EMBRACING MODERN C++ SAFELY

JOHN LAKOS | VITTORIO ROMEO | ROSTISLAV KHLEBNIKOV | ALISDAIR MEREDITH

Embracing Modern C++ *Safely*

Use Cases

Perfectly forwarding an expression to a downstream consumer

A frequent use of forwarding references and `std::forward` is to propagate an object, whose value category is invocation-dependent, down to one or more service providers that will behave differently depending on the value category of the original argument.

As an example, consider an overload set for a function, `sink`, that accepts an `std::string` either by **`const`** lvalue reference, e.g., with the intention of *copying* from it, or by rvalue reference, e.g., with the intention of *moving* from it:

```
void sink(const std::string& s) { target = s; }
void sink(std::string&& s)      { target = std::move(s); }
```

Now, let's assume that we want to create an intermediary function template, `pipe`, that will accept an `std::string` of any value category and will dispatch its argument to the corresponding overload of `sink`. By accepting a forwarding reference as a function parameter and invoking `std::forward` as part of `pipe`'s body, we can achieve our original goal without any code duplication:

```
template <typename T>
void pipe(T&& x)
{
    sink(std::forward<T>(x));
}
```

Invoking `pipe` with an *lvalue* will result in `x` being an lvalue reference and thus `sink(const std::string&)` being called. Otherwise, `x` will be an rvalue reference and `sink(std::string&&)` will be called. This idea of enabling *move* operations without code duplication, as `pipe` does, is commonly referred to as **perfect forwarding**; see *Perfect forwarding for generic factory functions* on page 388.

Handling multiple parameters concisely

Suppose we have a **value-semantic type (VST)** that holds a collection of attributes where some (not necessarily proper) subset of them need to be changed together to preserve some class invariant[2]:

```
#include <type_traits>  // std::decay, std::enable_if, std::is_same
#include <utility>      // std::forward

struct Person { /* UDT that benefits from move semantics */ };

class StudyGroup
```

[2]This type of value-semantic type can be classified more specifically as a *complex-constrained* attribute class; a discussion of this topic is planned for **lakos2a**, section 4.2.

```
{
    Person d_a;
    Person d_b;
    Person d_c;
    Person d_d;
    // ...

public:
    static bool isValid(const Person& a, const Person& b,
                        const Person& c, const Person& d);
        // Return true if these specific people form a valid study group under
        // the guidelines of the study-group commission, and false otherwise.
    // ...

    template <typename PA, typename PB, typename PC, typename PD,
        typename = typename std::enable_if<
            std::is_same<typename std::decay<PA>::type, Person>::value &&
            std::is_same<typename std::decay<PB>::type, Person>::value &&
            std::is_same<typename std::decay<PC>::type, Person>::value &&
            std::is_same<typename std::decay<PD>::type, Person>::value>::type>
    int setPersonsIfValid(PA&& a, PB&& b, PC&& c, PD&& d)
    {
        enum { e_SUCCESS = 0, e_FAIL };

        if (!isValid(a, b, c, d))
        {
            return e_FAIL;  // no change
        }

        // Move or copy each person into this object's Person data members.

        d_a = std::forward<PA>(a);
        d_b = std::forward<PB>(b);
        d_c = std::forward<PC>(c);
        d_d = std::forward<PD>(d);

        return e_SUCCESS;  // Study group was updated successfully.
    }
};
```

Because the template arguments used in each successive function parameter are deduced interdependently from the types of their corresponding function arguments, the `setPersonsIfValid` function template can be instantiated for a full Cartesian product of variations of qualifiers that can be on a `Person` object. Any combination of *lvalue* and *rvalue* `Person`s can be passed, and a template will be instantiated that will copy the *lvalues* and move from the *rvalues*. To make sure the `Person` objects are created externally, the function is restricted, using `std::enable_if`, to instantiate only for types that **decay** to `Person`, i.e.,

types that are **cv-qualified** or ref-qualified `Person`. Because each parameter is a forwarding reference, they can all implicitly convert to **`const Person&`** to pass to `isValid`, creating no additional **temporaries**. Finally, `std::forward` is then used to do the actual moving or copying as appropriate to **data members**.

Perfect forwarding for generic factory functions

Consider the prototypical standard-library generic **factory function**, `std::make_shared<T>`. On the surface, the requirements for this function are fairly simple: Allocate a place for a `T` and then construct it with the same arguments that were passed to `make_shared`. Correctly passing arguments to the constructor, however, gets reasonably complex to implement efficiently when `T` can have a wide variety of ways in which it might be initialized.

For simplicity, we will show how a two-argument `my::make_shared` might be **defined**, knowing that a full implementation would employ variadic template arguments for this purpose; see Section 2.1."Variadic Templates" on page 873. Furthermore, our simplified `make_shared` creates the object on the heap with **`new`** and constructs an `std::shared_ptr` to manage the lifetime of that object.

Let's now consider how we would structure the declaration of this form of `make_shared`:

```
namespace my {
template <typename OBJECT_TYPE, typename ARG1, typename ARG2>
std::shared_ptr<OBJECT_TYPE> make_shared(ARG1&& arg1, ARG2&& arg2);
}
```

Notice that we have two forwarding reference arguments, `arg1` and `arg2`, with deduced types `ARG1` and `ARG2`. Now, the body of our function needs to carefully construct our `OBJECT_TYPE` object on the heap and then create our output `shared_ptr`:

```
template <typename OBJECT_TYPE, typename ARG1, typename ARG2>
std::shared_ptr<OBJECT_TYPE> my::make_shared(ARG1&& arg1, ARG2&& arg2)
{
    OBJECT_TYPE* object_p = new OBJECT_TYPE(std::forward<ARG1>(arg1),
                                            std::forward<ARG2>(arg2));
    try
    {
        return std::shared_ptr<OBJECT_TYPE>(object_p);
    }
    catch (...)
    {
        delete object_p;
        throw;
    }
}
```

Notice that this simplified implementation needs to clean up the allocated object if the constructor for the return value throws; normally an RAII **proctor** to manage this ownership would be a more robust solution to this problem.

Importantly, using `std::forward` to construct the object means that the arguments passed to `make_shared` will be used to find the appropriate matching two-parameter constructor of `OBJECT_TYPE`. When those arguments are *rvalues*, the constructor found will again search for one that takes an *rvalue*, and the arguments will be moved from. What's more, because this function wants to forward exactly the **`const`**ness and reference type of the input arguments, we would have to write 12 distinct overloads, one for each argument, if we were not using perfect forwarding — the full Cartesian product of **`const`** (or not), **`volatile`** (or not), and `&` or `&&` (or neither). A full implementation of just this two-parameter variation would require 144 distinct overloads, all almost identical and most never used. Using forwarding references reduces that to just one overload for each number of arguments.

Wrapping initialization in a generic factory function

Occasionally we might want to initialize an object with an intervening function call wrapping the actual construction of that object. Suppose we have a tracking system that we want to use to monitor how many times certain initializers have been invoked:

```
struct TrackingSystem
{
    template <typename T>
    static void trackInitialization(int numArgs);
        // Track the creation of a T with a constructor taking numArgs
        // arguments.
};
```

Now we want to write a general utility function that can be used to construct an arbitrary object and notify the tracking system of the construction for us. Here we will use a variadic pack (see Section 2.1."Variadic Templates" on page 873) of forwarding references to handle calling the constructor for us:

```
template <typename OBJECT_TYPE, typename... ARGS>
OBJECT_TYPE trackConstruction(ARGS&&... args)
{
    TrackingSystem::trackInitialization<OBJECT_TYPE>(sizeof...(args));
    return OBJECT_TYPE(std::forward<ARGS>(args)...);
}
```

This use of a variadic pack of forwarding references lets us add tracking easily to convert any initialization to a tracked one by inserting a call to this function around the constructor arguments:

```
void myFunction()
{
    BigObject untracked("Hello", "World");
    BigObject tracked = trackConstruction<BigObject>("Hello","World");
}
```

On the surface there does seem to be a difference between how objects `untracked` and `tracked` are constructed. The first variable is having its constructor directly invoked, while the second is being constructed from an object being returned by-value from `trackConstruction`. This construction, however, has long been something that has been optimized away to avoid any additional objects and construct the object in question just once. In this case, because the object being returned is initialized by the **`return` statement** of `trackConstruction`, the optimization is called **return value optimization** (RVO). C++ has always allowed this optimization by enabling **copy elision**. Ensuring that this elision actually happens (on all current compilers of which the authors are aware) is possible by publicly **declaring** but not defining the **copy constructor** for `BigObject`.[3] We find that this code will still compile and link with such an object, providing observable proof that the copy constructor is never actually invoked with this pattern.

Emplacement

Prior to C++11, inserting an object into a Standard Library container always required the programmer to first create such an object and then copy it inside the container's storage. As an example, consider inserting a temporary `std::string` object in an `std::vector<std::string>`:

```
void f(std::vector<std::string>& v)
{
    v.push_back(std::string("hello world"));
        // invokes std::string::string(const char*) and the copy constructor
}
```

In the function above, a temporary `std::string` object is created on the **stack frame** of `f` and is then copied to the dynamically allocated buffer managed by `v`. Additionally, the buffer might have insufficient capacity and hence might require reallocation, which would in turn require every element of `v` to be copied from the old buffer to the new, larger one.

In C++11, the situation is significantly better thanks to rvalue references. The temporary will be moved into `v`, and any subsequent buffer reallocation will *move* the elements between buffers rather than copy them, assuming that the element's move constructor has a **`noexcept`** specifier (see Section 3.1."**`noexcept`** Specifier" on page 1085). The amount of work can, however, be further reduced: What if, instead of first creating an object externally, we constructed the new `std::string` object directly in `v`'s buffer?

This is where **emplacement** comes into play. All standard library containers, including `std::vector`, now provide an emplacement API powered by variadic templates (see Section 2.1."Variadic Templates" on page 873) and perfect forwarding (see *Perfect forwarding for generic factory functions* on page 388). Rather than accepting a **fully-constructed** element, emplacement operations accept an arbitrary number of arguments, which will in

[3] In C++17, this copy elision can be guaranteed and is allowed to be done for objects that have no copy or **move constructors**.

turn be used to construct a new element directly in the container's storage, thereby avoiding unnecessary copies or even moves:

```
void g(std::vector<std::string>& v)
{
    v.emplace_back("hello world");
        // invokes only the std::string::string(const char*) constructor
}
```

Calling `std::vector<std::string>::emplace_back` with a **`const char*` argument** results in a new `std::string` object being created **in place** in the next empty spot of the vector's storage. Internally, `std::allocator_traits::construct` is invoked, which typically employs **placement `new`** to construct the object in raw dynamically allocated memory. As previously mentioned, `emplace_back` makes use of both variadic templates and forwarding references; it accepts any number of forwarding references and internally *perfectly forwards* them to the constructor of `T` via `std::forward`:

```
template <typename T>
template <typename... Args>
void std::vector<T>::emplace_back(Args&&... args)
{
    // ...
    (void) new (d_data_p[d_size]) T(std::forward<Args>(args)...);  // pseudocode
    // ...
}
```

Emplacement operations remove the need for copy or move operations when inserting elements into containers, potentially increasing the performance of a program and sometimes, depending on the container, even allowing even noncopyable or nonmovable objects to be stored in a container.

As previously mentioned, declaring without defining the copy or move constructor of a noncopyable or nonmovable type to be private is often a way to guarantee that a C++11/14 compiler constructs an object in place. Containers that might need to move elements around for other operations, such as `std::vector` or `std::deque`, will still need **movable** elements, while node-based containers that never move the elements themselves after initial construction, such as `std::list` or `std::map`, can use emplace along with noncopyable or nonmovable objects.

Decomposing complex expressions

Many modern C++ libraries have adopted a more functional style of programming, chaining the output of one function as the arguments of another function to produce complex expressions that accomplish a great deal in relatively concise fashion. Consider a function that reads a file, does some spell-checking for every unique word in the file, and gives us a