even check whether the element type is **trivially copyable** and, if so, use `memcpy` instead; see Section 2.1."**??**" on page **??**.

As ever, copy assignment inherently has the most complex implementation of all the special member functions. First we must guard against assignment to self. If the two objects

are not the same size, then we will need to make them so. If the target object holds a resource, we'll need to free it before allocating another. Once the two resources are the same size, we'll need to copy the elements over:

```
template <typename T>  // copy-assignment operator
FixedArray<T>& FixedArray<T>::operator=(const FixedArray& rhs)
{
    if (&rhs != this)  // guard against self aliasing
    {
        if (d_size != rhs.d_size)  // If sizes differ, make this same as other.
        {
            if (d_ary_p)  // If this array was not null, clear it.
            {
                delete[] d_ary_p;  // Release resource as an array of T.
                d_ary_p = 0;        // Make null. (Note size isn't yet updated.)
                d_size  = 0;        // Make empty. (Reestablish obj. invariant.)
            }

            assert(!d_ary_p == !d_size);  // Assert object invariant.
            d_ary_p = new T[rhs.d_size];  // Default initialize each element.
            d_size  = rhs.d_size;         // Make this size same as rhs size.
        }

        assert(d_size == rhs.d_size);  // The two sizes are now the same.

        for (std::size_t i = 0; i < d_size; ++i)  // for each element
        {
            d_ary_p[i] = rhs.d_ary_p[i];  // Copy-assign value.
        }
    }
    return *this;  // lvalue reference to self
}
```

Notice that we (1) released any current resources assuming that they were allocated using array **new** and (2) deliberately default initialized each allocated element prior to copy assigning to it. Also notice that we have introduced two defensive checks in the middle of our implementation to serve as "active commentary" to state that, no matter how we got here, what is asserted is either **true** or else the program itself is defective.

Let's now turn to what is new in C++11. We'll first need to **#include** the standard header that defines `std::move`, namely `<utility>`. To implement the move constructor, we'll use the member-initializer list to copy the size and address of the resource to the object. After that, we'll just assign those values to render the source object *null*, i.e., managing no resources, as if it had just been default constructed:

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

template <typename T>      // move constructor
FixedArray<T>::FixedArray(FixedArray&& expiring) noexcept
: d_size(expiring.d_size)
, d_ary_p(expiring.d_ary_p)
```

```
{
    expiring.d_ary_p = 0;  // Relinquish ownership.
    expiring.d_size  = 0;  // Reestablish object invariant.
}
```

In this case, we always know that the value of a moved-from `FixedArray` object will be empty, but see *Potential Pitfalls — Inconsistent expectations on moved-from objects* on page 70.

Next, we consider move assignment. We must, as ever, first check for assignment to self. If not, we unconditionally delete our resource, knowing that it is a no-op if the resource handle is *null*. We then copy over the resource address and size from `expiring` and restore `expiring` to its default-constructed state:

```
template <typename T>       // move-assignment operator
FixedArray<T>& FixedArray<T>::operator=(FixedArray&& expiring)
{
    if (&expiring != this)  // Guard against self-aliasing.
    {
        delete[] d_ary_p;  // Release resource from this obj. as array of T.
        d_ary_p = expiring.d_ary_p;  // Copy address of resource.
        d_size  = expiring.d_size;   // Copy size of resource.
        expiring.d_ary_p = 0;        // Make expiring relinquish ownership.
        expiring.d_size  = 0;        // Re-establish object invariants in expiring.
    }

    return *this;  // Return lvalue (not rvalue) reference to self.
}
```

Note that the move-assignment operator takes an *rvalue* reference but, like its copy-assignment counterpart, returns an *lvalue* reference.

Finally, for completeness, we show the three methods to access the modifiable and **`const`** elements and the size of the array, respectively:

```
template <typename T>                              // modifiable element access
T& FixedArray<T>::operator[](std::size_t index)
{
    assert(index < d_size);  // Assert precondition.
    return d_ary_p[index];   // Return lvalue reference to modifiable element.
}

template <typename T>                              // const element access
const T& FixedArray<T>::operator[](std::size_t index) const
{
    assert(index < d_size);  // Assert precondition.
    return d_ary_p[index];   // Return lvalue reference to const element.
}

template <typename T>                              // number of array elements
std::size_t FixedArray<T>::size() const { return d_size; }
```

Note that the functionality provided by each of the three member functions above is entirely independent of the augmented functionality afforded by the two move-semantic special member functions, the move constructor, and the move-assignment operator.

For an industrial-strength implementation example involving a `push_back` operation overloaded on *rvalue* reference for a `std:vector`, see Section 2.1."??" on page ??.

## Move-only types

move-only-types

We may want some classes to represent **unique ownership** of a resource. For resources without intrinsic copyability, the standard copy operations would not make sense. A move-only type can be used when only one object should have a particular piece of state at any given time. As long as the particular object that owns the state does not need to remain constant (i.e., pointers to the owning object are not maintained in external data structures), adding move operations can bring additional utility to a type without requiring copyability.

For example, `std::thread`, introduced in C++11, represents an underlying operating system thread. There is no way to create a second copy of a running operating system thread, and it would be significantly unintuitive if `std::thread` objects could be easily copied when the underlying resource was not going to be duplicated. On the other hand, since a `std::thread` is primarily a handle to the underlying resource, its own identity does not need to be set in stone, so moving that handle to a different instance of `std::thread` is very natural. This is accomplished by defining a move constructor and then deleting the copy constructor; see Section 1.1."??" on page ?? and Section 3.1."??" on page ??:

```
class thread
{
    // ...
    thread() noexcept;
    thread(thread&& other) noexcept;
    thread(const thread&) = delete;
    // ...
};
```

To transfer the ownership of a thread, we must use `std::move`:

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

void test1()
{
    std::thread t{[] { std::cout << "hello!"; }};
    std::thread tCopy  = t;             // Error, cannot copy
    std::thread tMoved = std::move(t);  // OK
}
```

If we want to transfer the ownership of the thread to another `std::thread` instance, we are forced to explicitly convert `t` to an *rvalue* reference. By using `std::move` we communicate to the compiler and to those reading the code that a new object will now own the underlying operating system thread and that the old object will be put in an empty state.

A related type, `std::unique_lock`, exemplifies the ability to move responsibility between objects, specifically the responsibility to release a lock when destroyed. By combining movability with standard **RAII** there is increased flexibility, enabling the passing and

returning of responsibility without the risk of failing to execute it when all objects are destroyed:

```
#include <mutex>  // std::mutex, std::unique_lock

void test2()
{
    std::mutex m;
    std::unique_lock<std::mutex> ul{m};

    {
        std::unique_lock<std::mutex> ulMoved = std::move(ul);  // OK
    }  // ulMoved destroyed, lock released

    assert(ul.mutex() == nullptr);  // ul is moved-from.
}
```

Finally, the Standard provides `std::unique_ptr` to manage unique ownership of a resource identified by a pointer with a compile-time customizable deleter that will be used to free that resource when a `std::unique_ptr` is destroyed without having had its resource moved away. The most common and default use for this type is to manage heap-allocated memory, where the default deleter will simply invoke **`delete`** on the pointer. C++14 also adds a helper utility, `std::make_unique`, that encapsulates heap allocation with **`new`**:

```
#include <memory>  // std::unique_ptr, std::make_unique

void test3()
{
    std::unique_ptr<int> up1{new int(1)};       // OK, heap alloc #1
    up1 = std::make_unique<int>(2);             // OK, frees #1, new alloc #2

    std::unique_ptr<int> up2 = std::move(up1);  // OK, up2 now owns #2.
    assert( up1 == std::unique_ptr<int>());     // OK, up1 is moved-from.
    assert(*up2 == 2);                          // OK
}  // Destruction of up2 deletes alloc #2; destruction of up1 does nothing.
```

`std:unique_ptr` can be very useful when an object that cannot be copied needs to be referenced by an object whose state might need to move around. While one pre-C++11 solution to this need might be to dynamically allocate the object and track the object's lifetime with a reference-counted smart pointer (such as `std::tr1::shared_ptr` or `boost::shared_ptr`), the addition of that tracking incurs additional overhead that might not be warranted. Using `std::unique_ptr` instead will manage the lifetime of the heap-allocated object correctly, allowing it to remain in a stable location from construction to destruction and letting the client's handle to the object — a `std::unique_ptr` — move to where it needs to be.

enting-a-move-only-type

**Implementing a move-only type** For general purposes, the standard library templates already capture in a reusable fashion the proper implementation of a move-only type, e.g., `std:unique_ptr`. To understand what is involved in such implementations, let's explore how we might implement a subset of the functionality of `std::unique_ptr`. The declaration of our `UniquePtr` with no support for custom deleters simply needs to `=`**`delete`** the

copy constructor and provide the appropriate move constructor and move-assignment operator. Along with the basic accessors to implement a typical **smart pointer**, a complete implementation of a move-only owning pointer is not tremendously involved:

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

template <typename T>
class UniquePtr  // simple move-only owning pointer
{
    T* d_ptr_p;   // owned object

public:
    UniquePtr() : d_ptr_p{nullptr} { }         // construct an empty pointer
    UniquePtr(T* p) noexcept : d_ptr_p(p) { }  // value ctor, take ownership

    UniquePtr(const UniquePtr&) = delete;    // not copyable

    UniquePtr(UniquePtr&& expiring) noexcept // move constructor
    : d_ptr_p(expiring.release()) { }

    ~UniquePtr()                             // destructor
    {
        reset();
    }

    UniquePtr& operator=(const UniquePtr&) = delete;     // no copy assignment

    UniquePtr& operator=(UniquePtr&& expiring) noexcept  // move assignment
    {
        reset();
        std::swap(d_ptr_p, expiring.d_ptr_p);
        return *this;
    }

    T& operator*() const           { return *d_ptr_p; }  // dereference
    T* operator->() const noexcept { return d_ptr_p; }   // pointer

    explicit operator bool() const noexcept
    { return d_ptr_p != nullptr; }                       // conversion to bool

    T* release()  // Release ownership of d_ptr_p without deleting it.
    {
        T* p    = d_ptr_p;
        d_ptr_p = nullptr;
        return p;
    }

    void reset()  // Clear the value of this object.
    {
```

```
        T* p    = d_ptr_p;
        d_ptr_p = nullptr;
        if (p != nullptr) { delete p; }
    }
};
```

Even this straightforward implementation can then be used for a variety of purposes. With little overhead, RAII principles can be used for managing heap-allocated objects. `UniquePtr<T>` also meets the requirements for being placed in a standard container, letting it be used to build containers of types that are not themselves eligible to be placed in a container, such as creating a `std::vector` of nonmovable noncopyable objects. Marking the move operations **`noexcept`** will also allow standard containers to provide the strong exception guarantee even though our `UniquePtr` cannot be copied; see Section 3.1."**??**" on page **??**.

## Passing around resource-owning objects by value

owning-objects-by-value

**Passing movable objects by value** Prior to the introduction of move semantics, if we passed objects around by value, we would incur the cost of a lot of copies. With move semantics, this might be less problematic. There are several ways that resource-owning objects can be passed around.

Overload sets consisting of corresponding parameters that are passed by both reference and value, e.g., `poor` in the example below, are possible, but attempts to invoke such a function can result in overload resolution failures and, hence, are not typically useful in practice:

```
void poor(int);       // (1) pass by nonconst  value
void poor(int&);      // (2) pass by nonconst lvalue reference
void poor(int&&);     // (3) pass by nonconst rvalue reference

void testPoor()
{
    int        i;        //   i is a    modifiable lvalue.
    const int  ci  = i;  //  ci is a nonmodifiable lvalue.
    int&       ri  = i;  //  ri is a    modifiable lvalue reference.
    const int& cri = i;  // cri is a nonmodifiable lvalue reference.

    poor(3);    // Error, ambiguous: (1) or (3)
    poor(i);    // Error, ambiguous: (1) or (2)
    poor(ci);   // OK, invokes (1)
    poor(ri);   // Error, ambiguous: (1) or (2)
    poor(cri);  // OK, invokes (1)
}
```

In the next example, we see the benefits of an overload set, e.g., `good` in the example below, that consists of corresponding parameters declared as both a nonmodifiable *lvalue* reference (**`const`** `T&`) and a modifiable *rvalue* reference (`T&&`):

```
struct S  // some UDT that might benefit from being "moved"
{
```

```
    S();          // default constructor
    S(const S&);  // copy constructor
    S(S&&);       // move constructor
};

int good(const S& s);  // (4) binds to any S object, but with lower priority
int good(S&& s);       // (5) binds to movable S objects with high priority

void testGood()
{
    S        s;        //   s is a    modifiable lvalue.
    const S  cs  = s;  //  cs is a nonmodifiable lvalue.
    S&       rs  = s;  //  rs is a    modifiable lvalue reference.
    const S& crs = s;  // crs is a nonmodifiable lvalue reference.

    good(S());                 // OK, invokes (5) - guts of S() available
    good(s);                   // OK, invokes (4) - read only
    good(cs);                  // OK, invokes (4) - read only
    good(rs);                  // OK, invokes (4) - read only
    good(crs);                 // OK, invokes (4) - read only
    good(static_cast<S&&>(s)); // OK, invokes (5) - guts of s available
}
```

In the example code above, we have called function `good` with six different expressions involving the (unqualified) user-defined type `S`. Notice that passing anything other than the modifiable *prvalue* `S()` or the modifiable *xvalue* `static_cast<S&&>(s)` invokes the overload (4) which accepts the object by **`const`** *lvalue* reference, not modifying it. If a copy is needed internally within the function `good`, it can be made in the usual way using `S`'s *copy* constructor.

If the object being passed is either a *temporary* or an explicitly cast, unnamed *xvalue*, overload (5) is invoked and the object is passed as a non**`const`** *rvalue* reference. If a "copy" is needed internally, it can now be made safely — and *perhaps*[16] more efficiently — using `S`'s *move* constructor, but see *Potential Pitfalls — Failure to* `std::move` *a named rvalue reference* on page 66.

Let's now consider the alternative of having just one overload e.g., `func` in the example below, in which a potentially *movable*, e.g., user-defined or generic, type is passed *by value*:

```
template <typename T>
int func(T t);  // (6) single "overload" that binds to any T object

void testFunc()
{
    S        s;        //   s is a    modifiable lvalue
    const S  cs  = s;  //  cs is a nonmodifiable lvalue
```

[16]When data that is initially proximate in the virtual address space is allowed to *diffuse* due to either deallocation/reallocation or a *move* operation, *locality* of reference can suffer. Depending on the relative frequency with which the moved data is subsequently accessed, overall performance may be served by performing a *copy* instead, even when the *move* operation itself would be faster. We plan to discuss memory allocation and, specifically, *diffusion*, in **?**.

```
    S&       rs  = s;  //  rs is a    modifiable lvalue reference
    const S& crs = s;  // crs is a nonmodifiable lvalue reference

    func(S());                  // OK, invokes (6) - constructed in func
    func(s);                    // OK, invokes (6) - copied into func
    func(cs);                   // OK, invokes (6) - copied into func
    func(rs);                   // OK, invokes (6) - copied into func
    func(crs);                  // OK, invokes (6) - copied into func
    func(static_cast<S&&>(s));  // OK, invokes (6) - moved into func
}
```

A function (template), such as `func` in the example above, that accepts a movable object *by value* behaves, in some ways, similarly to the more traditional two-overload set, e.g., `good` in the example above. If the object being passed in is a *prvalue* (i.e., a not yet constructed temporary), it can be constructed *in place* as a local variable with no copy or move overhead at all. If the object is an *xvalue* (i.e., it already exists either as an unnamed *temporary* or as the result of an explicit cast to an unnamed *rvalue* reference), then `S`'s move constructor will be invoked to "copy" it. In all other cases, `S`'s copy constructor will be invoked. The net result is two-fold: (1) From the perspective of the user of `func`, an efficient copy will always be made from the arguments supplied; (2) from the implementer's perspective, a mutable copy for internal use will always be available as an automatic variable.

Passing a potentially movable argument by value to a function is not generally indicated. Unless the *contract* for the function states or implies that a potentially movable argument will necessarily be copied (e.g., by any implementation), passing that argument *by value* might incur gratuitous runtime overhead. In cases where it is appropriate, passing specifically a *prvalue* — e.g., just the first call to `func`, in the example above, which passes `S()` — will cause the object to be constructed within the function itself, thereby avoiding even a *move* operation but has absolutely no runtime performance benefit for any of the other calls where a *glvalue*, i.e., *xvalue* or *lvalue*, is passed.

Passing movable objects *by value* (where applicable) means that only a single function overload need be written. This argument becomes even more compelling when we consider a function taking multiple movable arguments:

```
int good2(const S&   s1, const S&  s2);  // both passed by const lvalue ref
int good2(const S&   s1,       S&& s2);  // passed by const lvalue, rvalue
int good2(      S&&  s1, const S&  s2);  // passed by rvalue, const lvalue
int good2(      S&&  s1,       S&& s2);  // both passed by rvalue reference

int func2(      S    s1,       S   s2);  // both passed by value
```

When passing potentially movable objects *by value* is either not applicable or otherwise undesirable, the general approach is to employ forwarding references (see Section 2.1."**??**" on page **??**) to preserve the value category of the argument through to the implementation:

```
template <typename T1, typename T2, typename T3>
int great3(T1&& t1, T2&& t2, T3&& t3)  // each passed by forwarding reference
```

See *Annoyances* — *Visual similarity to forwarding references* on page 80.

return-by-value

**Return by value** Now that we have covered the ideas behind passing around resource-owning objects by value, we provide more realistic examples of these principles at work.

Our first example illustrates a simple output parameter for a function that creates a temporary filename. We consider two ways in which such a function might be designed. First, we pass in an output parameter by address:

```
void generateTemporaryFilename(std::string* outPath, const char* prefix)
{
    char suffix[8];
    // ... Create a unique suffix.

    *outPath = prefix;
    outPath->append(suffix);
}
```

Alternately, we can return an output `std::string` by value:

```
#include <cstring>  // strlen

std::string generateTemporaryFilename(const char* prefix)
{
    char suffix[8];
    // ... create a unique suffix

    std::string rtnValue;
    rtnValue.reserve(strlen(prefix) + strlen(suffix));
    rtnValue.assign(prefix);
    rtnValue.append(suffix);
    return rtnValue;
}
```

In the first implementation, the caller must create a `std::string` on the stack and pass its address to the function. The second version presents a clearer interface for the caller. In C++03, without move semantics, this second version would have had more allocations and copying, but with *rvalue* references and move semantics, there is only an additional move. This is an example where *rvalue* references and move semantics come into effect and enable us to have interfaces that are easier and more natural for many to use but do not incur the often hidden penalties that these patterns can contain; however, see *Potential Pitfalls — Sink arguments require copying* on page 63 and *Potential Pitfalls — Disabling NRVO* on page 65.

In the second version, we always create a `std::string` object to return, which could become a performance issue if we are repeatedly invoking this function. In cases where a function may be called many times in a loop, a much more efficient alternative is to reuse the same capacity in the string.

## Sink arguments

sink-arguments

A **sink argument** is an argument to a function or constructor that will be retained or consumed. Before C++11, it was common to take sink arguments as **`const&`** and copy

them, e.g., how an `HttpRequest` class might have been written in C++03:

```
class HttpRequest
{
    std::string d_path;

public:
    HttpRequest(const std::string& path) : d_path(path) { }

    // path is a sink argument
    void set_path(const std::string& path)
    {
        d_path = path;
    }
};
```

With that interface, even in C++11, if an *rvalue*, e.g., a temporary, is passed, there is no way to avoid a copy. However, we can support move operations for sink arguments to prevent these unnecessary copies, e.g., writing `HttpRequest` another way:

```
class HttpRequest
{
    std::string d_path;

public:
    HttpRequest(const std::string& path) : d_path(path) { }  // as before

    HttpRequest(std::string&& path) : d_path{std::move(path)} { }

    void set_path(const std::string& path)  // as before
    {
        d_path = path;
    }

    void set_path(std::string&& path)
    {
        d_path = std::move(path);
    }
};
```

In this case, we have provided overloads for *rvalue* references in the constructor and the `set_path` function. Having the extra overloads is optimal for users of `HttpRequest`. Note, however, that this involves code repetition: The logic of the constructor and `set_path` are essentially repeated. In addition, this approach can become cumbersome; to provide this behavior in our classes, we have to write $2^N$ overloads for functions taking $N$ arguments. That is, an overload for each combination of **`const`** *lvalue* and *rvalue* reference types for each argument. Reducing this need for extra code bloat is the motivation for the "pass-by-value and move" idiom:

```
class HttpRequest
{
```

```
    std::string d_path;

public:
    HttpRequest(std::string path) : d_path(std::move(path)) { }

    void set_path(std::string path)
    {
        d_path = std::move(path);
    }
};
```

We achieve close-to-optimal behavior by taking sink arguments by value and unconditionally moving them. This idiom adds only the cost of a limited number of move operations per argument over the fully general case with $2^N$ overloads and, importantly, does not add any extraneous copies.

With this version, if a user passes an *lvalue* to `set_path`, the *lvalue* will be copied in the `path` argument, and then the argument will be moved into the data member: There will be one copy plus one move. If a user passes an *rvalue* to `set_path`, the *rvalue* will be moved into the the `path` argument, and then the argument will be moved into the data member: There will be two moves. In both cases, there is one more move than would be needed for the multioverload implementation.

The perfect-forwarding solution for `HttpRequest`, which will produce all possible overloads for qualified parameters at the cost of needing to be a template, would be:

```
class HttpRequest
{
    template <typename S>
    HttpRequest(S&& path) : d_path(std::forward<S>(path)) { }

    template <typename S>
    void set_path(S&& path)
    {
        d_path = std::forward<S>(path);
    }
};
```

Importantly, taking a sink argument by value will *always* make a copy. When that copy will be retained (such as the initialization of a member variable above), this is no additional cost. When there is a code path where the copy is not retained, this copy becomes unnecessary and would be best avoided; see *Potential Pitfalls — Sink arguments require copying* on page 63.

s-from-sink-arguments

**Factories from sink arguments** Occasionally, a **factory function** is designed to take an object of a particular type and produce a modified version of the same object. Classically, the input to such a function would be taken by **`const&`**, and the output would be a named local variable that would be initialized and eventually returned:

```
std::string to_uppercase(const std::string& input)
{
    std::string result;
```

```
    result.reserve(input.size());

    for (int i = 0; i < input.size(); ++i)  // copy input
    {
        result += toupper(input[i]);        // Modify as we populate result.
    }

    return result;
}
```

This implementation has the downside of making an extraneous copy when passed a temporary:

```
void testToUppercase()
{
    std::string upperHi = to_uppercase("Hi");  // copy twice
    assert(upperHi == "HI");
}
```

Alternatively, the same pattern used for sink arguments can be used to initialize what will be our return value; take a sink argument by value:

```
// by-value version
std::string to_uppercase(std::string input)
{
    for (int i = 0; i < input.size(); ++i)
    {
        input[i] = toupper(input[i]);       // Modify input in-place
    }

    return input;
}

std::string output = to_uppercase("hello");
```

This avoids extraneous copies, at the hopefully reasonable cost of always requiring an extra move. As with the previous sink argument examples, higher maintenance options, which come with different associated compile-time costs, would be to provide both `const std::string&` and `std::string&&` overloads, minimizing both moves and copies, or to reimplement the function as a template with a forwarding reference parameter.

ifying-value-categories

## Identifying value categories

Understanding the rules for which value category a particular expression belongs to can be challenging, and having a concrete tool to identify how a compiler will interpret an expression can be very helpful. Building such a tool requires functionality that will behave in a distinct and observably different fashion for each of the three disjoint value categories: *lvalue*, *xvalue*, and *prvalue*. An operator that has such distinct behavior is the **`decltype`** operator when applied to a non-**id-expression** argument; see Section 1.1."**??**" on page **??**.

When applied to a non-id-expression, `e`, with an underlying type of `T`, **`decltype`** will return the following types.

- If `e` is an *xvalue*, then **`decltype`**`(e)` is `T&&`.
- If `e` is an *lvalue*, then **`decltype`**`(e)` is `T&`.
- If `e` is a *prvalue*, then **`decltype`**`(e)` is `T`.

We can then apply this to various expressions, using `std::is_same` to verify that the type produced by the **`decltype`** operator is what we expect. When passed an id-expression naming an entity, we get the type of that entity, which is not helpful in identifying the value category of the id-expression, so we will always use an additional set of parentheses to obtain only the value-category-based determination of the type produced by **`decltype`**:

```
#include <utility>
#include <type_traits>  // std::is_same

int x = 5;
int& y = x;
int&& z = static_cast<int&&>(x);
int f();
int& g();
int&& h();

// lvalues
static_assert( std::is_same< decltype(( x )),   int& >::value, "" );
static_assert( std::is_same< decltype(( y )),   int& >::value, "" );
static_assert( std::is_same< decltype(( z )),   int& >::value, "" );
static_assert( std::is_same< decltype(( g() )), int& >::value, "" );

// xvalues
static_assert( std::is_same< decltype(( std::move(x) )), int&& >::value, "" );
static_assert( std::is_same< decltype(( std::move(y) )), int&& >::value, "" );
static_assert( std::is_same< decltype(( std::move(z) )), int&& >::value, "" );
static_assert( std::is_same< decltype(( h() )),          int&& >::value, "" );

// prvalues
static_assert( std::is_same< decltype(( 5 )),     int >::value, "" );
static_assert( std::is_same< decltype(( x + 5 )), int >::value, "" );
static_assert( std::is_same< decltype(( y + 5 )), int >::value, "" );
static_assert( std::is_same< decltype(( z + 5 )), int >::value, "" );
static_assert( std::is_same< decltype(( f() )),   int >::value, "" );
```

Note the importance of adding the additional set of `()`s around the expression when it is an id-expression, i.e., just a single qualified or unqualified identifier. For all expressions that are not just id-expressions, an extra pair of `()`s will not alter the type produced by **`decltype`**. Without the extra parenthesization, the reference qualifiers, or lack thereof, of the entity named by an id-expression become part of the type produced by **`decltype`**:

```
static_assert( std::is_same< decltype( x ),   int   >::value, "" );
static_assert( std::is_same< decltype( y ),   int&  >::value, "" );
static_assert( std::is_same< decltype( z ),   int&& >::value, "" );
```

Encapsulating this logic to build a utility will require working with **expressions** as operands. We do not have the ability to do high-level manipulation of expressions in the language, but we can use a lower-level and less-structured tool to do so in this case, building macros to identify value categories of expressions passed to them. To better handle any expression, including those with commas that are not nested within `()`s, we use a new feature that C++11 inherited from the C99 preprocessor, **variadic macros**:

```
#include <type_traits> // std::is_reference, std::is_lvalue_reference,
                       // std::is_rvalue_reference

#include <utility>

#define IS_LVALUE( ... ) \
    std::is_lvalue_reference< decltype(( __VA_ARGS__ )) >::value

#define IS_XVALUE( ... ) \
    std::is_rvalue_reference< decltype(( __VA_ARGS__ )) >::value

#define IS_PRVALUE( ... ) \
   !std::is_reference< decltype(( __VA_ARGS__ )) >::value

template <typename T, typename U>
struct S { };

S<int, long> s = {};

static_assert( IS_LVALUE(  s ),             "");  // OK
static_assert( IS_XVALUE(  std::move(s) ),  "");  // OK
static_assert( IS_PRVALUE( S<int, int>() ), "");  // OK, needs __VA_ARG__
```

Finally, for completeness, we can see how macros can be written to identify the remaining value categories:

```
#define IS_GLVALUE( ... ) (IS_LVALUE(__VA_ARGS__) || IS_XVALUE(__VA_ARGS__))
#define IS_RVALUE( ... ) (IS_XVALUE(__VA_ARGS__) || IS_PRVALUE(__VA_ARGS__))

static_assert( IS_GLVALUE(x),                                          "");  // OK
static_assert( IS_GLVALUE(std::move(x)) && IS_RVALUE(std::move(x)), "");  // OK
static_assert( IS_RVALUE(x + 5),                                       "");  // OK
```

tial-pitfalls-rvalueref

guments-require-copying

## Potential Pitfalls

### Sink arguments require copying

The introduction of move semantics and *rvalue* references has prompted many to try to employ these features wherever possible, but we need to be aware of a number of pitfalls. Perhaps the most important piece of advice is, Do not overuse it.

As we saw in *Use Cases — Sink arguments* on page 58, passing by value and moving can offer benefits. However, we must be careful not to employ this pattern without thinking through the ramifications. However, we must carefully think through the ramifications before

employing this pattern. If you design a class, settle on an implementation, decide to use pass-by-value in the constructor, and then later decide to change the underlying representation, you may wind up with worse performance. If a copy is inevitable in the implementation and there is no chance that will change, passing by value may be beneficial. If ever it is or will be the case that you don't need to make the copy, then providing both `const&` and `&&` overloads or a template using forwarding references is needed to mitigate the risk.

For example, here we have written a class `S` that holds a `std::string` data member. We decide to take a `std::string` by value in the constructor and initialize our data member by applying `std::move` to the argument. Later, we decide to change our implementation to use our own `String` class. Our `String` has a converting constructor that takes a `std::string` and copies it (and does whatever else presumably motivated our change to `String`). If we neglect to update the constructor of class `S` (i.e., it still takes a `std::string` by-value and initializes the member with `std::move` of that string), we will wind up with less-efficient code:

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

class S
{
    std::string d_s;  // initial implementation

public:
    S(std::string s) : d_s(std::move(s)) { }  // sink argument constructor
};

std::string getStr();

int main()
{
    std::string lval;

    S s1(lval);     // copy and move
    S s2(getStr()); // move and move
}
```

In the code above, we incur a copy and a move if we pass an *lvalue* to the `S` constructor, and we experience two moves if we pass a temporary.

Suppose we then change `S` to use our own `String` class, but we neglect to change the constructor:

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

class String
{
public:
    String(const std::string&);  // Copy the contents of string.
};

class S
```

```
{
    String d_s;  // Implementation changed.

public:
    S(std::string s) : d_s(std::move(s)) { }  // Implementation did not change.
};

int main()
{
    std::string lval;

    S s1(lval);     // 2 copies
    S s2(getStr()); // move and copy
}
```

The problem is that now we are copying `lval` once into the argument and again into the `String` data member, thereby copying twice.

Had we written the requisite overloads, we would not be in this situation:

```
class S
{
    String d_s;

public:
    S(const std::string& s) : d_s(s)            { }
    S(std::string&& s)      : d_s(std::move(s)) { }
};
```

So, unless we are absolutely certain that we will never change the implementation of our class, designing our class to take a sink argument by value can be a pitfall.

disabling-nrvo

## Disabling NRVO

Named return value optimization (NRVO) can only occur if the expression being returned is the name of a local variable. If we use `std::move` in a return statement, we are returning the return value of another function, i.e., `std::move`, and not a local variable by name, even though as developers we know that `std::move` is just going to be a cast applied to the argument we provide to it:

```
std::string expectingNRVO()
{
    std::string rtn;
    // ...
    return std::move(rtn);  // pessimization, no NRVO
}

std::string enablingNRVO()
{
    std::string rtn;
    // ...
    return rtn;             // optimization, NRVO possible
```

```
}
```

In the example above, the return value of the function is `std::string`, but, after invoking `std::move`, the return expression is of type `std::string&&`. In general, when returning an object by value, we avoid `std::move`. While moving was once thought to be a faster way to return values from a function, testing validates that this is not the case.[17]

amed-rvalue-reference

## Failure to `std::move` a named *rvalue* reference

It is important to remember that we must use `std::move` on a named *rvalue* reference if we wish to move the referenced object's contents somewhere. Even if a function parameter type is an *rvalue* reference, that parameter — by virtue of having a name — is, in fact, an *lvalue*. If an *rvalue* reference overload has the same implementation as the corresponding **`const`** *lvalue* reference overload, then it will likely be invoking the same *lvalue* overloads of any functions it calls. If the final use of the *rvalue* reference parameter does not employ `std::move`, then the function is likely failing to take advantage of whatever move operations the parameter type provides and is instead falling back to a higher-overhead copy operation.

Consider a large user-defined type, `C`, and an associated API that has a well-designed overload set that takes objects of type `C` by either **`const`** *lvalue* reference or *rvalue* reference:

```
class C { /*...*/ };  // some UDT that might benefit from being "moved"

void processC(const C&);  // lvalue reference overload for processing C objects
void processC(C&&);       // rvalue reference overload for processing C objects

void applicationFunction(const C& c)
{
    // ...
    processC(c);  // OK, invokes const C& overload of processC
    // ...
}

void applicationFunction(C&& c)
{
    // ...
    processC(c);  // Bug, invokes const C& overload of processC
    // ...
}
```

The intent of the second overload of `applicationFunction` was clearly to move the contents of `c` into the appropriate *rvalue* overload of `processC`, but as the function parameter is itself an *lvalue*, the wrong overload is invoked. The proper solution is for the *rvalue* overload of `applicationFunction` to make an *xvalue* out of `c` before passing it to `processC`, since the state of `c` is no longer needed by the function:

```
void applicationFunction(C&& c)
{
    // ...
```

[17] See **?**.

```
    processC(std::move(c();  // OK, invokes C&& overload of processC
    // ...
}
```

-named-rvalue-reference

## Repeatedly calling `std::move` on a named *rvalue* reference

Utilizing `std::move` on *rvalue* reference parameters in a function is necessary but should never be done ubiquitously. Recall that once an object has been moved from, the object's state should be considered unspecified, and, importantly, the object is certainly capable of no longer having the same value it originally had. When applying the same transformation from a **`const C&`** overload of a function to a `C&&` overload, as we did in *Failure to* `std::move` *a named rvalue reference* on page 66, it can be easy to falsely assume that all uses of the *rvalue* reference parameter should be wrapped in `std::move`:

```
void processTwice(const C& c)  // original lvalue reference overload
{
    processC(c);
    processC(c);
}

void processTwice(C&& c)  // naive transformation to rvalue overload
{
    processC(std::move(c));  // OK, invokes C&& overload of processC
    processC(std::move(c));  // Bug, c is already moved-from.
}
```

The proper approach here is to always be aware that `std::move` should be used only when an object's state is no longer needed. Though any non`std::move` call might result in a copy (depending on what `processC` does), using `std::move` solely on the last use of `c` in `processTwice` is the only approach that will keep this overload correct and consistent with the original overload:

```
void processTwice(C&& c)  // fixed rvalue overload
{
    processC(c);             // OK, invokes const C& overload of processC
    processC(std::move(c));  // OK, invokes C&& overload of processC
}
```

-pessimizes-performance

## Returning `const` *rvalues* pessimizes performance

Prior to the introduction of move semantics, marking objects **`const`** when returning by value was sometimes recommended as good practice. For example, a convincing argument was made that applying postfix **`operator++`** to a temporary is not only useless, but also almost certainly a bug.[18] In the specific case of postfix **`operator++`**, because the the operator returns the previous value, an object must be returned, i.e., not a reference but a temporary. It was further recommended that the operator return a **`const`** object to prevent the application of postfix **`operator++`** twice.

[18] See **?**, Item 6, "Distinguish between prefix and postfix forms of increment," pp. 31–34.

The goal was to prevent postfix **operator++** as well as any other non**const** member function from being applied to a returned temporary object:

```
struct A
{
   // ...
   A& operator++();    // prefix operator++
   A operator++(int);  // postfix operator++

};
const A operator+(const A&, const A&);

void test1()
{
    A a, b;
    (a + b)++;  // Error, result of a + b is const A.
}
```

The collateral damage of returning by const value, however, is that operations that seek to make use of the contents of the returned temporary will instead risk silently making additional copies, which can have significant overhead when not expected:

```
void processA(const A&a);  // Copy a and send it off for processing.
void processA(A&& a);      // Move contents of a to be sent for processing.

void test2()
{
    A a, b;
    processA(a + b);  // Bug, invokes processA(constA&)
}
```

Overall, though the advice to return by **const** value produced some minor benefits in averting highly dubious modifications of temporaries prior to modern C++, it is now a form of antipattern that should be avoided. For cases where explicitly calling out operations that should not be invoked on temporaries is helpful, consider using reference qualifiers; see Section 3.1."**??**" on page **??**.

If presented with a library that still chooses to return by **const** value, one potential workaround is to use a **const_cast** to move out of the **const** temporary return value:

```
void test3()
{
    A a, b;
    processA(const_cast<A&&>(a + b));  // OK, invokes processA(A&&)
}
```

## Move operations that throw

operations-that-throw

A move constructor that can throw is not useful in generic contexts where operations seek to provide the strong exception-safety guarantee; i.e., the operation will either succeed or throw an exception while leaving the object in its prior, valid state. Algorithms providing the strong

guarantee would need to copy objects rather than move them, since they need to maintain the ability to unwind their work when an exception is thrown without risking further exceptions. This issue was, in fact, the very reason for the introduction of the **`noexcept`** keyword. Late in the development of C++11, this issue was discovered specifically with `std::vector` reallocations; see Section 2.1."**??**" on page **??**.

re-equivalent-to-copies

## Some moves are equivalent to copies

There is no need to provide move operations for a type for which copying and moving have the same effect, and doing so simply increases the maintenance cost of a type, the cost of compilation, and the risk of errors. In particular, built-in types do not have move operations, as there is no advantage to employing `std::move` over just copying them. For example, suppose we have a `Date` type comprising three **`int`** fields:

```
class Date
{
    int day;
    int month;
    int year;

public:
    // ...
};
```

There is no added value in writing move operations for `Date` in the example above since copying cannot be optimized via move-like adoption of resources from the source object.

Given a move operation and a copy operation that have identical effects, eliminating the move operation will produce the same results in all situations with half the code. In general, it is best to avoid writing either operation and to let the compiler choose to generate both by following the **rule of zero**, and for a type such as `Date` in the example above, that is likely what we would do.

usly-nonmovable-objects

## Enabling moves on previously nonmovable objects

If a type is designed initially to be nonmovable, an object of that type often uses its address as a form of identity. When this is done, the object's address becomes a **salient attribute** of its value, and is one attribute that decidedly cannot move with an object's internal state if it were to be moved with a move operation.

The advent of move operations in C++11 might invite making all objects movable in some fashion, but doing so is actively dangerous for any object whose address is shared externally. The motivation to make such objects movable is often to put them in containers or to pass ownership of them around after construction and initialization. The cleanest answer in most cases is not to make the objects themselves movable, as that would require redesigning their interaction with all other components, but to instead manage instances of these nonmovable objects through a smart pointer, such as `std::unique_ptr` or `std::shared_ptr`, creating them in one location and then passing around the owning handle to the object with minimal overhead few limitations.

on-moved-from-objects

## Inconsistent expectations on moved-from objects

When creating a type that supports move operations, a key decision that needs to be made is in what states moved-from objects of that type may be left and what operations will be valid on such objects. When writing code that uses a movable type, especially generic code, it is equally important to understand and document what the requirements are on the template parameters that are supported. When a generic type has higher expectations for what can be done with moved-from objects than are actually supported, possibly subtle runtime bugs will likely arise. Importantly, this conundrum is entirely about objects in a certain state (moved-from) not being valid for certain operations (e.g., destruction, copying, assignment, comparison, user-defined utility functions, and so on), which is entirely a runtime property, and, hence, a source of runtime bugs that are potentially very hard to track down.

The choice of which of a type's operations should be valid on moved-from objects of that type has numerous ramifications on the user-friendliness of the type and what algorithms will safely work with the type. Let's explore, with example types that manage a simple heap-allocated **`int`** in different ways, five various choices that can be made regarding support for objects in a moved-from state. While these examples all fail in a common manner that is seemingly easily alleviated (dereferencing a **`nullptr`**), the structure of what works and what doesn't for these types will often occur in much larger contexts, and the considerations of what operations a type can and should support for moved-from objects apply equally to more involved scenarios.

1. The C++ language makes no explicit requirements that any operation in particular should be valid for moved-from objects. This freedom leads to the possibility for implementing a type that supports no operations on moved-from objects of that type, including destruction. Our first example type, `S1` in the example below, was originally written with the assumption that a heap-allocated **`int`** resource was always owned by every `S1` object. At a later time, move operations were added to `S1` that leave moved-from objects no longer managing a resource, and, furthermore, all operations were modified to have **undefined behavior** when invoked on moved-from objects. A misguided attempt to always set the value of a heap-allocated **`int`** to `-1` prior to its being deleted then makes even the destructor invalid for moved-from objects:

```
class S1
{
    int* d_r_p;  // owned heap-allocated resource

public:
    S1() : d_r_p(new int(1)) { }  // allocate on construction

    S1(const S1& original)
    : d_r_p(new int(*original.d_r_p)) { }  // no check for nullptr

    ~S1() { *d_r_p = -1; delete d_r_p; }   // "    "    "     "

    S1& operator=(const S1& rhs)
    {
        *d_r_p = *rhs.d_r_p;   // no check for either nullptr
```

```
        return *this;
    }

    void set(int i) { *d_r_p = i; }     // no check for nullptr
    int get() const { return *d_r_p; }  // "    "    "     "


    S1(S1&& expiring) : d_r_p(expiring.d_r_p)
    {
        expiring.d_r_p = nullptr;  // expiring now invalid for most operations
    }

    S1& operator=(S1&& expiring)
    {
        *d_r_p = -1;  // no check for nullptr
        delete d_r_p;
        d_r_p = expiring.d_r_p;
        expiring.d_r_p = nullptr;  // expiring now invalid for most operations
        return *this;
    }
};

void test1()
{
    S1 s1;
    S1 s2 = std::move(s1);  // OK, s1.d_r_p == nullptr
    s1.set(17);             // Bug, dereferences nullptr s1.d_r_p
}  // destruction of s1 dereferences nullptr
```

A type such as `S1` becomes very difficult to use in many places where an implicit move might occur:

```
S1 createS1(int i, bool negative)
{
    S1 output1, output2;
    output1.set(i); output2.set(-i);
    return negative ? output2 : output1;  // no NRVO possible
}

void test2()
{
   S1 s;
   s = createS1(17, false);  // creates rvalue temporary and move-assigns to s
                             // destruction of temporary dereferences nullptr
}
```

In general, a type with an unforgiving state like that of a moved-from `S1` object is possible, but using it without the greatest of care is difficult. Most object creation, by design, leads to invocation of destructors, and many common programming constructs can lead to the creation of temporaries that are then moved from and destroyed. The

only advantage of intentionally designing a type of this sort is that it pays no cost in checks for **`nullptr`** to support the moved-from state.

2. The primary downside of the fully unforgiving moved-from state can be alleviated by only making the destructor safe to invoke on a moved-from object:

```
class S2
{
    // ...          (identical to S1 above)

    ~S2() { delete d_r_p; }  // Safe to use if d_r_p == nullptr
};
```

While silent use of `S2` temporaries will not directly result in bugs, this minimal support for the moved-from state still leaves `S2` unusable in a number of algorithms. Consider the following use of `std::swap` on `S2` objects, an operation performed internally by many standard algorithms:

```
void test3()
{
    S2 a, b;
    std::swap(a, b);  // Bug!
}
```

Internally, the invocation of `std::swap` would expand:

```
void test4()
{
    S2 a, b;
    S2 temp = std::move(a);  // OK, a.d_r_p == nullptr
    a = std::move(b);        // Bug, dereferences nullptr a.d_r_p, and
                             // also b.d_r_p == nullptr.
    b = std::move(temp);     // Bug, dereferences nullptr b.d_r_p
}
```

Note that, even though `S2` is move-constructible and move-assignable, `std::swap` has undefined behavior when applied to `S2` objects. Supporting only destruction and no other operations allows for basic use of a type but still fails to work correctly with even the simplest of standard algorithms.

3. To enable our type to be used with `std::swap` and, consequently, many common algorithms, the *copy* and *move* assignment operators can be made safe for objects in the moved-from state. This allows algorithms such as `std::swap` and many algorithms that either rely on `std::swap` or directly move objects around within a container to safely work with objects that have been previously moved from:

```
class S3
{
    // ...          (identical to S2 above)

    S3& operator=(const S3& rhs)
```

```
    {
        if (d_r_p == nullptr)
        {
            d_r_p = new int(*rhs.d_r_p);  // no check for rhs.d_r_p == nullptr
        } else {
            *d_r_p = *rhs.d_r_p;          // no check for rhs.d_r_p == nullptr
        }
        return *this;
    }

    S3& operator=(S3&& expiring)
    {
        delete d_r_p;
        d_r_p = expiring.d_r_p;
        expiring.d_r_p = nullptr;  // expiring now in moved-from state.
        return *this;
    }
};
```

With the assignment operators now modified to support assignment *to* an object in the moved-from state (but not necessarily *from* an object in the moved-from state), we can now safely use `std::swap` and build algorithms on top of that:

```
void test5()
{
    S3 a, b;
    std::swap(a, b);
}

void sort3(S3& a, S3& b, S3& c)
{
    if (a.get() > b.get()) std::swap(a, b);
    if (b.get() > c.get()) std::swap(b, c);
    if (a.get() > b.get()) std::swap(a, b);
}
```

Frustratingly, the moved-from state of `S3` is not valid for all operations expected of an element of a standard container, so `S3` is not supported in any standard container; see *Annoyances — Standard Library requirements on a moved-from object are overly strict* on page 81.

That an `S3` object in the moved-from state cannot itself be moved means all objects of unknown provenance must be treated with great care. Any object that client code might have moved from cannot be used for any purpose other than as the target of an assignment, and, in the case of `S3`, there is not even a way to safely identify if an object has been moved from:

```
void test6(const S3& inputS)
{
    S3 localS = inputS;   // UB if inputS is in moved-from state
```

```
}
```

This could be addressed by simply giving a function, such as `test6`, a **narrow contract** prescribing that its argument must not be in the moved-from state. The difficulty is that this stipulation cannot be enforced at compile time and might be hard to diagnose at run time. The moved-from state can also make otherwise **wide-contract** operations on a container into a source of problems if an element is put in the "poisonous" moved-from state:

```
#include <vector>  // std::vector
void test7()
{
    std::vector<S3> vs1;  // OK
    vs1.push_back(S3());  // OK
    vs1.push_back(S3());  // OK

    S3 s = std::move(vs1[0]);   // OK

    std::vector<S3> vs2 = vs1;  // Bug, copying moved-from vs1[0]
}
```

4. Fully supporting moving objects in the moved-from state removes a significant source of pitfalls when dealing with objects of unknown provenance:

```
class S4
{
    // ...          (identical to S2 above)

    S4& operator=(const S4& rhs)
    {
        if (rhs.d_r_p == nullptr)
        {
            delete d_r_p;
            d_r_p = nullptr;
        }
        else if (d_r_p == nullptr)
        {
            d_r_p = new int(*rhs.d_r_p);
        }
        else
        {
            *d_r_p = *rhs.d_r_p;
        }

        return *this;
    }

    S4& operator=(S4&& expiring)
    {
```

```
        if (expiring.d_r_p == nullptr)
        {
            delete d_r_p;
            d_r_p = nullptr;
        }
        else if (d_r_p == nullptr)
        {
            d_r_p = expiring.d_r_p;
            expiring.d_r_p = nullptr;
        }
        else
        {
            *d_r_p = *expiring.d_r_p;
            delete expiring.d_r_p;
            expiring.d_r_p = nullptr;
        }

        return *this;
    }
};
```

This additional support for use of the moved-from state allows basic algorithms to manipulate collections of objects with no concern for their value or whether they are in a moved-from state. In general, though, without a way to identify the moved-from state, it is still not viable to make use of objects of unknown provenance. Before considering altering a type's functionality to make more operations valid for objects in the moved-from state, see *Potential Pitfalls — Requiring owned resources to be valid* on page 77.

5. Making additional user-defined operations usable for objects in a moved-from state can be done in a number of ways. The most common guidance and that expected by the Standard Library containers is for the moved-from state to be **valid but unspecified**; i.e., all operations that have wide contracts can still be invoked on objects in the moved-from state, but there is no guarantee what results those operations will have. We can adjust the remaining operations of `S4` accordingly:

```
class S5a
{
    // ...          (identical to S4 above)

    void set(int i)
    {
        if (d_r_p == nullptr)
        {
            d_r_p = new int(i);
        }
        else
        {
            *d_r_p = i;
```

```
        }
    }

    int get() const
    {
        return (d_r_p == nullptr) ? std::rand() : *d_r_p;
    }
};
```

`S5a` in the example above is the first type that meets the full requirements for being an element in a standard container. On the other hand, calling `get()` on a moved-from object and making use of that value is likely a sign of a bug, and `S5a` does nothing to facilitate identifying that bug.

An alternate approach is to make the moved-from state fully specified, which we could do by replacing the call to `std::rand()` above by a fixed return value, such as `0`. This attempt to have a reliable moved-from state can lead to confusion, as it cannot always be determined if a move, a copy, or nothing has happened when a move has been requested:

```
class S5b
{
    // ...          (identical to S5a above)

    int get() const
    {
        return (d_r_p == nullptr) ? 0 : *d_r_p;
    }
};

void mightMove(S5b&&);  // function that might move from its argument

void test8()
{
    S5b s;
    s.set(17);
    mightMove(std::move(s));
    assert(s.get() == 0);  // Bug, if mightMove did not actually move.
}
```

Consider the example of the standard containers themselves. A `std::vector` that has been moved from either will be empty or will maintain its original value. All of the wide-contract operations of `std::vector`, e.g., `push_back` or `size()`, can be applied to a `std::vector` that has been moved from. These operations can, in turn, be used to identify the full state of the object and check the preconditions of all of the other operations of `std::vector`, e.g., `front` or `operator[]`.

The various options available for what a type might support for moved-from objects must be matched to the requirements any given algorithm has on the types it makes use

of. This applies broadly to both concrete algorithms using types supplied by other libraries and generic algorithms using types that have not yet been written.

The most general approach is to require the minimum functionality from a type and to require only that functionality of the values that will actually be passed to a particular algorithm. This choice can lead to narrow contracts requiring that a client not pass in objects in a moved-from state but maximizes the flexibility available to the client as to what they need to support.

The most restrictive approach and the one taken by the Standard Library is to require all moved-from objects be in a valid state. This choice can make it far less likely for undefined behavior when combining an algorithm having these requirements with an arbitrary type also meeting these requirements, but it significantly inhibits the ability for code **sanitizers** and other debugging tools to detect bugs when a moved-from object is being used.

When writing a type that will be used in a wide variety of scenarios, failing to meet the broadest possible set of requirements is often risky unless there is a compelling reason to do so. An algorithm is maximally applicable when it has the fewest possible requirements on the types it will work with.

d-resources-to-be-valid

## Requiring owned resources to be valid

Objects that manage resources and support move operations will generally transfer ownership of their owned resource to the moved-to object when possible in lieu of somehow duplicating the owned resource. Fundamental to the design of such a resource-owning type that might move is deciding what the moved-from state should be and whether the moved-from state should also own a resource. Often, this moved-from state can match the default-constructed state and involves very similar trade-offs. Maintaining as an invariant that a resource is always owned can bring with it significant costs, namely the cost of acquiring a resource even if it will never be used. This price has to be weighed against the advantage of never needing to verify that the resource is there, simplifying some code and avoiding some branches.

Though it doesn't own resources outside of its own footprint, an important type worth considering is a very common one, **`int`**, or, in general, any of the various **fundamental types**. Moving from an **`int`** leaves it unchanged, more because of the cheaper cost of leaving a source **`int`** unchanged than it being fundamental to the design. The default-initialized state of an **`int`**, however, is fraught with undefined behavior any time an attempt is made to use its value. This state is, in many ways, similar to a moved-from state that is not valid for any operations other than destruction and being assigned to. The value of an uninitialized **`int`** cannot be used in any meaningful way, and there is no way to query if a particular **`int`** object is properly initialized. This behavior comes with a large advantage of keeping **`int`** trivial and the associated performance advantage of not having to do any writes when creating an **`int`** that will never be read:

```
void populate(int* i);
    // Populate the location pointed to by i with a value.

void test9()
{
   int i;         // OK, leave i uninitialized.
```

```
    populate(&i);  // OK, i is never read by populate.
}
```

The author of a heap-allocating movable type can learn an important lesson from **int** as to what the type's default-constructed state should be and, consequently, what its moved-from state should be. Consider the type `S4` discussed in *Potential Pitfalls — Inconsistent expectations on moved-from objects* on page 70, which supports assignment and destruction of moved-from objects and no other operations. Rather than have the default constructor allocate, we can instead make the default-constructed state be the same as the moved-from state:

```
class S4b
{
    // ...          (identical to S4 above)

    S4b() : d_r_p(nullptr) { }  // same state as the moved-from state
};
```

This implementation has a big advantage over the versions presented earlier that attempted to have a resource allocated for the default-constructed state in that it avoids that allocation completely. Any situation in which an object is default constructed and then immediately assigned a new value from a different object offers a potentially major performance improvement. The `String` example in *Use Cases — Creating a low-level value-semantic type (VST)* on page 29 achieved the same benefit by using a **sentinel** value with static storage duration for the moved-from and default-constructed states, with slightly different trade-offs and similar benefits.

## Annoyances

annoyances-rvalueref

### RVO and NRVO require a declared copy or move constructor

y-or-move-constructor

To create a factory function for a type that returns objects of that type by value, the type is required to have an accessible *copy* or *move* constructor, either implicitly or explicitly declared. Frustratingly, even if the copy or move is always elided by RVO or NRVO, at least one of the constructors must still be either implicitly generated or have an accessible declaration:

```
class S1  // noncopyable nonmovable type
{
    S1() = default;  // private constructibility needed by factory

public:
    S1(const S1&);             // never defined
    S1& operator=(const S1&);  // never defined

    static S1 factory()
    {
        S1 output;
        return output;
    }
```

```
};

int test1()
{
    S1 s1 = S1::factory();  // OK, links without definition of S1(const&)
    S1 s2 = s1;             // Link-Time Error
    return 0;
}
```

The publicly accessible copy operation needed to facilitate the static `factory` function, however, will cause link-time errors in any code that *does* attempt to copy an object of the noncopyable `S1` type. This delay until link time of what ideally should be a compile-time error can make use of types like this burdensome.[19] Move operations slightly mitigate this annoyance as declaring, but not defining, move operations, as in `S2` in the example below, instead of copy operations, as in `S1` in the example above, will both suppress implicit copy operations and make attempting to copy (but not move) objects a compile-time error:

```
class S2  // noncopyable nonmovable type
{
    S2() = default;  // private constructibility needed by factory

public:
    S2(const S2&&);             // never defined
    S2& operator=(const S2&&);  // never defined

    static S2 factory()
    {
        return S2{};
    }
};

int test2()
{
    S2 s1 = S2::factory();  // OK, links without definition of S2(const&)
    S2 s2 = s1;             // Error, no copy constructor
    S2 s3 = std::move(s1);  // Link-Time Error
    return 0;
}
```

std::move-does-not-move

## `std::move` does not move

Despite the name, `std::move` does not *move* anything and is simply an unconditional cast to an *rvalue* reference; see *The* `std::move` *utility* on page 20:

```
template <typename T>
void swap(T& t1, T& t2)
```

[19] C++17 introduced guaranteed copy elision not requiring declared copy and move constructors; copy and move constructors and assignment operators can be private or deleted, and factory functions can still be implemented to return such objects by value. C++23 seems likely to extend this guarantee to a limited number of NRVO eligible cases as well.

```
{
    T temp = std::move(t1);
    t1 = std::move(t2);
    t2 = std::move(temp);
}
```

We can note that the invocation of `std::move` did no moving of anything. The invocations of `std::move` just unconditionally cast the arguments to *rvalue* references. The constructor and assignment operator for `T` found through overload resolution that take a single *rvalue* reference to `T`, which might very well be the copy constructor and copy-assignment operator, are what do the work of `std::swap`, and though those might be move operations, nothing about that is guaranteed. This function can be written in a a more verbose, less expressive yet identical way:

```
template <typename T>
void swap(T& t1, T& t2)
{
    T temp = static_cast<T&&>(t1);
    t1 = static_cast<T&&>(t2);
    t2 = static_cast<T&&>(temp);
}
```

`std::move` might have been more expressively named `std::make_movable`, `std::as_xvalue`, or any similar name that conveyed that the qualities of the object are changed, but no action is explicitly being performed.

## Visual similarity to forwarding references

forwarding-references

The syntax for *rvalue* references has been overloaded with the similar but distinct concept of a forwarding reference; see Section 2.1."**??**" on page **??**. In hindsight, having a distinct syntax for forwarding references — even one as possibly distasteful as `&&&` — would allow for a clear distinction, preventing the case of *not* having a forwarding reference when one is intended.

To be a forwarding reference, a parameter's type must be an *rvalue* reference to a function template parameter that is *not* cv-qualified:

```
template <typename T>
void f1(T&& t);  // t is a forwarding reference.
```

Thus, using a class template parameter, adding a **`const`** or **`volatile`** qualifier, or using a concrete type will all make a function parameter an *rvalue* reference and not a forwarding reference:

```
template <typename T>
struct S
{
    void f2(T&& t);    // t is not a forwarding reference.
};

template <typename T>
void f4(const T&& t);  // t is not a forwarding reference.
```

```
void f5(int&& i);      // i is not a forwarding reference.
```

In practice, when implementing **perfect forwarding**, making a mistake in any one of these facets will result in not having a forwarding reference and compilation errors. Being unable to state clearly the intent to have a forwarding reference makes these compilation errors significantly more obtuse.

## Value categories are a moving target

ies-are-a-moving-target

C++98/03 had just *lvalues* and *rvalues*. In the original design of C++11, the only *xvalues* were once *lvalues*. In C++14, members of *prvalue* user-defined types also became *xvalues*. In C++17 even more *prvalues* were identified as *xvalues*. Some of these changes have been adopted as defect reports against older standards, and some have introduced subtle changes in behavior between language standards.

In any case, the progression is in one direction: there were no *rvalues* in C++03 that were not *prvalues* in C++11, and then the demarcation between *prvalue* and *xvalue* continued to drift so that the categories of non*lvalues* that were deemed to be *xvalues* grew. The criteria now is *not* that an *xvalue* is reachable but that it refers to an object in memory and that a *prvalue* is everything and must be a complete type. Note that the direction of motion is one way. Once something is an *xvalue* in the Standard, it can never go back. Understanding the evolution is helpful to understanding how the C++ language is evolving; see the *Appendix — The evolution of value categories* on page 87.

Overall, what the literature has lacked and the Standard's evolution has made difficult to understand is a clear designation of what the value categories are and what their purpose is. The realization that the *xvalue* category needed to encompass all objects whose data is no longer needed, whether due to being a temporary whose lifetime is ending or due to an explicit cast in code, took a great deal of time to clarify, and the various edge cases have only slowly been clarified.[20]

## Standard Library requirements on a moved-from object are overly strict

oject-are-overly-strict

**By Sean Parent**

Given an object, `rv`, of type `T` that has been moved from, the C++14[21] Standard specifies the required postconditions of a moved-from object[22]:

> rv's state is unspecified [*Note*: rv must still meet the requirements of the library component that is using it. The operations listed in those requirements must work as specified whether rv has been moved from or not. — *end note*]

The requirement applies to both move construction and move assignment for types used with the Standard Library containers and algorithms. The note is not **normative** but does clarify that the requirements on a moved-from object are not relaxed.

[20] Similarly, while the distinction between a *prvalue* and an *xvalue* is largely academic prior to C++17, with the adoption of proposal P0135R0 (**?**), the distinction is heavily leveraged for guaranteed copy elision.
[21] Similar wording with the same intent appears in every version of the C++ Standard since C++11.
[22] **?**, Table 20, p. 427

To understand how this requirement causes an issue in practice, consider the following simple class definition. The intent of `my_type` is to create a class that always holds a valid value, is **copyable** and **equality comparable**, and happens to contain a remote part. The remote part in this example is held as a `std::unique_ptr` to an `implementation` object. A remote part might be employed to improve compile times by separating the implementation from the interface, to allow a polymorphic implementation using inheritance, or to trade off a slower copy for a faster move:

```
class implementation;  // forward declaration

class my_type
{
    std::unique_ptr<implementation> d_remote;  // remote part

public:
    explicit my_type(int a)
    : d_remote{std::make_unique<implementation>(a)}
    { }

    my_type(const my_type& a)
    : d_remote{std::make_unique<implementation>(*a.d_remote)}
    { }

    my_type& operator=(const my_type& a)
    {
        *d_remote = *a.d_remote;
        return *this;
    }

    friend bool operator==(const my_type& a, const my_type& b)
    {
        return *a.d_remote == *b.d_remote;
    }
};
```

We can add the ability to move the object by using a default move constructor and move-assignment operator:

```
class my_type
{
    //...
public:
    //...
    my_type(my_type&&) noexcept = default;
    my_type& operator=(my_type&&) noexcept = default;
    // ...
};
```

If we ignore the library requirements and consider only the language requirements, this implementation is sufficient. The only language requirement is that a moved-from object

be destructible because, without a cast, the only operation the compiler will perform on a moved-from object is to destroy it. By definition, an **rvalue** is a temporary object, and no other operations will be performed. The assignment `a = f()` where `a` is of type `my_type` and `f()` returns a value of type `my_type`, will work correctly with the default member-wise implementations.

However, using `my_type` in a standard container or algorithm will likely fail. Consider inserting an element into a vector at a position, `p`:

```
void test1()
{
    my_type a{42};
    std::vector<my_type> v;
    //...
    v.insert(p, a); // undefined behavior
}
```

If `p` is not at the end of `v`, the implementation of `std::vector` may move the range of elements `[p, end(v))` and then copy `a` over a moved-from object. Implementations of the Standard Library may use a different approach to implementing `insert` that would not encounter this issue.[23] The copying of `a` results in a statement with the effect of `*p = a` where `*p` is a moved-from instance of `my_type`. The copy operation is likely to crash because of the implementation of copy assignment:

```
my_type& my_type::operator=(const my_type& rhs)
{
    *d_remote = *rhs.d_remote;
    return *this;
}
```

Following the move of the range of elements starting at `p`, `d_remote` of `*p` is equal to **`nullptr`**, and dereferencing `d_remote` has undefined behavior. There are multiple ways to fix the copy-assignment operator; for illustration purposes, we'll simply add a conditional to test `d_remote` and, if it is equal to **`nullptr`**, use an alternative implementation:

```
my_type& my_type::operator=(const my_type& rhs)
{
    if (d_remote == nullptr)
    {
        *this = my_type(rhs);  // copy-construct and move-assign
    }
    else
    {
        *d_remote = *rhs.d_remote;
    }

    return *this;
}
```

[23] The 11.0.1 version of the `libc++` Standard Library does use the described approach and will result in a crash.

The additional check is sufficient to make *all* of the standard containers and algorithms work correctly. Unfortunately, this check is not sufficient to satisfy a strict reading of the Standard's requirements for element types.

- Copy construction from a moved-from object will fail.
- Copy assignment from a moved-from object will fail.
- Equality will fail if either operand has been moved from.

All of these operations would cause a **`nullptr`** to be dereferenced. The Standard Library states that these operations must be valid for *all* values of a given type.

The implementations of functions associated with the containers and algorithms in the Standard Library will never perform any operation on a moved-from object other than to destroy it or assign a new value to it *unless* called with an object that has already been moved from, i.e., by the caller directly. The operations in the list above will never be invoked.

The `std::swap` algorithm imposes one additional requirement. Consider swapping a value — e.g., `std::swap(a, a)` — with itself:

```
void test2()
{
    my_type a;
    // inlined std::swap(a, a):
    my_type tmp = std::move(a);
    a = std::move(a); // self-move-assignment of a moved-from object
    a = std::move(tmp);
}
```

The statement `a = std::move(a)` is doing a self-move-assignment of a moved-from object. The default move-assignment in the above implementation of `my_type` will work correctly for self-move-assignment of a moved-from object. The default implementation satisfies the postconditions for both the right-hand and left-hand arguments and does not affect the value of `a`. The left-hand argument of move assignment must be equal to the prior value of the right-hand argument. The containers and algorithms in the Standard Library do not self-swap objects, but `std::swap` annoyingly provides the guarantee that self-swap will work if the arguments satisfy the requirements for the *move constructible* and *move assignable* concepts. The requirement for self-swap is both a legacy requirement from when `std::swap` was implemented in terms of copy and follows from a general requirement in the Standard that, unless otherwise specified, operations should work even if reference arguments alias each other in whole or in part. There is no known value in supporting self-swap, and a self-swap usually indicates a defect in the algorithm.

Adding the additional checks to satisfy the Standard's wording has an otherwise unnecessary performance impact and proves to be error-prone to implement. Beyond that, the additional code introduces a new *empty* state for `my_type`, which must be considered if we introduce an ordering with **`operator<()`** or any other operation the Standard Library may invoke. The gratuitously induced empty state defeats the purpose of value semantics because coding with an object that may or may not be empty is equivalent to coding with a pointer that may or may not be null.

The root cause of this issue is broader than just the postconditions of move operations. There is a Standard proposal to address these issues.[24] Until the proposal is adopted (and it may not be), a type must include these additional checks to adhere to the standard requirements.

ack-of-destructive-move

## Lack of destructive move

As far back as Hinnant's proposal,[25] the paper that first brought a complete approach to *rvalue* references to C++ in 2002, a gap was recognized: the lack of a single function that can both move the contents out of an object and also destroy it. The ability to combine moving from an object and destroying that same object into one operation would enable the design of types that do not have a resourceless moved-from state, avoiding the need for many of the considerations brought up in *Potential Pitfalls — Inconsistent expectations on moved-from objects* on page 70.

The complications in providing this form of destructive move or relocation functionality are numerous, and no refined proposal for a complete solution, that we are aware of, has come forward in the years since *rvalue* references were first officially proposed. A complete solution would need to address at least three items. 1. A syntactic and semantic mechanism to distinguish this new form for destroying an object from other ways in which an object can be passed and not destroyed would need to be designed. 2. The ability to apply operations of this sort to automatic variables would seem necessary to make the cost of another new language feature worth the benefits, but that would require some mechanism to ensure that destructively moved objects can no longer be referenced once destroyed. This would involve potentially complex changes to name-lookup rules and object-lifetime rules. 3. The definition of destructive moves in a class hierarchy would be complicated by the requirement that the destruction of the members and bases of an expiring object and the construction of the corresponding members and bases of a new object must happen in the opposite order and thus cannot be done in parallel. This was, perhaps, one of the biggest unsolved sticking points when Hinnant[26] explored this design space.

Many of these problems are not applicable when relocation can be accomplished with a **trivial copy operation** combined with simply not invoking a destructor on the source object. A surprising number of types meet this criteria since any type that uniquely owns a resource by pointer is a potential candidate, and this includes most common implementations of `std::string`, `std::vector`, and `std::unique_ptr`. Various production platforms have observed and leveraged this behavior, e.g., BDE[27] and Folly.[28] This partial approach to enabling types to support a limited form of destructive move is being considered for standardization.[29] The great benefit of having a trivial operation of this sort is that mass moves of objects between blocks of memory, such as that done by `std::vector` on insertions and

[24] **?**

[25] **?**

[26] **?**

[27] The BDE library from Bloomberg (**?**) identifies types that can support this form of relocation with a user-specializable type trait, `bslmf::IsBitwiseMoveable`, and takes advantage of that trait in many of the containers it provides.

[28] Facebook's Folly library (**?**) has a type trait, `folly::isRelocatable`, that identifies trivially relocatable objects, which is used to advantage in Folly containers such as `fbvector`.

[29] **?**

resizes, can become single invocations of `std::memcpy` with no loss of correctness.

see-also

## See Also

- "??" (§1.1, p. ??) ♦ explains an operator that depends heavily on the value category of its arguments.
- "??" (§1.1, p. ??) ♦
- "??" (§1.1, p. ??) ♦
- "??" (§1.1, p. ??) ♦
- "??" (§2.1, p. ??) ♦
- "??" (§2.1, p. ??) ♦
- "??" (§2.1, p. ??) ♦ describes another use of the double-ampersand (`&&`) syntax, closely related but distinct from *rvalue* references.
- "??" (§2.1, p. ??) ♦
- "??" (§2.1, p. ??) ♦
- "??" (§2.1, p. ??) ♦
- "??" (§3.1, p. ??) ♦
- "??" (Section 3.1, p. ??) ♦ explains a feature that allows for overloading member functions on the value category of the object they are invoked on.

further-reading

## Further Reading

- For the definitive retrospective on value category naming in C++11 by Stroustrup himself, see ?.
- For the trail of papers that introduced move semantics, *rvalue* references, and the refined C++11 value categories, start with N1377 (?) and continue to N3055 (?). Produced in 2006 during the evolution of the feature, N2027 (?) gives an overview of the basics and cites many of the papers that contributed to how the feature took shape.
- For a thorough treatment of reference declarations in C++, see ?.
- For a solid treatment of the theory value semantics along with its practical applications, see ? and ?.
- *Effective Modern C++* (?) discusses value categories, *rvalue* references, move semantics, and perfect forwarding in the way that only Scott Meyers can.
- *C++ Move Semantics — The Complete Guide* is a recent attempt by a world-renowned author to capture all things related to move semantics, including value categories, *rvalue* references and perfect forwarding; see ?.

appendix-rvalueref

# Appendix

## The evolution of value categories

**What is a value category?** In C++, we use declaration statements to introduce named objects and functions into a scope:

```
const int i = 5;  // variable i of type const int having the value 5
double d = 3.14;  // variable d of type double having the value 3.14
double* p = &d;   // variable p of type double* holding the address of d
char f();         // function f returning a value of type char
enum E { A } e;   // variable e of type E enumerating A
```

We can then combine these functions and objects along with literals to form expressions. Some of these expressions might identify an object, and these expressions are all collectively known as *lvalues*:

```
i        // a nonmodifiable int value whose address can be taken
(i)      // "  "      "       "     "      "      "      "    "   "
d        // a modifiable double value whose address can be taken
p        // "      "     double*  "     "      "      "    "   "
*p       // "      "     double   "     "      "      "    "   "
e        // a modifiable E value whose address can be taken
```

The "l" in *lvalue* is often taken to mean "left" since these expressions can all conceivably appear on the left-hand side of an assignment operator. Even expressions with **const**-qualified type, which actually makes them ineligible to be the target of assignment, are considered *lvalues* since they identify an object in memory. Another common interpretation of the "l" is "live" since the objects, in general, reside in memory throughout the duration of their lifetime. However, being an *lvalue* is a compile-time property that is not dependent on the runtime value of the expression; for example, even if an expression dereferences a null pointer, it is still considered an *lvalue*.

All other expressions are then collectively known as non*lvalues*. Often, this category is identified as *rvalues*, with the "r" taken to mean "right" since these expressions are those that can appear on the right-hand side of an assignment operator:

```
5        // int value whose address cannot be taken
(i + 1)  //   "     "     "      "        "     "   "
(d + i)  // double value whose address cannot be taken
f()      // char value whose address cannot be taken
f() + 1  // int    "     "               "     "   "
A        // E value whose address cannot be taken
```

Each of these non*lvalue* expressions identifies a value but not necessarily an object that resides in memory. All *lvalues* can also be implicitly converted to *rvalues*, referred to as ***lvalue*-to-*rvalue* conversion**, which is how the value in an *lvalue* is accessed. Another common interpretation for the "r" in *rvalue* is "read-only," as these values can be used to initialize other objects but cannot generally be modified.

tegories-prior-to-c++11

**Value categories prior to C++11** Early on — well before C++ — the classic, pre-Standard C programming language had already made the distinction between *lvalues* and

non*lvalues*, a.k.a *rvalues*.[30] In that characterization, the "l" in *lvalue* stood for "left" (as in what could appear on the left side of an assignment operator) and, similarly, the "r" in *rvalue* stood for "right," as in what could appear on the right side. Along with the introduction of ANSI C,[31] the common characterization of an *lvalue* had evolved: "L" had come to stand for where the object "lives," as in *object identity*; a value that is not associated with physical storage, i.e., one having a unique address, is referred to as a non*lvalue*. One could think of an *lvalue* as an expression — e.g., a named variable, an element of an array, or a field of a **`struct`** or **`union`** — whose address could be taken, e.g., using the unary address-of operator `&`.

C also identified a third category, **function designator**, that — except when used as the operand of `&` (address-of operator), **`sizeof`**, and `_Alignof` — **decays**, i.e., is converted automatically, to the non*lvalue* address of the designated function, much like how a C-style array decays to the address of its first element. As all function-like behavior became part of the C++ type system, anything C identified as a function designator became a nonmodifiable *lvalue* in classical C++.

C++ restored the term "*rvalue*," replacing "non*lvalue*," to refer to any value not associated with physical storage, e.g., an arithmetic literal, enumerator, or nonreference value returned from a function.

ations-prior-to-c++11

**Lvalue reference declarations prior to C++11** C++ introduced the notion of an *lvalue* reference that can be **`const`** or non**`const`**.[32] *Lvalue* references allow for giving a name to the result of an ***lvalue* expression**, and later that reference could be used anywhere the *lvalue* expression could be used:

```
      int     i;       // modifiable lvalue
const int    ci = 5;   // nonmodifiable lvalue initialized to value 5

      int&  ri1 =  i;  // OK
      int&  ri2 = ci;  // Error, modifiable reference to const object
const int& rci1 =  i;  // OK
const int& rci2 = ci;  // OK
```

The original use case motivating references in C++ was to declare overloaded operators for user-defined types:

```
struct Point  // user-defined value type
{
    int d_x;
    int d_y;
    Point(int x, int y) : d_x(x), d_y(y) { }  // value constructor
};

Point operator+(const Point& lhs, const Point& rhs)
```

[30] **?**, Appendix A, section 5, "Objects and lvalues," p. 183

[31] **?**, Appendix A, section A.5, "Objects and Lvalues," p. 197

[32] Independently of whether the reference is declared **`const`**, it can also be declared **`volatile`**; similar to **`const`** qualifiers and *pointer* variables, a non**`volatile`** *reference* may not be initialized with the address of a **`volatile`** object. As the **`volatile`** qualifier is seldom used (productively) in practice, we will omit further consideration of it here.

```
    // Return the vector sum of the specified lhs and rhs objects.
{
    return Point(lhs.d_x + rhs.d_x, lhs.d_y + rhs.d_y);
}
```

That said, *lvalue* references in C++ also make it possible for the value returned by a function to be an *lvalue*:

```
Point& singletonPoint()  // Scott Meyers is known for this pattern of singleton.
{
    static Point meyersSingleton(0, 0);
    return meyersSingleton;  // Return reference to function-local static Point.
}

Point *address = &singletonPoint();  // address of value returned from function returning lvalue
```

What's more, many expressions involving built-in operators that were considered non*lvalues* in C became *lvalues* in C++:

```
#include <cassert>  // standard C assert macro
void f0()
{
    int x = 1, y = 2;  // modifiable int variables
                       assert(1 == x);  assert(2 == y);
    (x = y) += 1;      assert(3 == x);  assert(2 == y);
    ++x += y;          assert(6 == x);  assert(2 == y);
    --x -= 2;          assert(3 == x);  assert(2 == y);
    x++ *= 3;          // Error, x++ is a nonlvalue (even in C++).
    (y, x) = 6;        assert(6 == x);  assert(2 == y);
    (x ? x : y) = 7;   assert(7 == x);  assert(2 == y);
}
```

As illustrated above, the affected operations include each of the built-in assignment operators (e.g., `x *= 2`), the built-in *prefix* but not *postfix* increment (`++x`) and decrement (`--x`) operators, potentially the comma operator (`x, y`), and potentially the ternary operator (`x ? y : z`):

```
void f1()
{
    int x, y = 0;  // modifiable int variables
    x = 1;         // lvalue in C++ (but not in C)
    x *= 2;        //     "     "  "    "   "   " "
    ++x;           //     "     "  "    "   "   " "
    --x;           //     "     "  "    "   "   " "
    x, y;          //     "     "  "    "   "   " "
    1, x;          //     "     "  "    "   "   " "
    x ? x : y;     //     "     "  "    "   "   " "

    x++;           // nonlvalue in C++ (and C too)
    x--;           //  "     "      "  "    "  "  "
    x, 1;          //  "     "      "  "    "  "  "
    x ? 1 : y;     //  "     "      "  "    "  "  "
```

```
    y ? x : 1;     //  "     "      "  "    "  "  "
}
```

With *lvalue* originally deriving from "left," intuition might lead one to think that being an *lvalue* is fundamentally tied to modifiability through assignment. C++ instead focuses on whether an expression represents an object in memory and whether taking the address of that object is a wise decision. Expressions that resolve to **const** built-in types or to user-defined types that fail to find a matching overload of **operator=** cannot be on the left-hand side of an assignment expression. Interestingly, for admittedly odd, user-defined types with a **const** overload of **operator=**, a non*lvalue* can be placed on the left-hand side of an assignment expression:

```
struct Odd
{
    const Odd& operator=(const Odd& rhs) const { return *this; }
};

void test()
{
    Odd() = Odd();  // rvalue assignment to rvalue

    const int x = 7;
    x = 8;          // Error, cannot assign to const int lvalue
}
```

The address-of operator, &, is the primary language tool that is applicable only to *lvalues*. Any *lvalue* expression identifies an object in memory, generally one that has a lifetime beyond that expression, so the address-of operator is allowed to apply to such expressions:

```
void f2(bool e)
{
     int a = 1, b = 2;   // modifiable integer variables

     &(a);          // OK
     &(++a);        // OK

     &(a + 5);      // Error, a + 5 is a nonlvalue.
     &(a++);        // Error, a++ is a nonlvalue.
     &(b, a);       // OK
     &(e ? a : b);  // OK
}
```

urned-from-a-function

**Temporaries: nonreference values returned from a function** When a function returns an object via a nonreference type, an object might be created in memory whose lifetime will generally end after the statement invoking the function. Expressions that create and refer to such temporary objects are a form of non*lvalue*:

```
double f() { return 3.14; }  // function returning a nonref. type by value
```

The lifetime of a temporary is guaranteed to last until the end of the largest expression in which it is contained, after which the temporary is destroyed:

```
#include <iostream>  // std::cout
void g() { std::cout << "pi = " << f() << '\n'; }  // prints: pi = 3.14
```

If more than one temporary is created within an expression, they are destroyed in the reverse order in which they were created:

```
struct S  // This struct prints upon each construction and destruction.
{
     int d_i;  // holds constructor argument
     S(int i) : d_i(i) { std::cout << " C" << d_i; }  // print: C*i*
     S(const S&) { std::cout << "COPY";  exit(-1); }  // never called!
     ~S()              { std::cout << " D" << d_i; }  // print: D*i*
};

S f(int i) { return S(i); }  // factory function returning S(i) by value

void g()  // demonstrates relative order of ctor/dtor of temporary objects
{
    f(1);              // prints: C1 D1
    f(2), f(3);        // prints: C2 C3 D3 D2
    f(4), f(5), f(6);  // prints: C4 C5 C6 D6 D5 D4
}
```

In the example above, `S` prints a `C*i*` on construction and a corresponding `D*i*` when the object is destroyed. Notice that, despite the factory function constructing and returning an `S` by value, no copy operation occurs due to an optimization known as return-value optimization (RVO). Although neither of the C++11 and C++14 Standards mandate it, this optimization has been implemented by virtually every popular C++ compiler since the early 2000s, provided an accessible *copy* or *move* constructor is declared, even if none is defined.

ry-bound-to-a-reference

**Lifetime extension of a temporary bound to a reference** Recall from *Lvalue reference declarations prior to C++11* on page 88 that *lvalue* references can bind to an *lvalue* expression of matching type. Non**const** *lvalue* references cannot bind to **const** *lvalues*, whereas **const** *lvalue* references can bind to both **const** and non**const** *lvalues*:

```
      int  i;          // nonconst int variable  i
const int ci = 0;      //    const int variable ci

      int&  ri0 =  i;  // OK, nonconst lvalue reference, nonconst lvalue
      int&  ri1 = ci;  // Error, nonconst lvalue reference, const lvalue
const int& cri0 =  i;  // OK, const lvalue reference, nonconst lvalue
const int& cri1 = ci;  // OK, const lvalue reference, const lvalue
```

More interestingly, a **const** *lvalue* reference can bind to *rvalue* expressions of matching type, while a non**const** *lvalue* reference cannot bind to *any rvalue* expression:

```
      int fi();          // function returning an rvalue temporary int
const int fci();         // function returning a const rvalue temporary int

      int&  ri2 =  fi();  // Error, nonconst lvalue ref, nonconst rvalue
```

```
      int&  ri3 = fci();  // Error, nonconst lvalue ref, const rvalue
const int& cri2 =  fi();  // OK, const lvalue reference, nonconst rvalue
const int& cri3 = fci();  // OK, const lvalue reference, const rvalue
```

Whenever a temporary object or a subobject of a temporary object is bound to an *lvalue* reference, the lifetime of the temporary is extended to be that of the reference to which it is bound:

```
struct S  // example struct containing data members and accessor functions
{
    int d_i;                    // integer data member
    int d_a[5];                 // integer array data member
    int i() { return d_i; }     // function returning data by value
    int& ir() { return d_i; }   // function returning data by reference
};

S f()  // example function constructing temporaries of varying lifetimes
{
    S();                         // temporary S destroyed after the semicolon
    const   S& r0 = S();         //     "      "      "     when r0 leaves scope
    const int& r1 = S().d_i;     //     "      "      "     when r1 leaves scope
    const int& r2 = S().d_a[3];  //     "      "      "     when r2 leaves scope
    const int& r3 = S().i();     //     "      "      "     after the semicolon
    const int& r4 = S().ir();    //     "      "      "     after the semicolon

    int i1 = r3;                 // OK, copies from lifetime-extended temporary
    int i2 = r4;                 // Bug, undefined behavior

    return S();                  // temporary S returned as rvalue to caller
}
```

Note that binding a reference to a member keeps the entire temporary object alive. Similarly, binding a reference to an element of an array keeps the array alive, and transitively keeps the complete object alive. There is no such connection between the return value of a function, such as the return values of `i()` and `ir()` above, and the object on which that function is invoked, so the `S` objects used to initialize `r3` and `r4` do not offer anything that extends their lifetimes beyond the statement in which they are created. In the case of `r3`, a temporary **`int`** *is* created, and the lifetime of that **`int`** extends to the end of the scope. In the case of `r4`, the referenced **`int`** is destroyed in the statement where `r4` is initialized, making access through `r4` have undefined behavior anywhere later in the function.

`modifiable-rvalues`

### Modifiable *rvalues*! [AUs: was the ! in the title intended?]

Given *rvalue*'s historical roots in C, some may find it incongruous that an "rvalue" could ever be modified because many used to think the "r" stood for "read-only." Before expounding on modern C++ value categories in the next section, let's observe that modifiability is, and always has been, a separable property of a C++ expression that is largely orthogonal to its value category. User-defined, non**`const`** operators can be invoked on temporary objects capable of modifying — or even relinquishing the address of — such temporaries:

```
struct S
```

```
{
    int d_i;

    S() : d_i(0) { }

    S* addr() { return this; }          // Accessor mostly equivalent to &.
    S& incr() { ++d_i; return *this; }  // manipulator
};

void test()
{
    S* tempPtr = S{}.addr();  // Address of temporary acquired.
                              // tempPtr invalidated.

    int i = S{}.incr().d_i;   // Temporary created, modified, and accessed.
    assert(i == 1);
}
```

Importantly, this sort of access to temporaries does not alter their fundamental nature; they are temporary. Even though the address of a temporary is acquired in `tempPtr` in the example above, the temporary itself will be destroyed after the statement is completed. Similarly, the member variable `d_i` of the temporary in the initializer for `i` is initialized itself, modified, accessed, and then destroyed, all within the same expression.

*Rvalues* of fundamental types in C++03, however, are not modifiable. The reason is two-fold: (1) *rvalues* are not permitted to bind to non**const** *lvalue* references, and (2) fundamental types have no member functions. Hence, all operations that mutate fundamental types behave as if they were passed in as the first argument to a free operator with the first parameter passed as a non**const** *lvalue* reference:

```
// pseudocode illustrating how operators on fundamental types behave

int& operator=(int& lhs, const int& rhs);  // free operator function
    // Assign the value of rhs to the modifiable int object bound to lhs,
    // and return an lvalue reference to lhs.

int& operator+=(int& lhs, const int& rhs);  // free operator function
    // Assign the value that is the sum of rhs and lhs to the modifiable
    // int object bound to lhs, and return an lvalue reference to lhs.
```

In the same way that a member function can be restricted to only apply to **const** objects with a **const** qualifier, in C++11 a member function can be restricted to only apply to *lvalues* with an *lvalue*-reference qualifier; see Section 3.1."**??**" on page **??**. Applying such a reference qualifier to the assignment operator is the only way to get the same behavior for the assignment operator on a user-defined type that exists for fundamental types.

Occasionally, the ability to modify (and, usually, cannibalize) the state of a temporary is eminently useful. This will turn out to be the driving force behind the addition of *rvalue* references and the expansion of value categories to include *xvalues* in C++11.

-we-want-move-semantics

**Rationale: Why do we want move semantics?** Like many engineering solutions, necessity is the mother of invention. Make no mistake: The very notion of an *rvalue* reference

was *invented* as part of a much larger feature engineered to solve a common, objectively verifiable performance problem involving (1) gratuitous memory allocations and deallocations and (2) excessive data copying. Consider the following program that does nothing more than build up a `std::vector` of `std::vector`s of `std::string`s, either by appending to the nested vectors or by inserting them at the front:

```
#include <vector>   // std::vector
#include <string>   // std::string
#include <cstdlib>  // std::atoi, std::abs

int main(int argc, char *argv[])
{
    int k = argc > 1 ? std::atoi(argv[1]) : 8;
    bool front = k < 0;
    int N = 1 << std::abs(k);

    std::string s = "The quick brown fox jumped over the lazy dog.";
        // string value that is too long for short-string optimization

    std::vector<std::string> vs;
    for (int i = 0; i < N; ++i) { vs.push_back(s); }
        // Create an (inner) vector-of-strings *exemplar* of size N.

    std::vector<std::vector<std::string> > vvs;
        // Create an empty vector of vectors of strings to be loaded in two ways.

    for (int i = 0; i < N; ++i)  // Make the outer vector of size N as well.
    {
        if (front)
        {
            vvs.insert(vvs.begin(), vs);  // Insert copy of vs at the beginning.
        }
        else
        {
            vvs.push_back(vs);            // Append copy of vs at the end.
        }
    }

    return 0;
}
```

This program, valid in both C++03 and C++11, behaves very differently with the changes in C++11, and algorithms such as this that were the intended target of the introduction of move operations to the language.

In C++03, the calls to `push_back` will, when needed, grow the internal capacity buffer of `vvs` a logarithmic number of times, e.g., capacity $= 1, 2, 4, \ldots, 2^N$, and copy all of the already-added elements to the new storage on each resize. Alternatively, when inserting at the front, each individual `insert` operation must copy all elements in `vvs` to the next element in the capacity buffer and then put the new element at index `0`.

In C++11, both of these operations are vastly improved by the addition of move semantics to `std::vector`. Independently of the mechanics of the language feature, when a `std::string` or `std::vector` is moved from one location to another, it gives ownership of the allocated data buffer to the target object and leaves the source object empty (with `0` size and capacity and no data buffer). This constant-time operation, consisting of nothing more than the assignment of a small number of fundamental data members replaces a linear operation involving many allocations, comes at the cost of altering the state of the source object. When able, growing the data buffer in C++11 can take advantage of this moving behavior to move elements from the older, smaller data buffer to the newer, larger-capacity buffer. A constant-time move operation allows insertion at the front of the vector to become a linear-time operation, with no need to perform any extra operations regardless of the contents of the contained elements.

Running this program on a range of input values, all on the same host and compiler,[33] can show dramatic differences in performance for the same source code; see Table 3.

**Table 3: Runtime impact of move semantics**

rvalueref-table3

| | | `push_back` | | `insert` | |
|---|---|---|---|---|---|
| $k$ | $N = 2^k$ | C++03 | C++11 | C++03 | C++11 |
| 8 | 256 | 0.029s | 0.030s | 0.028s | 0.030s |
| 9 | 512 | 0.037s | 0.033s | 0.235s | 0.032s |
| 10 | 1,024 | 0.065s | 0.039s | 1.560s | 0.048s |
| 11 | 2,048 | 0.179s | 0.114s | 13.704s | 0.112s |
| 12 | 4,096 | 0.628s | 0.359s | 99.057s | 0.373s |
| 13 | 8,192 | 2.409s | 1.338s | 764.613s | 1.364s |
| 14 | 16,384 | 9.728s | 5.347s | 5,958.029s | 5.463s |
| 15 | 32,768 | 66.789s | 35.418s | 40,056.858s | 34.318s |
| 16 | 65,536 | core dump | 97.943s | core dump | 92.920 |

This dramatic improvement comes from the move semantics of `std::vector`'s move operations enabling a constant-time instead of linear-time move, combined with the language facilitating `std::vector` being able to generically take advantage of that functionality when its `value_type` supports it. While nothing about these algorithms is impossible in C++03, having generic types able to reliably express and take advantage of this kind of improvement was deemed worth the cost of dramatically changing the language.

need-rvalue-references?

**Why do we need *rvalue* references?** The ability to *move* the internal data structure of an allocating type from one object to another in C++11 — rather than always having to *copy* it, in the C++03 sense — can, under appropriate circumstances lead to profoundly superior performance characteristics, but see *Potential Pitfalls* — **??** on page **??**. **AUs: there is no Pitfalls section called "Over/Misuse (too much hype)"; what did you intend?**

[33] The numbers shown were generated by timing the program on a T480 Thinkpad laptop with GCC version 7.4.0, setting optimization to `-O2`, and using `-std=c++03` or `-std=c++11` appropriately.

Classic C++, i.e., C++98/03, did not provide a systematic syntactic means for indicating that it was OK to extract the *guts* of an object and transplant them into another object of like type. The only time that doing such a thing would have been guaranteed to be safe in C++03 was when the object was a *temporary*, but there was no way to overload the copy constructor or assignment operator so that they would behave differently (and more optimally) when passed a *temporary*.

Classic C++ did, however, provide one Standard Library type, `std::auto_ptr`, that attempted to implement move semantics. This ill-fated, smart pointer type had a non**`const`** *lvalue* reference copy constructor that would take ownership from and reset its source object when "copied." While `std::auto_ptr` functioned as a pioneer for move semantics in C++03, it also helped to identify the dangers of attempting to implement move semantics without the more fundamental changes that came with *rvalue* references. Many attempts to work with containers of `std::auto_ptr` or to leverage standard algorithms with `std::auto_ptr` showed how easy it was for generic code to assume that copies were safe to make and promptly destroy the data on which they were attempting to operate. Move semantics enabled the introduction of `std::unique_ptr` as a true move-only type without the pitfalls of `std::auto_ptr`, and at the same time, `std::auto_ptr` was deprecated[34] and then finally removed in C++17.[35]

The introduction of *rvalue* references, using a syntax of `&&` instead of the single `&` used for *lvalue* references, provided the key way in which implementations that take advantage of objects being in a movable state can be written in a safe and robust manner. `std::auto_ptr` showed that modifiable *lvalue* references were not sufficient for this task, and thus a new reference type was introduced to enable move operations. A new reference type facilitated new rules for what it could bind to and how it integrated with overload resolution and template argument deduction and provided a distinct format to identify implementations of move operations that would have minimal risk of changing the meaning of existing types.

a-new-value-category?

**Why do we need a new value category?** Simply put, the challenge for the designers of this C++11 feature was to enable move operations to occur when either (1) the compiler knows for certain that it is safe to do so or (2) the programmer takes responsibility for explicitly authorizing the compiler to enable a move operation that the compiler would not otherwise consider safe on its own. Their solution was two-fold: 1. Invent the notion of an *rvalue* reference as a means for binding more aggressively than a **`const`** *lvalue* reference to expressions that are eligible to be moved from during overload resolution. 2. Invent a new value category, known as an *xvalue*, that could distinguish when an expression identifies an object that is eligible to be moved from, including temporary objects that are about to be implicitly destroyed and will no longer be reachable as well as objects manually identified, through an explicit cast to *rvalue* reference, as being eligible.

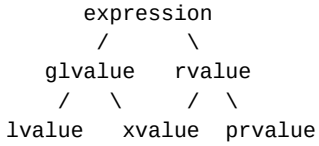
After a great deal of effort during the C++11 standardization process, the set of value categories that we have come to know today evolved and is represented in the Standard[36] with the following chart[37]:

[34] **?**

[35] **?**

[36] **[AUs: Where in the standard? cite.]**

[37] The motivation for what would become the terminology for the new value categories is explained by Bjarne Stroustrup himself (**?**).




This taxonomy helped greatly in formalizing the mechanics of move operations in the language, culminating in the final form of value categories in C++11.[38] A fundamental property that led to this categorization was identifying two primary, independent properties that could apply to a value.

1. A value can **have identity** or be **reachable** if it has an address and exists independently of the current expression.
2. A value can be **movable** if it is OK to cannibalize the internal representation of the object representing the value. Distinguishing values such as this was the primary goal of seeking to introduce move semantics into the language in the first place.

The two classical value categories, *lvalues* and non*lvalues*, both had opposite orientations for these two properties. An *lvalue* was reachable and not movable, whereas a non*lvalue* was not reachable and was movable. As the realization emerged that the goal was to treat these two properties orthogonally, it became apparent that the other pairings of these properties needed to be considered. A nonmovable, nonreachable value was deemed essentially not useful or worth considering, as nothing could fundamentally be done with such a value. A movable and reachable object, however, was the missing piece of the puzzle needed to manually identify that an object was ready to be moved from.

Given this understanding, the task then became to identify reasonable names for values with these sets of properties.

- The category of *lvalue* was already formalized in the Standard and clearly embodied values that were reachable and not movable.
- At a fundamental level, historical work in computer science had trained people to intuitively conclude that *lvalues* and *rvalues* were a partitioning of the set of all values; i.e., every value was either an *lvalue* or an *rvalue*, and no value was both. This led to the *rvalue* category implicitly becoming all movable values, both reachable and not reachable.
- The set of values that are movable and not reachable matched well with many of the classical notions of *rvalue*, as it included *pure* values that had no object representation yet, such as integer literals and enumerators. Thus, this category was called prvalues, or *pure rvalues*.
- The set of all values that are reachable was also something that would need a name, as this category would come into play in the language in a number of places as a generalization of what was previously workable only with *lvalues*. Many fundamental operations that apply to an object in memory are worded in terms of the new *glvalue* category, or *generalized lvalues*.

[38] ?

- Finally, the category of movable objects that were reachable needed a name. As this was a functionally new invention, a suitable name was not evident, and, perhaps serendipitously, the noncommittal letter "x" was chosen for the new *xvalue* category. Originally, this was chosen to represent "the unknown, the strange, the xpert only, or even the x-rated."[39] Over time, the "x" evolved to capture the movability of the values in this category, and it now stands for "expiring."

It is worth noting that *rvalue* changed meaning in C++11 by expanding to include *xvalues* that are reachable. This coincides well with *rvalue* references being able to functionally bind to any *rvalue* but was a shift from classical C++ where *rvalues* were never reachable. This change was deemed worth the potential confusion as it had much less impact than a similar change to *lvalue* would have had, and it kept *lvalue* and *rvalue* as a disjoint partition of the set of all values.

In C++11 as originally specified, the only way to arrive at an *xvalue* was via an explicit cast to an *rvalue* reference or by calling a function that returns an *rvalue* reference; hence, *xvalues* referencing unreachable temporaries were significantly harder to acquire. The original understanding many had was that an *xvalue* was essentially a *movable* nontemporary. Even as originally formulated, though, member functions invoked on temporaries were able to access and distribute *lvalue*, and thus *xvalue*, references to the temporaries through the **`this`** pointer. Two major core issues that impacted *xvalues*[40,41] were addressed in C++14 and applied retroactively as defect reports to C++11. These rule changes led to the treatment of subobjects of both *prvalues* and *xvalues* as *xvalues*, such as when doing data member access, pointer to data member dereferencing, and array subscripting.

As the understanding of where the differing value categories can have an impact has evolved, it has become clear that *xvalue* expressions identify movable objects that exist in memory, i.e., their lifetime has begun, but places no restrictions on whether the objects in question will continue their lifetimes beyond the end of the current statement.[42] In normal use, all *rvalues* — both *xvalues* and *prvalues* — bind equally well to *rvalue* references and are, for the most part, indistinguishable programmatically. One of the few places where they are treated differently is as a parenthesized expression used with the **`decltype`** operator (see Section 1.1."**??**" on page **??**), which can be leveraged to identify value categories (see *Use Cases — Identifying value categories* on page 61).

With this new set of value categories in hand, the only remaining pieces needed to integrate move semantics into the language were mechanisms to produce *xvalues* in code to which *rvalue* references may be bound.Given a *prvalue*, this mechanism could obviously be an implicit conversion, and thus many automatic benefits of move semantics could be enabled." Supporting only such implicit conversions and moving from only temporaries, however, was

[39] **?**

[40] CWG issue 616; **?**

[41] CWG issue 1213; **?**

[42] This same refinement of what is an *xvalue* and what is a *prvalue* inspired the formalization of guaranteed copy elision in C++17. The cases where no extra temporary needs to be created from the return value from a function are exactly the cases where a function call expression is a *prvalue* of the appropriate type, enabling the choice to simply materialize that *prvalue* in the location of the variable being initialized. This restructuring avoids the need to define intermediate temporaries and renders unnecessary the ability to optimize away those temporaries, making object lifetime more deterministic as well as enabling RVO for types that can neither be copied nor moved.

not going to solve strong motivational cases, like our `std::vector<std::vector<std::string>>` example described above; see *Rationale: Why do we want move semantics?* on page 93. [**AUs: where is the example? Do you mean it is in the Rationale section? If so, just say that. "Above" is too vague to be really helpful.**] To enable such cases, *xvalues* also needed to include *lvalues* that had been explicitly cast to *rvalue* references, enabling the moving of objects already stored in a data structure. To further enable the use of *rvalue* references, functions that return an *rvalue* reference also create *xvalues* when invoked, enabling Standard Library functions like `std::move` and other cases where encapsulating the ability to enable moving from a preexisting object is desired.

sec-conditional-cpp14

# Chapter 3

## Unsafe Features

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

# Chapter 3 Unsafe Features

sec-unsafe-cpp14