# Functions and Classes

Let's learn about coming C++ concepts in detail in this lesson.

WE'LL COVER THE FOLLOWING



Concepts are part of the template declaration.

## Functions

Using the concept `Sortable`.

### Implicit

```
template<Sortable Cont>
void sort(Cont& container){
...
}
```

The container has to be `Sortable`.

The implicit version from the left is syntactic sugar to the explicit version:

### Explicit

```
template<typename Cont>
   requires Sortable<Cont>()
void sort(Cont& container){
    ...
}
```

`Sortable` has to be a constant expression that is a predicate. That means that the expression has to be evaluable at compile-time and has to return a boolean.

If you invoke the sort algorithm with a container `lst` that is not sortable, you will get a unique error message from the compiler.

- Usage:

```
std::list<int> lst = {1998, 2014, 2003, 2011};
sort(lst); // ERROR: lst is no random-access container with <
```

> You can use concepts for all kind of templates.

## Classes #

We can define a class template `MyVector` that will only accept objects as template arguments:

```
template<Object T>
class MyVector{};

MyVector<int> v1; // OK
MyVector<int&> v2 // ERROR: int& does not satisfy the constraint Object
```

Now, the compiler complains that the reference `(int&)` is not an object. `MyVector` can be further adjusted.

> A reference is not an object.

### Methods of a Class #

```
template<Object T>
class MyVector{
   ...
   requires Copyable<T>()
   void push_back(const T& e);
   ...
};
```

Now the method `push_back` from `MyVector` requires that the template

argument has to be copy-able. The concepts have to be placed before the method declaration.

## More Requirements

A template can have more than one requirement for its template parameters.

```
template <SequenceContainer S,
          EqualityComparable<value_type<S>> T>
Iterator_type<S> find(S&& seq, const T& val){
    ...
}
```

The function template `find` has two requirements. On one hand, the container has to store its elements in a linear arrangement (`SequenceContainer`), on the other hand, the elements of the container have to be equally comparable: `EqualityComparable<value_type<S>>`.

## Overloading

Concepts support the overloading of functions.

```
template<InputIterator I>
void advance(I& iter, int n){...}

template<BidirectionalIterator I>
void advance(I& iter, int n){...}

template<RandomAccessIterator I>
void advance(I& iter, int n){...}

std::list<int> lst{1,2,3,4,5,6,7,8,9};
std::list<int>:: iterator i = lst.begin();
std::advance(i, 2); // BidirectionalIterator
```

The function template `std::advance` puts its iterator `n` positions further. Depending if the iterator is a forward, a bidirectional, or a random access iterator, different function templates will be used. In case of a `std::list`, the `BidirectionalIterator` will be chosen.

Concepts also support the specialization of class templates.

## Specialization

```
template<typename T>
class MyVector{};

template<Object T>
class MyVector{};

MyVector<int> v1; // Object T
MyVector<int&> v2; // typename T
```

For `MyVector<int&> v2`, the compiler uses the general template in the first line; on the contrary, the compiler uses for `MyVector<int>` the specialization `template<Object T> class MyVector{}`.

- `MyVector<int&>` goes to the unconstrained template parameter.
- `MyVector<int>` goes to the constrained template parameter.

---

In the next lesson, we'll study the placeholder syntax.