Type-Traits Overview

In this lesson, we'll study the type traits library and its goals along with type-checks. This section could only provide an overview of the many functions of the type-traits library.

WE'LL COVER THE FOLLOWING

- Type-Traits Library
- Type-Traits: Goals
  - Optimization
  - Correctness
- Type Checks

Type-Traits Library

Type-traits enable type checks, type comparisons, and type modifications at compile-time.

Below are some applications of template metaprogramming:

- Programming at compile-time
- Programming with types and values
- Compiler translates the templates and transforms it in C++ source code

We need to add a `type_traits` library in the header to enable all the functions present in the library.

```
#include <type_traits>
```

Type-Traits: Goals

If you look carefully, you'll see that type-traits have a significant optimization potential. In the first step, type-traits help to analyze the code at compile-time and in the second step, to optimize the code based on that analysis. How is

that possible? Depending on the type of variable, a faster variant of an algorithm will be chosen.

Optimization

- Code that optimizes itself. Depending on the type of a variable another code will be chosen.
- Optimized version of `std::copy`, `std::fill`, or `std::equal` is used so that algorithms can work on memory blocks.
- The optimized version of operations happens on all the elements in a container in one step and not on each element individually.

Correctness

- Type checks will be performed at compile-time.
- Type information, together with `static_assert`, defines the requirements for the code.
- With the concepts in C++20, the correctness aspect of the type-traits becomes less important.

Type Checks

C++ has 14 primary type categories. They are complete and orthogonal. This means, that each type is a member of exactly one type category. The check for the type categories is independent of the type qualifiers `const` or `volatile`.

Let's have a look at these categories syntactically:

```
template <class T> struct is_void;
template <class T> struct is_integral;
template <class T> struct is_floating_point;
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_reference;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;
template <class T> struct is_lvalue_reference;
template <class T> struct is_rvalue_reference;
```

We can divide the type-traits into smaller sets for simplicity.

- Primary type category (`::value`)

```
std::is_pointer<T>,
std::is_integral<T>,
std::is_floating_point<T>
```

- Composed type category (`::value`)

```
std::is_arithmetic<T>,
std::is_object<T>
```

- Type comparisons (`::value`)

```
std::is_same<T,U>,
std::is_base_of<Base,Derived>,
std::is_convertible<From,To>
```

- Type transformation (`::type`)

```
std::add_const<T>,
std::remove_reference<T>
std::make_signed<T>,
std::make_unsigned<T>
```

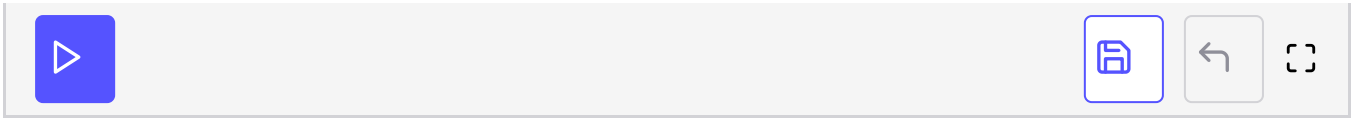
- Others (`::type`)

```
std::enable_if<bool,T>
std::conditional<bool,T,F>
std::common_type<T1, T2, T3, ... >
```

The above-mentioned functions, from the type-traits, give only a rough idea of their power. To learn more about type checks, click here. The above-mentioned functions are available on the given link with more detail.

```
// removeConst.cpp

#include <iostream>
#include <string>
#include <type_traits>

namespace rgr{

  template<class T, class U>
  struct is_same : std::false_type {};
```

```
  template<class T>
  struct is_same<T, T> : std::true_type {};


  template< class T >
  struct remove_const{
    typedef T type;
  };

  template< class T >
  struct remove_const<const T> {
    typedef T type;
  };
}


int main(){

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

  std::cout << std::is_same<int,std::remove_const<const int>::type>::value << std::endl;
  std::cout << rgr::is_same<int,rgr::remove_const<const int>::type>::value << std::endl;

  typedef rgr::remove_const<double>::type myDouble;
  std::cout << rgr::is_same<double,myDouble>::value << std::endl;

  typedef rgr::remove_const<const std::string>::type myString;
  std::cout << rgr::is_same<std::string,myString>::value << std::endl;

  typedef rgr::remove_const<std::add_const<int>::type>::type myInt;
  std::cout << rgr::is_same<int,myInt>::value << std::endl;

  std::cout << std::endl;

}
```



We have implemented `is_same` and `remove_const` in the namespace `rgr`. This corresponds to the type-traits library. For simplicity reason, we use the static constants `std::false_type` and `std::true_type` (lines 10 and 13). Thanks to the base class `std::false_type`, the class template has a member value. Respectively for `std::true_type`. The key observation of the class template `is_same` is to distinguish the general template (lines 9 and 10) from the partially specialized template (line 12 and 13). The compiler will use the partially specialized template if both template arguments have the same type. The partially specialized template, as opposed to the general template, has only one type parameter.

Our reasoning for the class template `remove_const` is similar. The general template returns, via its member `type`, exactly the same type; the partially specialized template returns the new type after removing the `const` property.

specialized template returns the new type after removing the `const` property

(line 22). The compiler will choose the partially specialized template if its template argument is `const`.

The rest is quickly explained. In lines 31 and 32, we used the functions of the type-traits library and our own versions. We declared a `typedef mydouble` (line 34), a `type myString` (line 37), and a `type myInt` (line 40). All types are non-constant.

In the next lesson, we'll study type-traits correctness and their optimization with the help of an example.