Move Semantic

We will talk about some important, often overlooked properties of the move semantic in this lesson.

WE'LL COVER THE FOLLOWING

Containers of the standard template library (STL) can have non-copyable elements. The copy semantic is the fallback for the move semantic. Let's learn more about the move semantic.

std::move

The function `std::move` moves its resource.

- The function needs the header `<utility>`.
- The function converts the type of its argument into a rvalue reference.
- The compiler applies move semantic to the rvalue reference.
- `std::move` is under the hood a `static_cast` to an rvalue reference.

```
static_cast<std::remove_reference<decltype(arg)>::type&&>(arg);
```

- What is happening here?
  - `decltype(arg)`: deduces the type of the argument

- `std::remove_reference<....>` removes all references from the type of the argument
- `static_cast<....&&>` adds two references to the type

> 💡 **Copy semantic is a fallback for move semantic.** This means if we invoke `std::move` with a non-moveable type, copy-semantic is used. This is due to the fact that an rvalue can be bound to an rvalue reference and a constant lvalue reference.

STL

Each container of the STL and `std::string` gets two new methods:

- Move constructor
- Move assignment operator

These new methods get their arguments as **non-constant** rvalue references.

Example

```
vector{
  vector(vector&& vec);                    //move constructor
  vector& operator = (vector&& vec);       //move assignment
  vector(const vector& vec);               //copy constructor
  vector& operator = (const vector& vec); // copy assignment
```

> The classical copy constructor and copy assignment operator get their arguments as **constant** lvalue references.

User-Defined Data Types

User-defined data types can support the move and copy semantics as well.

Example:

```
class MyData{
  MyData(MyData&& m) = default;
  MyData& operator = (MyData&& m) = default;
```

```
  MyData(const MyData& m) = default;


  MyData& operator = (const myData& m) = default;
};
```

> The move semantic has priority over the copy semantic.

The Strategy of the Move Constructor

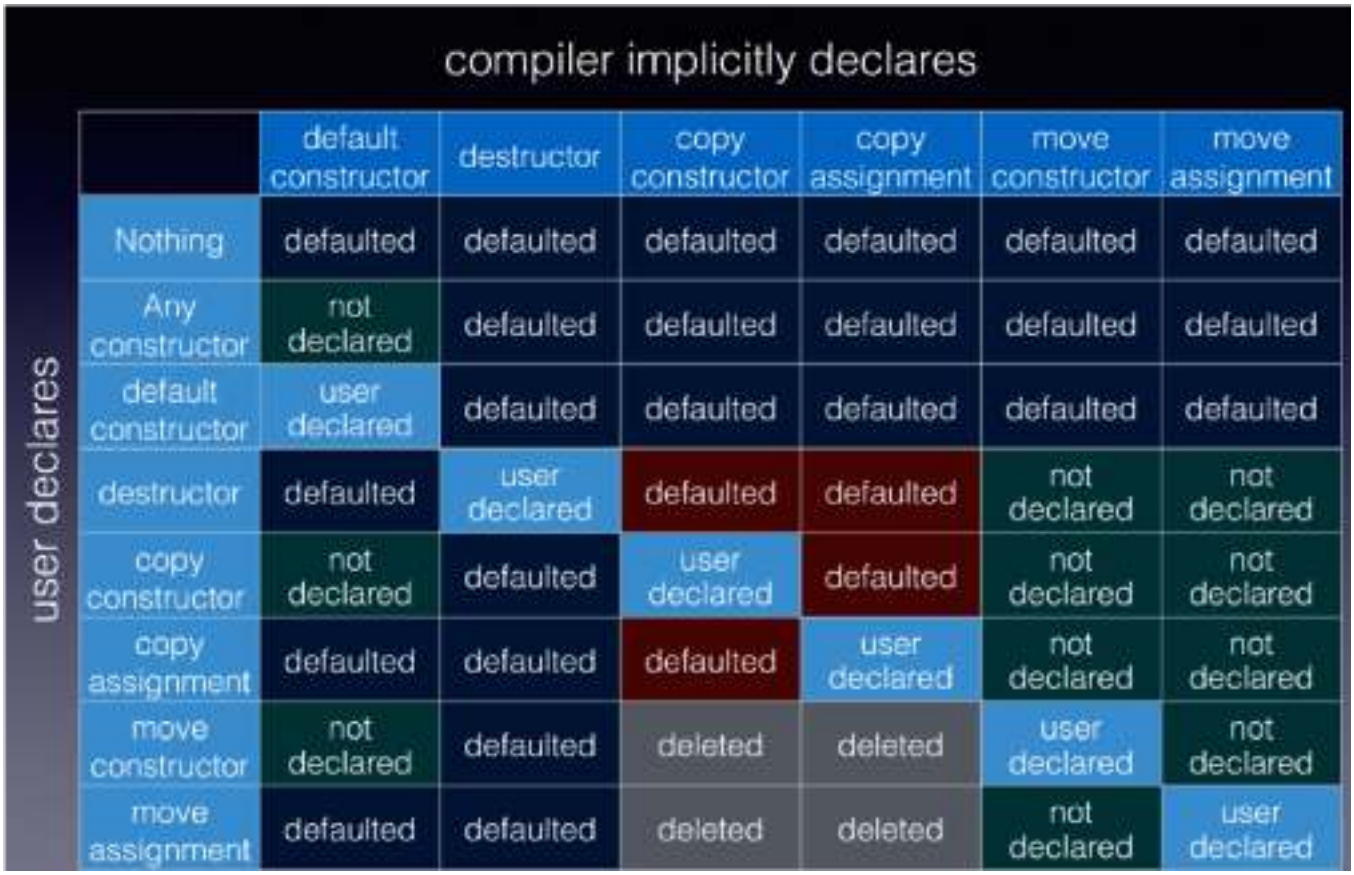
1. Set the attributes of the new object.
2. Move the content of the old object.
3. Set the old object in a valid state.

The move constructor is automatically created if all attributes of the class and all base classes have one move constructor. The automatically-created move constructor delegates its job to the attributes of the class and all base classes.

- This rule holds for the big six:
  - Default constructor
  - Destructor
  - Move and copy constructor
  - Move and copy assignment operator

Automatically Generated Methods

- The following table shows the dependency generated when you create one of the big six or a constructor.
- The key idea is it to start simple when you create a class. The compiler automatically creates the big six if your class and the base classes use
  - built-in data-types
  - Containers of the STL or `std::string`
- When your class has a pointer, put it into a `std::unique_ptr` or `std::shared_ptr` depending on the semantic you want to express
  - `std::shared_ptr` can be copied
  - `std::unique_ptr` cannot be copied



- **user-declared:** a method which is used (defined, defaulted, or deleted).
- **defaulted:** a method that the compiler generates or a method that is requested via the default.

How should you read this table?

- Start with the first column and move upwards.
  - When you create *Nothing*, you get all from the constructor to the move assignment operator to the right.
  - When you create *Any constructor*, you get no default constructor but the remaining five special methods.
  - When you create a *default constructor*, you get all five remaining specials methods.
  - And so on ...

We must explicitly mention that the compiler can only create special methods if our class is simple enough. If you want to have move details to the automatically created methods, watch Howard Hinnands presentation titles, Everything you need to know about move semantic.

Rule of Zero or Rule of Six

The rule of zero or the rule of six means that you must implement on or all of the six special methods.

> Do not implement one of the six special methods if not necessary. When the compiler complains because one of the six special methods is not available, think about the semantic of your user-defined type, and implement the missing methods.

Let's take a look at a few examples of copy and move semantic in the next lesson.