POINTERS, REFERENCES, AND MORE

- **Pointers, References and Optional References in C++**
- **The Pitfalls of Aliasing Pointers in Modern C++**
- **auto + const + smart pointer = bad mix?**
- **Understanding lvalues, rvalues and their references**
- **How to Construct C++ Objects Without Making Copies**

Fluent {C++}

Pointers, References and Optional References in C++

In C++, one can manipulate objects directly or via something else, which is commonly called a **handle**. At the beginning of C++, handles could be pointers, references and iterators. Modern C++ brought in reference wrappers, and boost introduced optional references.

The fact that a given piece of code chooses to use one particular handle **expresses something**. For this reason it is important to know the meaning of each handle, so that you can leverage on them while you read and write code.

Before getting into the specificities of each type of handle, let's make a brief…

Point about naming

Here is the naming guideline that I recommend for handles:

The name of a handle must be what the name of the object would have been **if we held it directly**. For example, pointers names shouldn't start with "`p`", and iterators names shouldn't start with "`it`".

Indeed there is no need to clutter a name with such extra information: it's already in its type, if we really want to know. And when reading code, we mostly don't want to know anyway.

Following this guideline is in fact the natural thing to do. Handles are just thingies that help manipulate another object. Very much like the handle of a piece of luggage in fact. To illustrate, consider the following two versions of the same story:

Version 1:

Developer: “Shall we leave for the airport now?”
Spouse: “Sure, let’s go!”
Developer: “Ok, let me just grab my suitcase and I’m ready to go!”

Version 2:

Developer: “Shall we leave for the airport now?”
Spouse: “Sure, let’s go!”
Developer: “Ok, let me just grab *the handle of* my suitcase and I’m ready to go!”
Spouse: “You’re so weird.”

Even though it is true and accurate that a suitcase is manipulated with a handle, you don’t want this detail to show in its denomination. The same goes for code.

The good ol' pointers and references

I learned a significant part of this section from the opening item of Scott Meyer's More Effective C++.

Nullability

A pointer can point to nothing. A reference can't (*).

A way to express a pointer pointing to nothing before C++11 is to make it equal to zero:

```
T* pointer = 0;
```

C++11 introduces `nullptr`, making it more explicit:

```
T* pointer = nullptr;
```

This also helps static analysers better understand the code.

(*) *A reference can, technically, be null:*

```
T* pointer = nullptr;
T& reference = *pointer;
```

This seems dumb, but if the reference and the pointer are several stack layers away from each other, it's harder to spot. Anyway, the convention for references is that they should never be null.

Rebinding

We can make a pointer point to something different in the course of its life. A reference points to the same object during all its lifetime.

To rebind a pointer:

```
T object1;
T object2;
```

```
T* pointer = &object1; // pointer points to object1
pointer = &object2; // pointer points to object2
```

The same syntax transposed to references makes an assignment on `object1`:

```
T object1;
T object2;

T& reference = object1; // reference points to object1
reference = object2; // equivalent to: object1 = object2
```

Should I use a pointer or a reference?

Pointers are more powerful than references in the sense that they allow two things that references don't: nullability and rebinding. And as you know, a great power comes with great responsibilities: you need to worry about a pointer not being null, and to follow its life to check for target changes.

For this reason, **unless you need the additional functionalities of pointers, you should use references.**

Other differences

Pointers and references have a different syntax: pointers access the pointed object with `*` or `->`, and references have the same syntax as direct access to the object.

Finally, a failed `dynamic_cast` doesn't have the same effect on a pointer and reference:

- a failed `dynamic_cast` on a pointer returns a **null pointer**,
- a failed `dynamic_cast` on a reference **throws an exception** of type `std::bad_cast`. Which makes sense because it can't return a null reference.

std::reference_wrapper

The fact that references can't rebind makes them unfriendly with `operator=`. Consider the following class:

```
class MyClass
{
```

```
public:
    MyClass& operator=(MyClass const& other)
    {
        ???
    }
    // ...
private:
    T& reference;
};
```

What should `operator=` do? The natural thing would be to make `reference` point to the same object as `other.reference` does, but references can't rebind. For this reason, the compiler gives up and doesn't implement a default assignment operator in this case.

`std::reference_wrapper`, from the `<functional>` header, provides a way out of this, by wrapping a reference into a assignable (and copyable) object. It comes with the `std::ref` helper to avoid typing template parameters:

```
T object1;
auto reference = std::ref(object1); // reference is of type
std::reference_wrapper<T>
```

Its `operator=` does the natural thing, rebinding:

```
T object1;
auto reference = std::ref(object1); // reference points to object1
T object2;
reference = std::ref(object2); // reference now points to object2
// object 1 hasn't changed
```

Replacing `T&` with `std::reference_wrapper<T>` in `MyClass` solves the problem of `operator=`, because the compiler can then implement it by just calling `operator=` on the `std::reference_wrapper<T>`.

Note that we can assume that `std::refrence_wrapper` **always points to something**, since it wraps a reference that is supposed to point to something.

If you wonder how it works, `std::reference_wrapper` can be implemented with a pointer to the object pointed by the reference it is passed.

Optional References

Optional objects were first introduced in boost. An `optional<T>` represents an object of type T, but that can be "null", "empty" or "not set" as you will.

In the case where T is a reference `boost::optional<T>` has interesting semantics:

- when the optional is not null **it points to something**, like a normal reference,
- **it can point to nothing**, by being a null optional (an optional can be `nullopt`)
- **it can rebind** through its `operator=`, like `std::reference_wrapper`.

And this looks exactly like… the features of a pointer!

What differentiates the very modern-looking optional reference from our old-fashioned pointer then?

The answer is the low-level aspects of pointers. Like pointer arithmetics, array semantics, and the fact that a pointer can be used to model a memory address.

For this reason, optional references better model a **handle** than a pointer does.

However, since the C++ standard committee members weren't all convinced that assignment on optional references should do rebinding, **optional references didn't make it into C++17**. Maybe the committee will reconsider them for a future version of the language though.

A practical consequence of this is that if you're using boost optional references now, your code won't integrate seamlessly with `std::optional` when you upgrade to C++17. This constitutes a drawback to optional references, even if it isn't coming from an intrinsic problem.

Pack up and go

In summary,

- **References** cannot be null and cannot rebind,

- **`std::reference_wrapper`** cannot be null but can rebind,
- **Pointers** can be null and can rebind (and can do low-level address manipulations),
- **boost optional references** can be null and can rebind (but are incompatible with `std::optional`).

As you see, there are multiple handles that can hold a suitcase. You just need to pick the one that fits your need and nothing more, and off you go.

The Pitfalls of Aliasing Pointers in Modern C++

This is guest post written by a guest author Benjamin Bourdin. If you're also interested to share your ideas on Fluent C++, check out our guest posting area.

With the advent of smart pointers in Modern C++, we see less and less of the low-level concerns of memory management in our business code. And for the better.

To go further in this direction, we could be tempted to make the names of smart pointers themselves disappear: `unique_ptr`, `shared_ptr`... Maybe you don't want to know those details, and only care about that an object is a "pointer that deals with memory management", rather that the exact type of pointer it is:

```
using MyClassPtr = std::unique_ptr<MyClass>;
```

I've seen that sort of code at multiple occasions, and maybe you have this in your codebase too. But there are several issues with this practice, that make it *not* such a good idea. The following presents the argument against aliasing pointer types, and if you have an opinion we'd be glad to hear it in the comments section!

Smart pointers

Let's make a quick recap on smart pointers. The point here is not to enumerate all the types of smart pointers C++ has, but rather to refresh to your memory on the basic usages of smart pointers that will have issues when using an alias. If youre memory is already fresh on smart pointers, you can safely skip to the next section.

std::unique_ptr

`std::unique_ptr` is probably the most commonly used smart pointer. It represents the unique owner of a memory resource. The (C++14) standard way to create a `std::unique_ptr` is to use `std::make_unique`:

```
std::unique_ptr<MyClass> ptr = std::make_unique<MyClass>(0, "hi");
```

`std::make_unique` performs a perfect forwarding of its parameters to the constructor of `MyClass`. `std::unique_ptr` also accepts raw pointers, but that's not the recommended practice:

```
std::unique_ptr<MyClass> ptr(new MyClass(0, "hi"));
```

Indeed, in certain cases it can lead to memory leaks, and one of the goals of smart pointers is to get rid of `new` and `delete` in business code.

Functions (or, more frequently, class methods) can acquire the ownership of the memory resource of a `std::unique_ptr`. To do this, they take a `std::unique_ptr` by value:

```
void fct_unique_ptr(std::unique_ptr<MyClass> ptr);
```

To pass arguments to this function, we have to invoke the move constructor of `std::unique_ptr` and therefore pass it an rvalue, because `std::unique_ptr` doesn't have a copy constructor. The idea is that the move constructor transfers the ownership from the object moved-from to the object moved-to.

We can invoke it this way:

```
std::unique_ptr<MyClass> ptr = std::make_unique<MyClass>(0, "hi");
fct_unique_ptr(std::move(ptr)); // 1st way
fct_unique_ptr(std::make_unique<MyClass>(0, "hi")); // 2nd way
fct_unique_ptr(std::unique_ptr<MyClass>(new MyClass(0, "hi"))); //
3rd way (compiles, but not recommended to use new)
```

std::shared_ptr

A `std::shared_ptr` is a pointer that can share the ownership of a memory resource with other `std::shared_ptr`s.

The (C++11) standard way to create `std::shared_ptr`s is by using `std::make_shared`:

```
std::shared_ptr<MyClass> ptr = std::make_shared<MyClass>(0, "hi");
```

Like `std::make_unique`, `std::make_shared` perfect forwards its arguments to the constructor of `MyClass`. And like `std::unique_ptr`, `std::shared_ptr` can be built from a raw pointer, and that is not recommended either.

Another reason to use `std::make_shared` is that it can be more efficient than building a `std::shared_ptr` from a raw pointer. Indeed, a shared pointer has a reference counter, and with `std::make_shared` it can be constructed with the `MyClass` object all in one heap allocation, whereas creating the raw pointer and then the `std::shared_ptr` requires two heap allocations.

To share the ownership of a resource with a function (or, more likely, a class method), we pass a `std::shared_ptr` by value:

```
void fct_shared_ptr(std::shared_ptr<MyClass> ptr);
```

But contrary to `std::unique_ptr`, `std::shared_ptr` accepts **lvalues**, and the copy constructor then creates a additional `std::shared_ptr` that refers to the memory resource:

```
std::shared_ptr<MyClass> ptr = std::make_shared<MyClass>(0, "hi");
fct_shared_ptr(ptr);
```

Passing rvalues would not make sense in this case.

Alias to pointer: danger!

Back to the question of aliasing pointer types, are the following aliases good practice?

```
using MyClassPtr = std::unique_ptr<MyClass>;
```

or

```
using MyClassPtr = std::shared_ptr<MyClass>;
```

Throughout the above examples, we've seen **different semantics and usages** for the various smart pointers. As a result, hiding the type of the smart pointers behind an alias leads to issues.

What sort of issues? The first one is that we lose the information about ownership. To illustrate, consider the following function:

```
void do_something(MyClassPtr handler);
```

As a *reader* of the function, I don't know what this call means: is it a transfer of ownerhip? Is it a sharing of ownership? Is it simply passing an pointer to access its underlying resource?

As the *maintainer* of the function, I don't know what exactly I'm allowed to do with that pointer: can I safely store the pointer in a object? As its name suggests, is `MyClassPtr` a simple raw pointer, or is it a smart pointer? I have to go look at what is behind the alias, which reduces the interest of having an alias.

And as a *user* of the function, I don't know what to pass to it. If I have a `std::unique_ptr<MyClass>`, can I pass it to the function? And what if I have a `std::shared_ptr<MyClass>`? And even if I have a `MyClassPtr`, of the same type of the parameter of `do_something`, should I *copy* it or *move* it when passing it to `do_something`? And to instantiate a `MyClassPtr`, should we use `std::make_unique`? `std::make_shared`? `new`?

A too high level of abstraction

In all the above situations (maintenance, function calls, instantiations), using an alias can force us to go look what it refers to, making the alias a problem rather than a help. It's a bit like a function whose name would not be enough to understand it, and that would require you to go look at its implementation to understand what it does.

The intention behind aliasing a smart pointer is a noble one though: raising its level of abstraction, by hiding lower-level details related to resources life cycle. The problem here is that those "lower-level" details are in fact at the same level of abstraction as the code using those smart pointers. Therefore the alias is too high in terms of levels of abstraction.

Another way to see it is that, in general, making an alias allows to some degree to change the type it refers to without going over all of its usages and changing them (a bit like `auto` does).

But as we've seen in this article, changing the type of pointer, from raw pointer to `std::unique_ptr` or from `std::unique_ptr` to `std::shared_ptr` for example, changes the semantics of the pointers and requires to modify many of their usages anyway.

What is your opinion on this? Are you in favor or against aliasing pointer types? Why?

auto + const + smart pointer = bad mix?

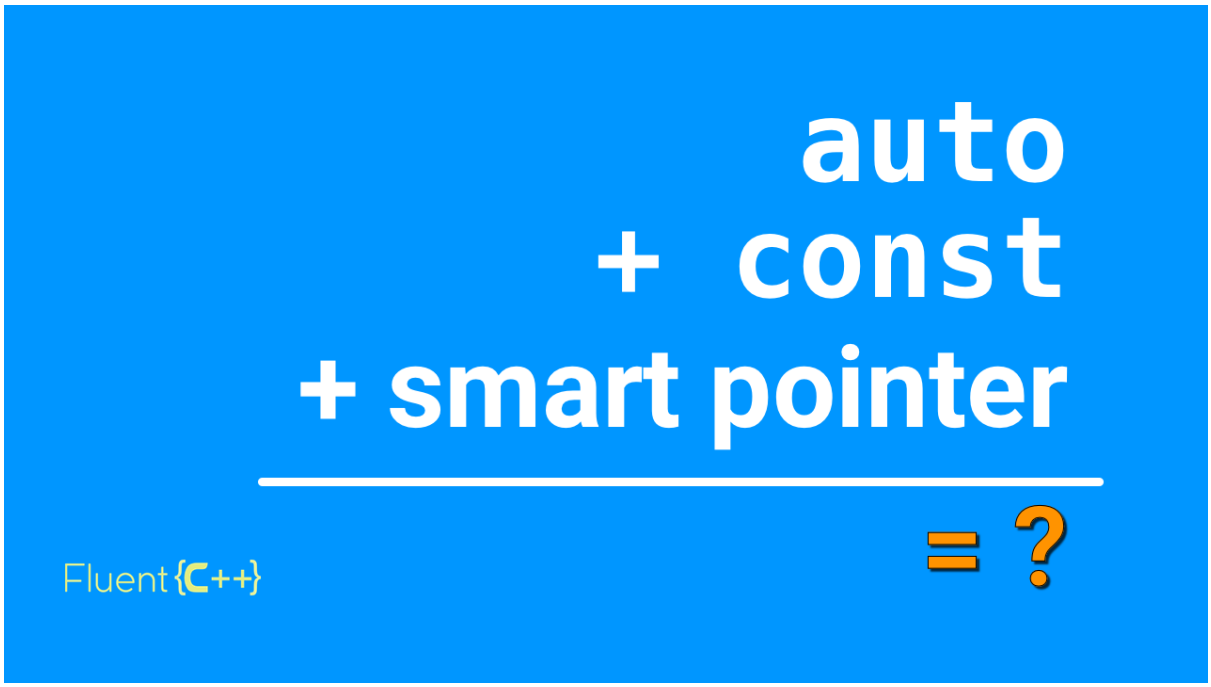
`const` is a feature that has been appreciated by C++ developers for decades of good services, to make code more robust by preventing accidental modifications.

Smart pointers have been around for a long time too, and simplified the life cycle of many objects along with the life balance of many developers across the years.

`auto` is a more recent feature (C++11), designed to make code simpler, and it has been promoted for years for us to use it almost always.

So when we put `auto`, `const` and a smart pointer together, we should expect it to produce a great mix of simple, robust and expressive code.

But this combination can lead to deceptive code rather than expressive code. As in code that looks like it does something, but it fact doesn't. And deceptive code is one of the most dangerous sorts of code.



auto + const + pointer

When declaring an object, using `auto` and `const` implies that the object is indeed `const`:

```
auto const numbers = std::vector<int>{1, 2, 3, 4, 5};
```

The above vector `numbers` is `const`: we can't add, remove or modify anything to it otherwise the code wouldn't compile. If this vector is meant to be an input, it prevents us to modify it by accident and create a bug.

Now consider the following case: assigning a **pointer** into an `auto const` value:

```
Thing* getSomething();

auto const thing = getSomething();
```

What does this code look like? It says that `thing` is `const`. But `thing` is a pointer, which means that `thing` cannot point to anything else than whatever `getSomething` has returned. This is the equivalent of:

```
Thing* const thing = getSomething();
```

The pointer is const, but not the value it points to.

But when using `thing` in business code, do you really care about the value of the pointer? If the point of using `thing` is to reach to the object it points to, like it's often the case, you don't. The role of `thing` is to embody the object is points to, and it so happens that you're given a pointer to manipulate it.

Therefore, what it looks like to me is that the code suggests that we're manipulating a `const Thing`, and not a `const` pointer to `Thing`. True, this is not what is happening, but when reading code you don't check out every prototype of every function that is called. All the more so if the prototype of `getSomething` is not in the immediate vicinity (which it generally isn't):

```
auto const thing = getSomething();
```

This code screams that you are protected by a read-only `thing`, whereas it's just a read-only pointer to a modifiable object. Doesn't it look deceptive to you?

One way to get around this problem could be to use `auto const*`, to make the pointed-to object `const`:

```
auto const* thing = getSomething();
```

Or is it a case for the Hungarian notation to come back?

```
auto const pThing = getSomething();
```

Ew, no, we don't like the Hungarian notation.

But you may be thinking, who returns a raw pointer from a function anyway? We even evoked the possibility of removing raw pointers from the C++ (okay, it was on the 1st of April but still, the idea didn't come out of nowhere). We should use smart pointers now, right?

Right, we should. But first, there is still legacy code out there that hasn't caught up yet, and it's safe to say that there will still be some for a while.

And second, smart pointers suffer from the same problem, but worse. Let's see why.

auto + const + smart pointer

Let's modernize the interface of `getSomething` and make it return a smart pointer to express that it relinquishes the ownership of the objet to its caller:

```
std::unique_ptr<Thing> getSomething();
```

Our calling code looks like this:

```
auto const thing = getSomething();
```

Even if in terms of ownership the code is much more robust, in terms of what is `const` and what is not, the situation is identical to the one with raw pointers.

Indeed, in the above code the smart pointer is `const`, which we rarely care about, but the object it points to is not. And the code gives that false feeling of protection by luring a reader passing by into thinking that the object really used by the code (likely the `Thing` the smart pointer points to) is `const` and that all is safe.

What is worse with smart pointers is that there is no way to add info around the `auto`. With a raw pointer we could resort to:

```
auto const* thing = getSomething();
```

But with a smart pointer, we can't.

So in this case, I guess the best option is to remove the `const` altogether, to avoid any confusion:

```
std::unique_ptr<Thing> getSomething();
auto thing = getSomething();
```

Have you encountered this problem in your code? How did you go about it? All your comments are welcome.

Understanding lvalues, rvalues and their references

Even though rvalue references have been around since C++11, I'm regularly asked questions about how they work and how to use them. For this reason I'm going to explain my understanding of them here.

I think this is relevant to the topic of Fluent C++, expressive code in C++, because not understanding them adds a layer of confusion over a piece of code that tries to tell you its meaning.

Why am I writing this here? Indeed, you can read about rvalue references in C++ reference books and even on other blog posts on the Internet, and my purpose is not to duplicate them.

Rather, I will explain **what helped me understand them**. Indeed, I used to be very confused about them at the beginning, and this is because I was missing just a couple of **key pieces of information**. In particular one that I detail in the third section of this post.

If you find yourself confused about lvalues, rvalues and their references, this article is for you. And if you master them already, I hope you'll be kind enough to ring the bell if by chance you were to spot any meestayck.

About that, I'm very grateful to Stephan T. Lavavej for taking the time (once again!) to signal the errors that he saw in the post.

What is an lvalue and what is an rvalue?

In C++, every **expression** is either an lvalue or an rvalue:

- **an lvalue denotes an object whose resource cannot be reused,** which includes most objects that we can think of in code**.** Lvalues include expressions that designate

objects directly by their **names** (as in `int y = f(x)`, `x` and `y` are object names and are lvalues), but not only. For instance, the expression `myVector[0]` also is an lvalue.

- **an rvalue denotes an object whose resource can be reused**, that is to say a disposable object. This typically includes **temporary objects** as they can't be manipulated at the place they are created and are soon to be destroyed. In the expression `g(MyClass())` for instance, `MyClass()` designates a temporary object that `g` can modify without impacting the code surrounding the expression.

Now an **lvalue reference** is a reference that **binds to an lvalue**. lvalue references are marked with one ampersand (&).
And an **rvalue reference** is a reference that **binds to an rvalue**. rvalue references are marked with two ampersands (&&).

Note that there is one exception: there can be lvalue **const** reference binding to an rvalue. Anyway, let's not worry about this case just now, let's focus on the big picture first.

What is this all for?

rvalue references add the possibility to express a new intention in code: **disposable objects**. When someone passes it over to you (as a reference), it means **they no longer care about it**.

For instance, consider the rvalue reference that this function takes:

```
void f(MyClass&& x)
{
    ...
}
```

The message of this code to `f` is this: "The object that `x` binds to is YOURS. Do whatever you like with it, no one will care anyway." It's a bit like giving a copy to `f`... but without making a copy.

This can be interesting for two purposes: **improving performance** (see move constructors below) and taking over **ownership** (since the object the reference binds to has been abandoned by the caller – as in `std::unique_ptr`)

Note that this could not be achieved with lvalue references. For example this function:

```
void f(MyClass& x)
{
    ...
}
```

can modify the value of the object that `x` binds to, but since it is an lvalue reference, it means that somebody probably cares about it at call site.

I mentioned that lvalue **const** references could bind to rvalues:

```
void f(MyClass const& x)
{
    ...
}
```

but they are const, so even though they can bind to a temporary unnamed object that no one cares about, `f` can't modify it.

THE one thing that made it all click for me

Okay, there is one thing that sounds extra weird, but that makes sense given the definitions above: **there can be rvalue references that are themselves lvalues**.

One more time: **there can be rvalue references that are themselves lvalues**.

Indeed, a reference is defined in a **certain context**. Even though the object it refers to may be disposable in the context it has been created, it may not be the case in the context of the reference.

Let's see this in an example. Consider `x` in the following code:

```
void f(MyClass&& x)
{
    ...
}
```

Within `f`, the expression "`x`" is an lvalue, since it designates the name of an object. And indeed, if some code inside of `f` modifies `x`, the remaining code of `f` will certainly notice. In the context of `f`, `x` is not a disposable object.

But `x` refers to an object that is disposable **in the context that called `f`**. In that sense, it refers to a disposable object. This is why its type has `&&` and is a **rvalue reference**.

Here is a possible call site for `f`:

```
f(MyClass());
```

The rvalue expression `MyClass()` denotes a temporary, disposable object. `f` takes a reference to that disposable object. So by our definition this is an **rvalue reference**. However this doesn't prevent the expression denoting this reference from being an object name, "x", so the reference expression itself is an **lvalue**.

Note that we cannot pass an lvalue to `f`, because an rvalue reference cannot bind to an lvalue. The following code:

```
MyClass x;
f(x);
```

triggers this compilation error:

```
error: cannot bind rvalue reference of type 'MyClass&&' to lvalue
of type 'MyClass'
f(x);
  ^
```

Understanding this made a big difference for me: an lvalue expression can designate an rvalue reference. If this doesn't sound crystal clear yet, I suggest you read this section over one more time before moving on.

There is a way to call f with our lvalue `x`: by casting it explicitly into an rvalue reference. This is what `std::move` does:

```
MyClass x;
f(std::move(x));
```

So when you `std::move` an lvalue, you need to be sure you won't use it any more, because it will be considered like a disposable object by the rest of the code.

Movable objects

In practice we don't encounter that many functions accepting rvalue references (except in template code, see below). There is one main case that accepts one though: move constructors:

```
class MyClass
{
public:
    // ...
    MyClass(MyClass&& other) noexcept;
};
```

Given what we've seen so far, we have all the elements to understand the meaning of this constructor. It builds an object using another one, like the copy constructor but, unlike in the copy constructor, no one cares about the object it is passed.

Using this information can allow the constructor to operate faster. Typically, an `std::vector` will steal the address of the memory buffer of the passed object, instead of politely allocating a new memory space and copying every elements over to it.

It also allows transferring ownership, like with `std::unique_ptr`.

Note that objects can also be **assigned to** from disposable instances, with the move assignment operator:

```
class MyClass
{
public:
    // ...
    MyClass& operator=(MyClass&& other) noexcept;
};
```

Even if this looks like the panacea for performance issues, let's keep in mind the guideline in Effective Modern C++'s Item 29 which is that when you don't know a type (like in generic code) assume that move operations are not present, not cheap and not used.

The case of templates

rvalue references have a very special meaning with templates. What made me understand how this works is the various talks and book items of Scott Meyers on this topic. So I will only sum it up, also because if you understood everything until now, there is not that much more here. And for more details I suggest you read Items 24 and 28 of Effective Modern C++.

Consider the following function:

```
template<typename T>
void f(T&& x)
{
    ...
}
```

`x` is an lvalue, nothing to question about that.

But even if it looks like it is an rvalue reference (it has `&&`), it may not be. In fact, by a tweak in template argument deduction, the following happens:

- `x` is an lvalue reference if `f` received an lvalue, and
- `x` is an rvalue reference if `f` received an rvalue.

This is called a forwarding reference or a universal reference.

For this to work though, it has to be exactly `T&&`. Not `std::vector<T>&&`, not `const T&&`. Just `T&&` (Well, the template parameter can be called something else than T of course).

Now consider the following code:

```
template<typename T>
void g(T&& x)
{
    ...
}

template<typename T>
void f(T&& x)
{
    g(x);
}
```

`g` also receives a forwarding reference. But it will always be an lvalue reference, regardless of what was passed to `f`. Indeed, in the call `g(x)`, "`x`" is an lvalue because it is an object name. So the forwarding reference `x` in `void g(T&& x)` is an lvalue reference.

To pass on to `g` the value with the same reference type as that was passed to `f`, we need to use `std::forward`:

```
template<typename T>
void g(T&& x)
{
    ...
}

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

`std::forward` keeps the reference type of `x`. So:

- if `x` is an rvalue reference then `std::forward` does the same thing as `std::move`,
- and if `x` is an lvalue reference then `std::forward` doesn't do anything.

This way the `x` in `g` will have the same reference type as the value initially passed to `f`.

This technique is called "perfect forwarding".

An illustrating example: std::make_unique

Let's see an example, with the implementation of `std::make_unique`. This helper function from the C++ standard library takes somes arguments and uses them to construct an object on the heap and wrap it into a `std::unique_ptr`.

Here is its implementation:

```
template<typename T, typename... Args>
std::unique_ptr<T> make_unique(Args&&... args)
{
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
}
```

(As observed by /r/Xeverous on Reddit, note this is not the exact official implementation as it doesn't cover all cases, in particular it should prevent an array with known bounds from compiling).

Note how the arguments `args` passed on to the constructor of `T`:

```
T(std::forward<Args>(args)...)
```

Indeed, for all we know, `T` could have several constructors that accept lvalue references or rvalue references. The purpose of `make_unique` is to hide the call to `new` but to pass on the arguments just like if we had passed them ourselves to `new`.

Here `std::forward` allows to keep the reference type of the arguments.

That's pretty much it... for an introduction

There is more to the subject, like reference types in methods prototypes, when and how move constructors are generated by the compiler, or how move constructors should avoid throwing exceptions and what implications this has, on `std::vector` in particular. You could look up a reference (hey what a pun) book for more about this.

But I hope the fundamental concepts are here. Those are the keys that made me understand lvalues, rvalues and their references and I hope that, with these keys, you can understand this topic more quickly than I did. And that it will be one less thing to figure out for you when you read code.

How to Construct C++ Objects Without Making Copies

Today's guest post is written by guest author Miguel Raggi. Miguel is a Computer Science and Math professor at UNAM, Mexico's largest university. He loves clean, expressive, performant C++ code (and strives to convince students to write it in this way!). Miguel is the author of *discreture*, an open source C++ library to efficiently generate combinatorial objects, such as combinations, partitions, set partitions, and many more.
Interested to write on Fluent C++ too? Check out the guest posting area.

C++ references are a powerful but tricky tool: used correctly, they can improve performance with little impact on the clarity of code. But used badly, they can hide performance issues, or even send a peaceful program into the realm of undefined behaviour.

In this post, we will explore how to use the various references of C++ to minimize copies when constructing an object that holds a value, and how in some cases we can even reach zero copies.

This article assumes that you're familiar with move semantics, lvalue, rvalue and forwarding references. If you'd like to be refreshed on the subject, you can take a look at lvalues, rvalues and their references.

Copying from an lvalue, moving from an rvalue

Let's imagine we have a `TextBox` class that holds a string, maybe to edit and display.

```
class TextBox
{
public:
   // constructors: see below
private:
   std::string text_;
};
```

We want to be able to construct a `TextBox` by passing it a `std::string`, and make a copy only when necessary. That is, when we pass it an lvalue. But when we pass it an rvalue, we would like to only move from that rvalue and into `text_`.

One way to go about this is to create two constructors:

```
class TextBox
{
public:
   explicit TextBox(const std::string& text) : text_(text) {}
   explicit TextBox(std::string&& text) : text_(std::move(text)) {}
private:
   std::string text_;
};
```

The first one takes an lvalue reference (no copy), and *copies* it into `text_` (one copy).

The second one takes an rvalue reference (no copy) and *moves* it into `text_` (no copy).

To make this class simpler, we can merge those two constructors into one:

```
class TextBox
{
public:
   explicit TextBox(std::string text) : text_(std::move(text)) {}
private:
   std::string text_;
};
```

What's going on here? If we pass it an lvalue, the copy constructor of `std::string` gets called to construct the `text` parameter (one copy), then `text` is moved into `text_` (no copy).

And if we pass it an rvalue, the move constructor of `std::string` gets called to construct the `text` parameter (no copy), and then text is moved into `text_` (no copy).

Referencing an lvalue, moving from an rvalue

But what if we don't need to modify or own the object that is passed to us? This is often the case with helper or connecting classes.

Then we really just need a reference or pointer to the object, not a full copy. For example, if we have a class called `TextDisplayer` whose main purpose is to display some text to the window, we would like to do something like this:

```
class TextDisplayer
{
public:
   explicit TextDisplayer(const std::string& text) : text_(text) {}
private:
   const std::string& text_;
};
```

And this sometimes works fine. Except that it has an error just waiting to happen.

Consider the following three construction contexts:

```
std::string txt = "Hello World";
TextDisplayer displayer1(txt); // fine!
TextDisplayer displayer2(get_text_from_file()); // error!
TextDisplayer displayer3("Hello World"); // error!
```

Oops. Versions two and three have undefined behavior lying in wait, because the references that `displayer2` and `displayer3` hold are now invalid, since they were destroyed right after the constructors finish.

What we really want is for `TextDisplayer` to hold a reference if we are given an lvalue (that we assume will keep on existing) or alternatively, hold (and own) the full string if given an rvalue (and acquire it by moving from it).

In either case, there is no reason to make a copy, so we would like to avoid it if possible. We will see how to do just that.

Forwarding references

So how do we make a class that holds a reference if given an lvalue, but moves (and owns) when given rvalues?

This is where forwarding references come in. We wish create a template `T` which will be deduced as:

- An lvalue reference if given an lvalue
- Not a reference if given an rvalue

Fortunately, some really smart people already thought of this and gave us reference collapsing. Here is how we would like to use it to make our wrapper that never makes a copy.

```
template <class T>
class TextDisplayer
{
public:
   explicit TextDisplayer(T&& text) : text_(std::forward<T>(text))
{}
private:
   T text_;
};
```

Note: in real code we would choose a more descriptive name for `T`, such as `String`. We could also add a `static_assert` that `std::remove_cvref<T>` should be `std::string`.

(As pointed out by FlameFire and John Lynch in the comments section, the template parameter `T` in the constructor is not a forwarding reference, contrary to what the first version of this article was suggesting. However, we shall make use of forwarding references below in the deduction guide and helper function.)

If we pass an lvalue reference to the constructor of `TextDisplayer`, `T` is deduced to be an `std::string&`, so no copies are made. And if we pass an rvalue reference, `T` is deduced to be an `std::string`, but it's moved in (as `T` is moveable in our case), so there are no copies made either.

Making the call site compile

Unfortunately, the following doesn't compile:

```
std::string txt = "Hello World";
TextDisplayer displayer(txt); // compile error!
```

It gives the following error (with clang)

```
error: no viable constructor or deduction guide for deduction of
template arguments of 'TextDisplayer'
```

```
   TextDisplayer displayer(txt);
                 ^
```

Strangely, using the rvalue version does compile and work (in C++17):

```
TextDisplayer displayer(get_string_from_file()); // Ok!
```

The problem when passing an lvalue is that constructor type deduction is done in two steps. The first step is to deduce the type for class template parameters (in our case, `T`) and instantiate the class. The second step is to pick a constructor, after the class has been instantiated. But once `T` is deduced to be a std::string, it can't choose the constructor taking a parameter of type `std:string&&`. Perhaps surprisingly, the constructor chosen in the second step doesn't have to be the one used for template parameter deduction.

We would then need to construct it like this:

```
TextDisplayer<std::string&> displayer1(txt);
```

which is not very elegant (but nonetheless works).

Let's see two ways of solving this: The way before C++17 and the C++17 way.

Before C++17, we can create a helper function similar to `make_unique` or any of the `make_*` functions, whose main purpose was to overcome the pre-C++17 limitation that the compiler can't deduce class templates using constructors.

```
template <class T>
auto text_displayer(T&& text)
{
   return TextDisplayer<T>(std::forward<T>(text));
}
```

In C++17 we got automatic deduction for class templates using constructors. But we also got something else that comes along with it: deduction guides.

In short, deduction guides are a way to tell the compiler how to deduce class templates when using a constructor, which is why we are allowed to do this:

```
std::vector v(first, last); // first and last are iterators
```

and it will deduce the value type of the `std::vector` from the value type of the iterators.

So we need to provide a deduction guide for our constructor. In our case, it consists in adding the following line:

```
template<class T> TextDisplayer(T&&) -> TextDisplayer<T>; // deduction guide
```

This allows us to write the following code:

```
std::string txt = "Hello World";
TextDisplayer displayer1(txt);
TextDisplayer displayer2(get_string_from_file());
```

and both cases compile. More importantly, they *never*, for any reason, make a copy of the string. They either move or reference the original.

Making it const

One thing that we lost from the original implementation of TextDisplayer which simply saved a reference, was the constness of the std::string reference. After all, we don't want to risk modifying the original std::string that the caller trusted us with! We should store a const reference when given an lvalue, not a reference.

It would be nice to simply change the declaration of the member variable text_ to something like:

```
const T text_; // doesn't work, see below
```

The `const` is effective when we are given rvalues, and `decltype(text_)` will be const std::string. But when given lvalues, `decltype(text_)` turns out to be `std::string&`. No `const`. Bummer.

The reason is that `T` is a reference, so const applies to the *reference itself,* not to what is *referenced to*. which is to say, the const does nothing, since every reference is already constant, in the sense that, unlike pointers, it can't "point" to different places. This is the phenomenon described in The Formidable Const Reference That Isn't Const.

We can work around this issue with a bit of template magic. In order to add const to the underlying type of a reference, we have to remove the reference, then add const to it, and then take a reference again:

```
using constTref =  const std::remove_reference_t<T>&;
```

Now we have to ask `T` whether it is a reference or not, and if so, use `constTref`. If not, use const `T`.

```
using constT = std::conditional_t<std::is_lvalue_reference_v<T>,
constTref, const T>;
```

And finally, we can just declare text_ as follows:

```
constT text_;
```

The above works in both cases (lvalues and rvalues), but is ugly and not reusable. As this is a blog about expressive code, we should strive to make the above more readable. One way is to add some extra helpers that can be reused: `const_reference`, which gives a const reference to a type (be it a reference or not), and `add_const_to_value`, which acts as `std::add_const` on normal types and as `const_reference` on references.

```
template<class T>
struct const_reference
{
   using type = const std::remove_reference_t<T>&;
};

template <class T>
using const_reference_t =  typename const_reference<T>::type;

template <class T>
struct add_const_to_value
{
   using type =  std::conditional_t<std::is_lvalue_reference_v<T>,
const_reference_t<T>, const T>;
};

template <class T>
using add_const_to_value_t =  typename add_const_to_value<T>::type;
```

And so our TextDisplayer class can now be declared like this:

```
class TextDisplayer
```

```
{
   // ...
private:
   add_const_to_valuet<T> text_;
};
```

Isn't there a risk of invalidating our references?

It's difficult (but possible) to invalidate our reference to the string. If we hold the string (when given an rvalue), there is no way for it to be invalidated. And when given an lvalue, if both the lvalue and TextDisplayer live in stack memory, we *know* the lvalue string will outlive the `TextDisplayer`, since the `TextDisplayer` was created after the string, which means the `TextDisplayer` will be deleted before the string. So we're good in all those cases.

But some more elaborate ways of handing memory in client code could lead to dangling references. Allocating a `TextDisplayer` on the heap, for example, as in `new TextDisplayer(myLvalue)`, or getting it from a `std::unique_ptr`, leaves the possibility of the `TextDisplayer` outliving the lvalue it is referring to, which would cause undefined behaviour when we try to use it.

One way to work around this risk would be disable `operator new` on `TextDisplayer`, to prevent non-stack allocations. Furthermore, as is always the danger when holding pointers or references, making copies of `TextDisplayer` could also lead to issues and should also be forbidden or redefined.

Finally, I guess we might still manually `delete` the string before `TextDisplayer` goes out of scope. It shouldn't be the common case, but I don't think there is anything we can do about that. But I'll be happy to be proven wrong in the comments section. Bonus points if your solution doesn't involve `std::shared_ptr` or any other extra free store allocations.