Non-const iterators allow you to modify what they point to:

```
std::vector<int> v{0};
std::vector<int>::iterator it = v.begin();
*it = 1;
assert(v[0] == 1);
```

Const iterators don't:

```
const std::vector<int> v{0};
std::vector<int>::const_iterator cit = v.begin();
// Compile time error: cannot modify container with const_iterator.
//*cit = 1;
```

As shown above, `v.begin()` is `const` overloaded, and returns either `iterator` or `const_iterator` depending on the const-ness of the container variable:

- How does begin() know which return type to return (const or non-const)?
- how does overloading of const and non-const functions work?

A common case where `const_iterator` pops up is when `this` is used inside a `const` method:

```
class C {
    public:
        std::vector<int> v;
        void f() const {
            std::vector<int>::const_iterator it = this->v.begin();
        }
        void g(std::vector<int>::const_iterator& it) {}
};
```

`const` makes `this` const, which makes `this->v` const.

You can usually forget about it with `auto`, but if you starting passing those iterators around, you will need to think about them for the method signatures.

Much like const and non-const, you can convert easily from non-const to const, but not the other way around:

```
std::vector<int> v{0};
std::vector<int>::iterator it = v.begin();

// non-const to const.
std::vector<int>::const_iterator cit = it;

// Compile time error: cannot modify container with const_iterator.
//*cit = 1;

// Compile time error: no conversion from const to no-const.
//it = ci1;
```

Which one to use: analogous to `const int` vs `int`: prefer const iterators whenever you can use them (when you don't need to modify the container with them), to better document your intention of reading without modifying.