Define your Concepts: Equal and Ord

In this lesson, we'll define the concepts Equal and Ord for C++.

WE'LL COVER THE FOLLOWING

`Eq` versus `Equal`

The Type Class Eq (Haskell)

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

The Concept Equal (C++)

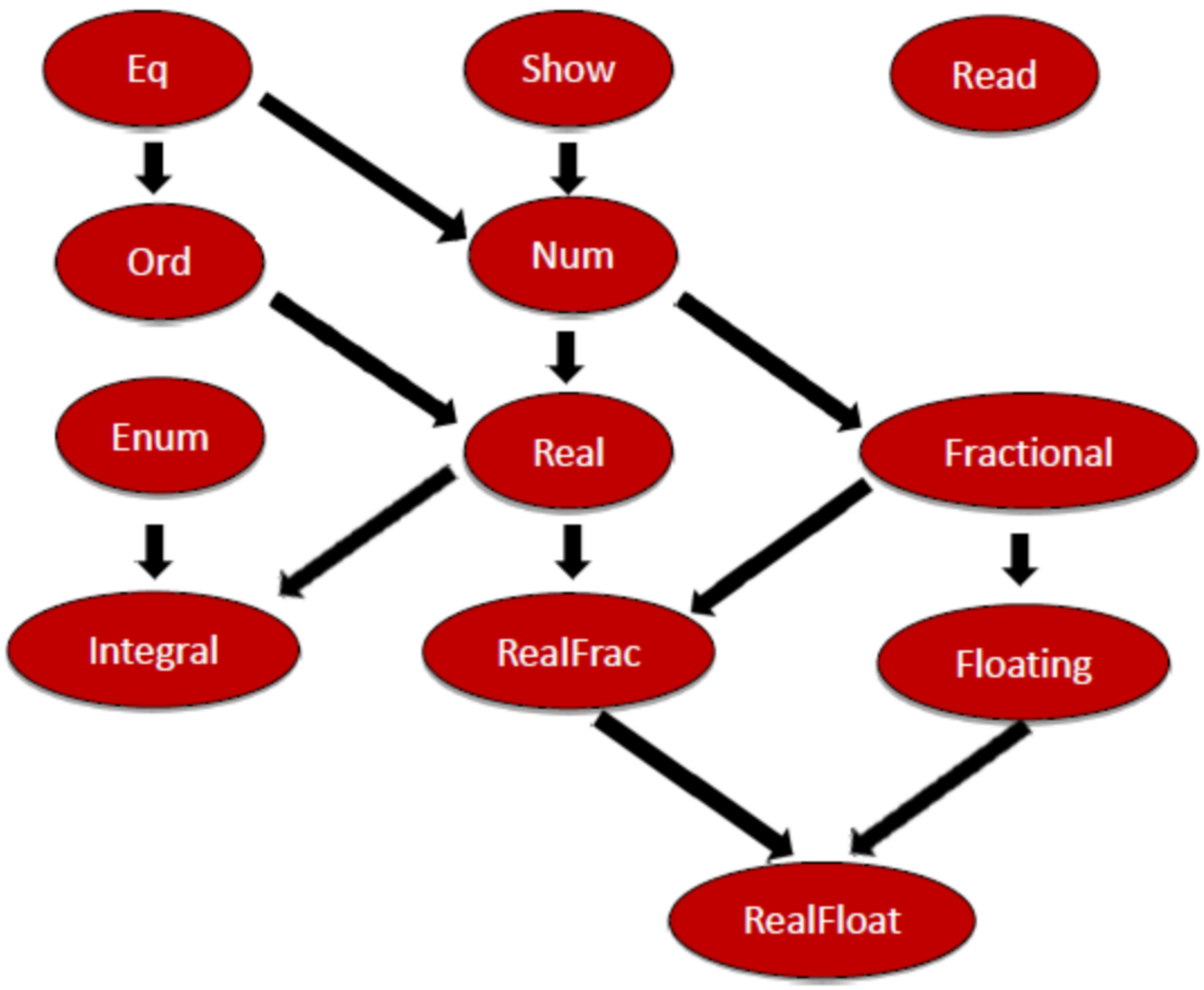
```
template <typename T>
concept bool Equal(){
  return requires(T a, T b){
    { a == b } -> bool;
    { a != b } -> bool;
  };
}
```

Let's have a closer look at Haskell's type class `Eq`. `Eq` requires from its instances, that

- they have equal `==` and unequal `/=` operation that returns a Bool.
- both take two arguments (`a -> a`) of the same type.

If you compare Haskell's type class with C++'s concept, you see the similarity.

Of course, the instances are the concrete types such as `int`.



Haskell Type Class

Now we have two questions in mind if we look at Haskell's type hierarchy above. How is the definition of the type class `Ord` in Haskell and can we model the inheritance relation in C++?

Haskell's Type Class `Ord`

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  max :: a -> a -> a
```

> Each type supporting `Ord` must support `Eq`.

The most interesting point about the typeclass `Ord` is the first line of its definition. An instance of the typeclass `Ord` has to be already an instance of the typeclass `Eq`. Ordering is an enumeration having the values `EQ`, `LT`, and `GT`.

The Concept `Equal` and the Concept `Ord`

Let's define the corresponding concepts in C++.

The concept Equal

```
template<typename T>
concept bool Equal(){
  return requires(T a, T b){
    { a == b } -> bool;
    { a != b } -> bool;
  };
}
```

The Concept Ord

```
template <typename T>
concept bool Ord(){
  return requires(T a, T b){
    requires Equal<T>();
      { a <= b } -> bool;
      { a < b } -> bool;
      { a > b } -> bool;
      { a >= b } -> bool;
  };
}
```

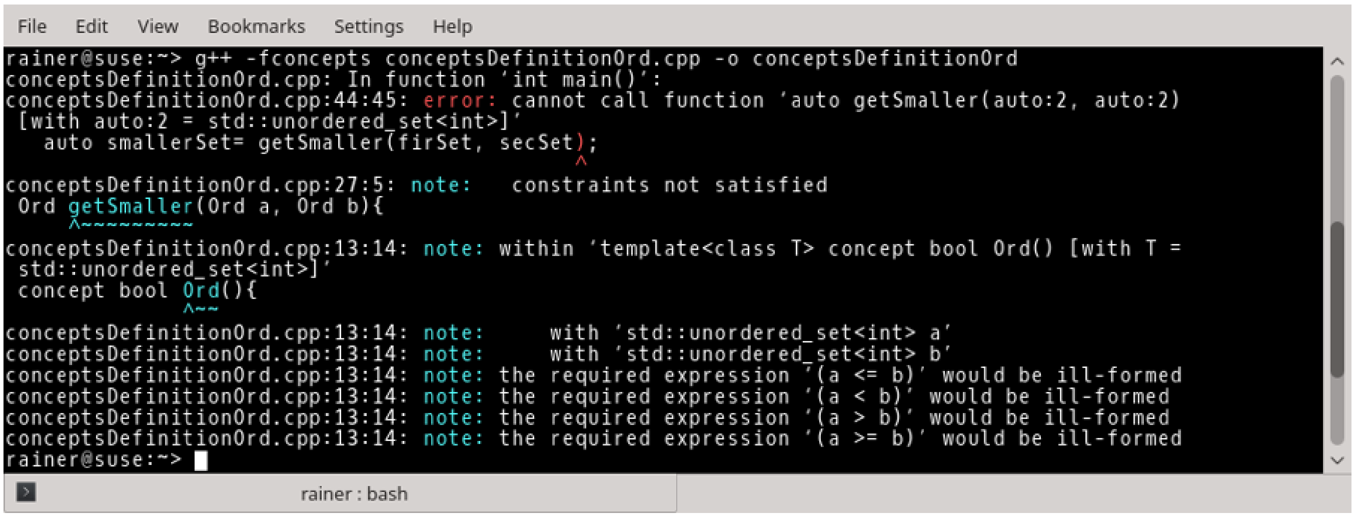
To make the job a little bit easier, we ignored the requirements `compare` and `max` from Haskell's type class in the `concept Ord`. The key point about the concept is that the line requires `Equal<T>()`. Here we required that the type parameter `T` has to fulfill the requirement `Equal`. If we use more requirements such as in the definition of the concept `Equal`, each requirement from top to bottom will be checked. That will be done in a short-circuiting evaluation. So, the first requirement returning `false` will end the process.

```
// conceptsDefintionOrd.cpp
#include <iostream>
#include <unordered_set>

template<typename T>
concept bool Equal(){
  return requires(T a, T b){
    { a == b } -> bool;
    { a != b } -> bool;
    { a != b } -> bool;
  };
}

template <typename T>
concept bool Ord(){
  return requires(T a, T b){
    requires Equal<T>();
    { a <= b } -> bool;
    { a < b } -> bool;
    { a > b } -> bool;
    { a >= b } -> bool;
  };
}

bool areEqual(Equal a, Equal b){
  return a == b;
}

Ord getSmaller(Ord a, Ord b){
  return (a < b) ? a : b;
}

int main(){

  std::cout << std::boolalpha << std::endl;

  std::cout << "areEqual(1, 5): " << areEqual(1, 5) << std::endl;

  std::cout << "getSmaller(1, 5): " << getSmaller(1, 5) << std::endl;

  std::unordered_set<int> firSet{1, 2, 3, 4, 5};
  std::unordered_set<int> secSet{5, 4, 3, 2, 1};

  std::cout << "areEqual(firSet, secSet): " << areEqual(firSet, secSet) << std::endl;

  auto smallerSet= getSmaller(firSet, secSet);

  std::cout << std::endl;

}
```

Equality and inequality are defined for the data types `int` and `std::unordered_set`.

What would happen, when we uncomment line 45 and compare `firSet` and `secSet`. To remind you, the type of both variables is `std::unordered_set`. This says very explicitly that they don't support an ordering.

Let's check what happens when we run this code:



```
rainer@suse:~> g++ -fconcepts conceptsDefinitionOrd.cpp -o conceptsDefinitionOrd
conceptsDefinitionOrd.cpp: In function ‘int main()’:
conceptsDefinitionOrd.cpp:44:45: error: cannot call function ‘auto getSmaller(auto:2, auto:2)
 [with auto:2 = std::unordered_set<int>]’
   auto smallerSet= getSmaller(firSet, secSet);
                                             ^
conceptsDefinitionOrd.cpp:27:5: note:   constraints not satisfied
 Ord getSmaller(Ord a, Ord b){
     ^~~~~~~~~~
conceptsDefinitionOrd.cpp:13:14: note: within ‘template<class T> concept bool Ord() [with T =
 std::unordered_set<int>]’
 concept bool Ord(){
              ^~~
conceptsDefinitionOrd.cpp:13:14: note:     with ‘std::unordered_set<int> a’
conceptsDefinitionOrd.cpp:13:14: note:     with ‘std::unordered_set<int> b’
conceptsDefinitionOrd.cpp:13:14: note: the required expression ‘(a <= b)’ would be ill-formed
conceptsDefinitionOrd.cpp:13:14: note: the required expression ‘(a < b)’ would be ill-formed
conceptsDefinitionOrd.cpp:13:14: note: the required expression ‘(a > b)’ would be ill-formed
conceptsDefinitionOrd.cpp:13:14: note: the required expression ‘(a >= b)’ would be ill-formed
rainer@suse:~>
```

Of course, the compilation would fail.

In the next lesson, we’ll discuss other predefined concepts: Regular and SemiRegular.