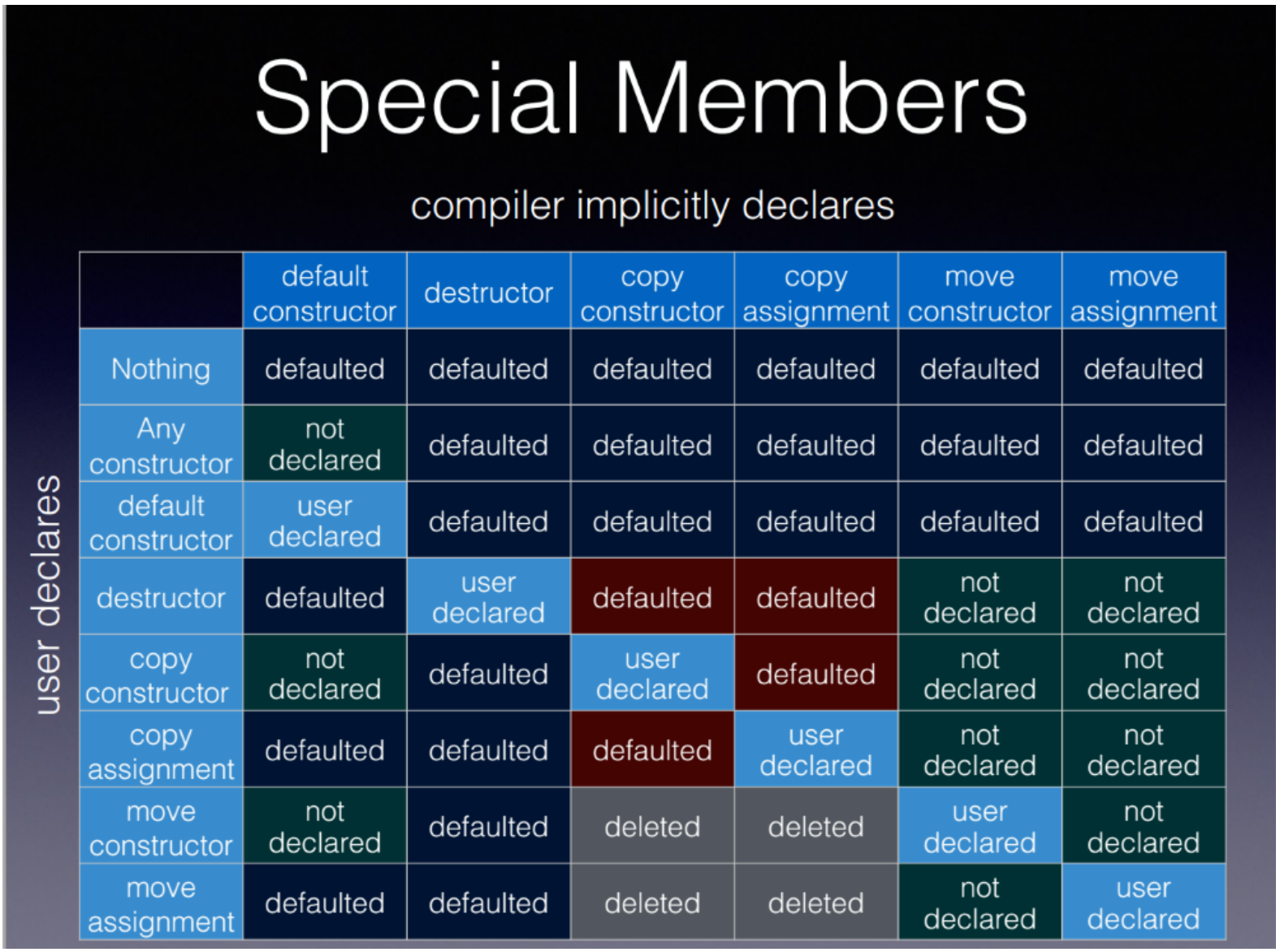
Tutorial: When to Write Which Special Member

When explaining someone the rules behind the special member functions and when you need to write which one, there is this diagram that is always brought up. I don't think the diagram is particularly useful for that, however.

It covers way more combinations than actually make sense. So let's talk about what you actually need to know about the special member functions and when you should write which combination.

The Special Member Function Diagram

The diagram in question was created by Howard Hinnant:



A couple of points need explanation:

- A "user-declared" special member function is a special member function that is in any way *mentioned* in the class: It can have a definition, it can be `defaulted`, it can be `deleted`. This means that writing `foo(const foo&) = default` prohibits a move constructor.
- A compiler declared "defaulted" special member behaves the same as `= default`, e.g. a defaulted copy constructor copy constructs all members.
- A compiler declared "deleted" special member behaves the same as `= delete`, e.g. if overload resolution decides to use that overload it will fail with an error that you are invoking a deleted function.
- If a compiler does not declare a special member, it does not participate in overload resolution This is different from a deleted member, which does participate. For example, if you have a copy constructor, the compiler will *not declare* move constructor. As such, writing `T obj(std::move(other))` will result in a call to a copy constructor. If on the other hand the move constructor were *deleted*, writing that would select the move constructor and then error because it is deleted.
- The behavior of the boxes marked red is deprecated, as the defaulted behavior in that case is dangerous.

Yes, that diagram is complicated. It was given in a talk about move semantics with the desired purpose of showing the generation rules.

But you don't need to know them, you only need to know which of the following situations apply.

Majority of Cases: Rule of Zero

```
class normal
{
public:
    // rule of zero
};
```

The absolute majority of classes do not need a destructor. Then you also don't need a copy/move constructor or copy/move assignment operator: The compiler generated defaults do the right thing™.

This is known as the rule of zero. Whenever you can, follow the rule of zero.

If you don't have any constructors, the class will have a compiler generated default constructor. If you have a constructor, it will not. In that case add a default constructor if there is a sensible default value.

> It is often not a good idea to introduce an artificial "null" state, just use `std::optional<T>` instead.

Container Classes: Rule of Five (Six)

```
class container
{
public:
    container() noexcept;
    ~container() noexcept;

    container(const container& other);
    container(container&& other) noexcept;

    container& operator=(const container& other);
    container& operator=(container&& other) noexcept;
};
```

If you need to write a destructor — because you have to free dynamic memory, for example — the compiler generated copy constructor and assignment operator will do the wrong thing. Then you have to provide your own.

This is known as the rule of five. Whenever you have a custom destructor, also write a copy constructor and assignment operator that have matching semantics. For performance reasons also write a move constructor and move assignment operator.

The move functions can steal the resources of the original objects and leave it in an empty state. Strive to make them `noexcept` and fast.

As you now have a constructor, there will not be an implicit default constructor. In most cases it makes sense to implement a default constructor that puts the class in the empty state, like the post-move one.

This makes it the rule of six.

Resource Handle Classes: Move-only

```
class resource_handle
{
public:
    resource_handle() noexcept;
    ~resource_handle() noexcept;

    resource_handle(resource_handle&& other) noexcept;
    resource_handle& operator=(resource_handle&& other) noexcept;

    // resource_handle(const resource_handle&) = delete;
    // resource_handle& operator=(const resource_handle&) = delete;
};
```

Sometimes you need to write a destructor but cannot implement a copy. An example would be class that wraps a file handle or a similar OS resource.

Make those classes *move-only*. In other words: write a destructor and move constructor and assignment operators.

If you look at Howard's chart, you'll see that in that case the copy constructor and assignment operators are deleted. This is correct, as the class should be move-only. If you want to be explicit, you can also manually `= delete` them.

Again, it makes sense to add a default constructor that puts it in the post-move state.

Immoveable Classes

```
class immoveable
{
public:
    immoveable(const immoveable&) = delete;
```

```
    immoveable& operator=(const immoveable&) = delete;

    // immoveable(immoveable&&) = delete;
    // immoveable& operator=(immoveable&&) = delete;
};
```

Sometimes you want that a class cannot be copied or moved. Once an object is created it will always stay at that address. This is convenient if you want to safely create pointers to that object.

In that case you want to delete your copy constructor. The compiler will then not declare a move constructor, meaning all kinds of copying or moving will try to invoke the copy constructor, which is deleted. If you want to be explicit, you can also manually `= delete` it.

You should also delete the assignment operator. While it does not physically move the object, assignment is closely related to the constructors, see below.

Avoid: Rule of Three

```
class avoid
{
public:
    ~avoid();

    avoid(const avoid& other);
    avoid& operator=(const avoid& other);
};
```

If you implement only copy operations, moving a class will still invoke copy. Lots of generic code assumes that a move operations is cheaper than a copy, so try to respect that.

If you have C++11 support, implement move for a performance improvement.

Don’t: Copy-Only Types

```
class dont
{
public:
    ~dont();

    dont(const dont& other);
    dont& operator=(const dont& other);

    dont(dont&&) = delete;
    dont& operator=(dont&&) = delete;
};
```

If you have copy operations, don’t manually delete the move operations. This is because all functions participate in overload resolution, even if they’re deleted:

```
dont a(other);            // calling with const T&, selecting copy
dont b(std::move(other)); // calling with T&&, selecting deleted move - compiler error
```

So if you’re passing your type to generic code that wants to use optimized operations and move whenever possible – such as the standard library, you will get compiler errors. If you hadn’t done anything, and simply omitted the move operations, overload resolution would have called the copy operations instead:

```
class no_move
{
public:
    ~no_move();

    no_move(const no_move& other);
    no_move& operator=(const no_move& other);

    // We don't mention move operations at all.
};

no_move a(no_move);            // calling with const T&, selecting copy
no_move b(std::move(no_move)); // calling with T&&, selecting copy as well
```

This might be less efficient, but if you can’t logically move your type, it’s the best you can do. At least it compiles.

Don’t: Deleted Default Constructor

```
class dont
{
public:
    dont() = delete;
};
```

There is no reason to `= delete` a default constructor, if you don't want one, write another one.

The only exception would be a type that cannot be constructed in any way, but such a type isn't really useful without language support for "bottom" or "never" types.

So just don't do it.

Don't: Partial Implementation

```
class dont
{
public:
    dont(const dont&);
    dont& operator=(const dont&) = delete;
};
```

> The following also applies to move construction and move assignment.

Copy construction and copy assignment are a pair. You either want both or none.

Conceptually, copy assignment is just a faster "destroy + copy construct" cycle. So if you have copy construct, you should also have copy assignment, as it can be written using a destructor call and construction anyway.

Generic code often requires that type can be copied. If it is not carefully crafted, it might not make a distinction between copy construction and copy assignment.

While there can be philosophical arguments for a type that can only be copy constructed and not assigned or vice-versa, do the pragmatic thing and avoid them.

Consider: Swap

```
class consider
{
public:
    friend void swap(consider& lhs, consider& rhs) noexcept;
};
```

Some algorithms, especially pre-move ones, use `swap()` to move objects around. If your type does not provide a `swap()` that can be found via ADL, it will use `std::swap()`.

`std::swap()` does three moves:

```
template <typename T>
void swap(T& lhs, T& rhs)
{
    T tmp(std::move(lhs));
    lhs = std::move(rhs);
    rhs = std::move(tmp);
}
```

If you can implement a faster `swap()`, do it. Of course, this only applies to classes that have a custom destructor, where you've implemented your own copy or move.

Your own `swap()` should always be `noexcept`.

Conclusion

Based on that I've created a new overview of the special member functions: special member chart

Next time you need to explain the rules, consider using this overview or this blog post instead of the generation diagram.