Predefined Concepts

Let's dive deep into predefined concepts of C++20 in this lesson.

WE'LL COVER THE FOLLOWING

Before moving on to predefined concepts, let's get to know about **Syntactic Sugar**.

> **Syntactic Sugar:** This is from Wikipedia: In computer science, syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express. It makes the language *sweeter* for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.

Predefined Concept

We should use the predefined concepts. cppreference.com gives a great introduction to predefined concepts. Here are a few of them:

- Core language concepts
  - Same
  - DerivedFrom
- Object concepts
  - Destructible
  - Constructible

    - ConvertibleTo
    - Common
    - Integral
    - Signed Integral
    - Unsigned Integral
    - Assignable
    - Swappable
- Comparison concepts
    - Boolean
    - EqualityComparable
    - StrictTotallyOrdered
    - DefaultConstructible
    - MoveConstructible
    - Copy Constructible
    - Movable
    - Copyable
    - Semi-regular
    - Regular
- Callable concepts
    - Callable
    - RegularCallable
    - Predicate
    - Relation
    - StrictWeakOrder

There are two ways to define concepts: variable concepts and function concepts. If we use a variable template for our concept, it's called a *variable concept*; in the second case a *function concept*.

Concepts Definition: Variable Concepts

```
template<typename T>
concept bool Integral =
    std::is_integral<T>::value;
}
```

We have defined the `concept Integral` by using a variable template. Variable templates are new with `C++14` and declare a family of variables. The `concept Integral` will evaluate to `true` if the predicate `std::is_integral<T>::value` returns `true` for `T`. `std::is_integral<T>` is a function of the type-traits library. The functions of the type-traits library enable, amongst other things, that we can check types at compile time.

Concepts Definition: Function Concepts

The original syntax of Concepts Technical Specification (Concepts TS) was a bit adjusted to the proposed Draft C++20 Standard. Here is the original syntax

from the Concepts TS, which is used in this course.

Concepts TS

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

`Integral` is a variable `concept` and `Equal` is a *function* concept. Both return a boolean.

- The type parameter `T` fulfills the variable concept Integral if `std::is_integral<T>::value` returns true.
- The type parameter `T` fulfills the function `concept Equal` if there are overloaded operators `==` and `!=` for `T` that returns a boolean.

Concepts Draft

The proposed syntax for C++20 is even more concise.

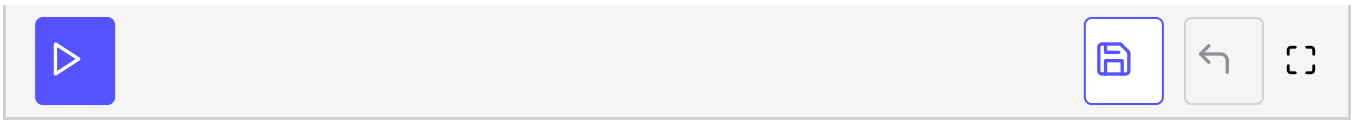
```
template<typename T>
concept Equal =
  requires(T a, T b) {
    { a == b } -> bool;
    { a != b } -> bool;
};
```

- `T` fulfills the function concept if `==` and `!=` are overloaded and return a boolean.

The Concept Equal

```
// conceptsDefintionEqual.cpp

#include <iostream>

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

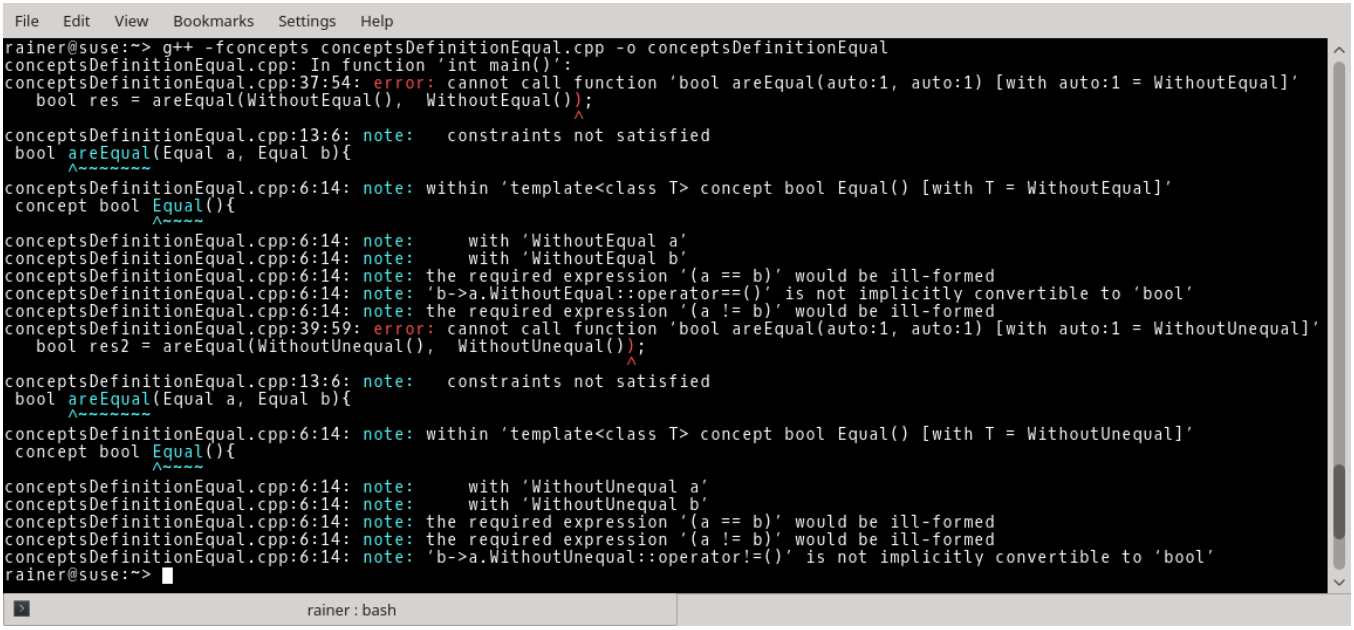
```
  };
}


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


struct WithoutEqual{
  bool operator==(const WithoutEqual& other) = delete;
};

struct WithoutUnequal{
  bool operator!=(const WithoutUnequal& other) = delete;
};



int main(){

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

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

  bool res = areEqual(WithoutEqual(),  WithoutEqual());

  bool res2 = areEqual(WithoutUnequal(),  WithoutUnequal());

  std::cout << std::endl;

}
```



We used the concept `Equal` in the (generic) function `areEqual` (line 13 to 15) and that's not so exciting.

What's more interesting is if we use the class `WithoutEqual` and `WithoutUnequal`. We set for both the `==` or respectively the `!=` operator to `delete`. The compiler complains immediately that both types do not fulfill the concept.

Let's have a look at the screenshot of the error taken from the machine:



In the next lesson, we will study the predefined concepts `Equal` and `Ord` in detail.