Define your Concept: Regular and SemiRegular

In this lesson, we'll gain an understanding of the important predefined concepts: Regular and SemiRegular.

WE'LL COVER THE FOLLOWING

The first question we have to answer is quite obvious. What is a Regular or a SemiRegular type? Our answer is based on the proposal p0898. We assume you may have already guessed it that Regular and SemiRegular are concepts, which are defined by other concepts. Given is the list of all concepts.

Regular

- DefaultConstructible
- CopyConstructible, CopyAssignable
- MoveConstructible, MoveAssignable
- Destructible
- Swappable
- EqualityComparable

SemiRegular

- Regular without EqualityComparable

The term Regular goes back to the father of the Standard Template Library Alexander Stepanov. He introduced the term in his book Fundamentals of Generic Programming. Here is a short excerpt. It's quite easy to remember the eight concepts used to define a regular type. First, there is the well-known rule of six:

- Default constructor: `X()`
- Copy constructor: `X(const X&)`

- Copy constructor: `X(const X&)`
- Copy assignment: `operator=(const X&)`
- Move constructor: `X(X&&)`
- Move assignment: `operator=(X&&)`
- Destructor: `~X()`

Second, add the Swappable and `EqualityComparable` concepts to it. There is a more informal way to say that a type `T` is regular: `T` behaves like an `int`.

To get SemiRegular, we have to subtract `EqualityComparable` from Regular.

References are not Regular

Thanks to the type-traits library the following program checks at compile-time if `int&` is a SemiRegular type.

Let’s have a look at the example:

```
#include <iostream>
#include <type_traits>

int main(){

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

    std::cout << "std::is_default_constructible<int&>::value: " << std::is_default_constructi
    std::cout << "std::is_copy_constructible<int&>::value: " << std::is_copy_constructible<in
    std::cout << "std::is_copy_assignable<int&>::value: " << std::is_copy_assignable<int&>::v
    std::cout << "std::is_move_constructible<int&>::value: " << std::is_move_constructible<in
    std::cout << "std::is_move_assignable<int&>::value: " << std::is_move_assignable<int&>::v
    std::cout << "std::is_destructible<int&>::value: " << std::is_destructible<int&>::value <
    std::cout << std::endl;
    //std::cout << "std::is_swappable<int&>::value: " << std::is_swappable<int&>::value << st

    std::cout << std::endl;
}
```

First of all, the function `std::is_swappable` requires C++17 that’s why we have commented it out otherwise it will give an error. We see that the reference such as `int&` is not default-constructible. The output shows that a reference is not SemiRegular and, therefore, not Regular. To check, if a type is Regular at compile-time, we need a function `isEqualityComparable` which is not part of the type-traits library. Let’s define it.

```
#include <experimental/type_traits>
#include <iostream>

template<typename T>
using equal_comparable_t = decltype(std::declval<T&>() == std::declval<T&>());

template<typename T>
struct isEqualityComparable:
       std::experimental::is_detected<equal_comparable_t, T>{};

struct EqualityComparable { };
bool operator == (EqualityComparable const&, EqualityComparable const&) { return true; }

struct NotEqualityComparable { };

int main() {

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

    std::cout << "isEqualityComparable<EqualityComparable>::value: " <<
                 isEqualityComparable<EqualityComparable>::value << std::endl;

    std::cout << "isEqualityComparable<NotEqualityComparable>::value: " <<
                 isEqualityComparable<NotEqualityComparable>::value << std::endl;

    std::cout << std::endl;

}
```

The new feature is in the experimental namespace in line 1. Line 9 is a crucial one. It detects at compile-time if the expression in line 5 is valid for the type `T`. The type-trait `isEqualityComparable` works for an `EqualityComparable` (line 11) and a `NotEqualityComparable` (line 14) type. Only `EqualityCompable` returns `true` because we overloaded the Equal-Comparison operator.

Now, we have all the ingredients to define Regular and SemiRegular. Here are our new type-traits.

```
#include <experimental/type_traits>
#include <iostream>

template<typename T>
using equal_comparable_t = decltype(std::declval<T&>() == std::declval<T&>());

template<typename T>
struct isEqualityComparable:
       std::experimental::is_detected<equal_comparable_t, T>
       {};
```

```
    {};

template<typename T>
struct isSemiRegular: std::integral_constant<bool,
                                      std::is_default_constructible<T>::value &&
                                      std::is_copy_constructible<T>::value &&
                                      std::is_copy_assignable<T>::value &&
                                      std::is_move_constructible<T>::value &&
                                      std::is_move_assignable<T>::value &&
                                      std::is_destructible<T>::value &&
                                      std::is_swappable<T>::value >{};

template<typename T>
struct isRegular: std::integral_constant<bool,
                                         isSemiRegular<T>::value &&
                                         isEqualityComparable<T>::value >{};


int main(){

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

    std::cout << "isSemiRegular<int>::value: " << isSemiRegular<int>::value << std::endl;
    std::cout << "isRegular<int>::value: " << isRegular<int>::value << std::endl;

    std::cout << std::endl;

    std::cout << "isSemiRegular<int&>::value: " << isSemiRegular<int&>::value << std::endl;
    std::cout << "isRegular<int&>::value: " << isRegular<int&>::value << std::endl;

    std::cout << std::endl;

}
```

The usage of the new type-traits `isSemiRegular` and `isRegular` makes the main program quite readable.

In this chapter, we have learned about the future concept of C++20. Let's move on to the next lesson for the conclusion of this course.