Type-Traits

In this lesson, we will learn about type traits, their advantages, and their application in embedded programming.

WE'LL COVER THE FOLLOWING

Template Metaprogramming

Template Metaprogramming is programming at compile time. However, what does template metaprogramming have in common with the type-traits library? As it turns out, quite a lot! The type-traits library is pure template metaprogramming enclosed in a library.

Applications of Template Metaprogramming

- Programming at compile time.
- Programming with types such as `double` or `int` and not with values such as 5.5 or 5.

- Compiler instantiates the templates and generates C++ code.

Type Traits: Goals

The type-traits library requires the following header:

```
#include <type_traits>
```

If you look carefully, you will see that type-traits have a big optimization potential. In the first step, the type-traits analyze the code at the compile time, and in the second step, they optimize the code based on that analysis. How is that possible? Depending on the type of the variable, a faster variant of an algorithm will be chosen.

Optimization

- Optimised version of `std::copy`, `std::fill`, or `std::equal` is used so the algorithms can work on memory blocks.

Correctness

- Type checks will be performed at compile time
- Together with `static_assert`, the type information defines the requirements for the code.

Check Types

The type-trait library supports primary and composite type categories. You get the answer with the attribute `::value`.

Primary Type Categories

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

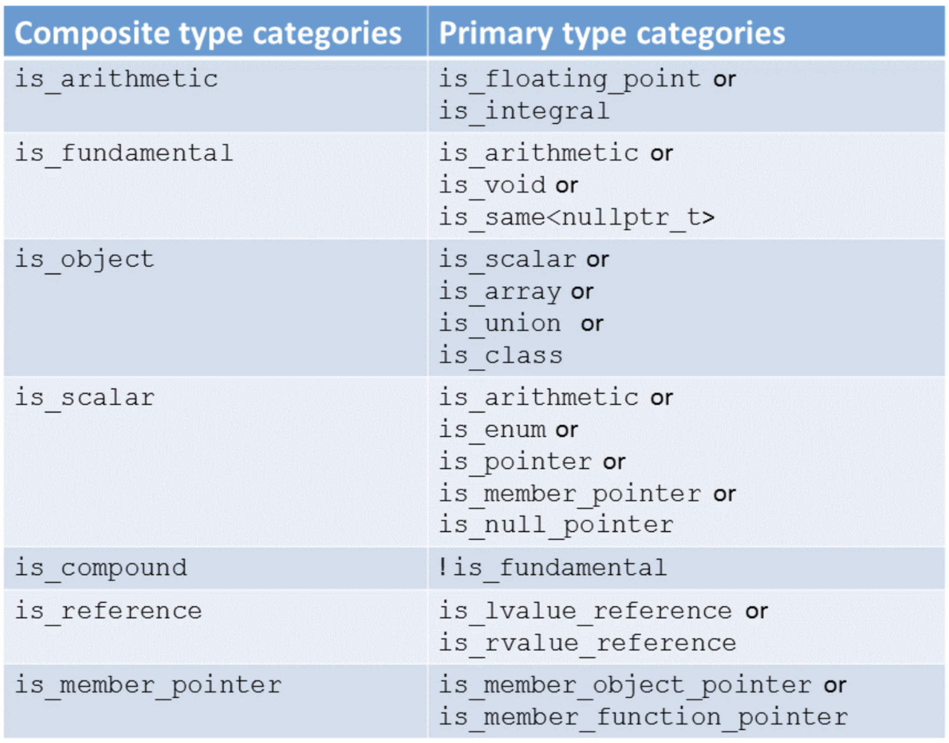
The 14 primary type categories are as follows:

```
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;
```

Composite Type Categories

Based on the 14 primary type categories, there are 7 composite type categories in C++.



Composite type categories	Primary type categories
`is_arithmetic`	`is_floating_point` or `is_integral`
`is_fundamental`	`is_arithmetic` or `is_void` or `is_same<nullptr_t>`
`is_object`	`is_scalar` or `is_array` or `is_union` or `is_class`
`is_scalar`	`is_arithmetic` or `is_enum` or `is_pointer` or `is_member_pointer` or `is_null_pointer`
`is_compound`	`!is_fundamental`
`is_reference`	`is_lvalue_reference` or `is_rvalue_reference`
`is_member_pointer`	`is_member_object_pointer` or `is_member_function_pointer`

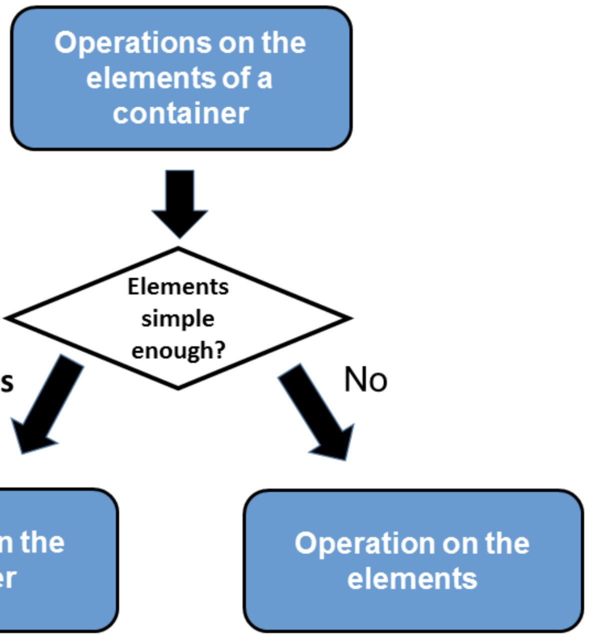
Performance - Working on the Entire Memory Area

The idea is quite straightforward and is used in current implementations of the Standard Template Library (STL). If the elements of a container are simple enough, the algorithm of the STL, such as `std::copy`, `std::fill`, or `std::equal`, will directly be applied to the memory area. Instead of using `std::copy` to copy the elements individually, the process is completed in one, complete step. Internally, C functions like `memcmp`, `memset`, `memcpy`, or `memmove` are used. The small difference between `memcpy` and `memmove` is that `memmove` can handle overlapping memory areas.

The implementations of the algorithm `std::copy`, `std::fill`, or `std::equal` use a simple strategy as described below.

`std::copy` functions like a wrapper. This wrapper checks if the element is simple enough to perform the funtion. If so, the wrapper will delegate the work to the optimized copy function. If not, the general copy algorithm will be used instead. This algorithm copies each subsequent element after each other. To make the right decision, if the elements are simple enough, the functions of the type-traits library will be used instead.

The following illustration explains this concept further:


Operations on the elements of a container
Elements simple enough?
Yes
No
Operation on the container
Operation on the elements

Example - std::fill

```
// fill.cpp

#include <cstring>
#include <chrono>
#include <iostream>
#include <type_traits>

namespace my{

  template <typename I, typename T, bool b>
  void fill_impl(I first, I last, const T& val, const std::integral_constant<bool, b>&){
    while(first != last){
      *first = val;
      ++first;
    }
  }
  template <typename T>
  void fill_impl(T* first, T* last, const T& val, const std::true_type&){

    std::memset(first, val, last-first);
  }

  template <class I, class T>
  inline void fill(I first, I last, const T& val){
    // typedef std::integral_constant<bool,std::has_trivial_copy_assign<T>::value && (sizeof(
    typedef std::integral_constant<bool,std::is_trivially_copy_assignable<T>::value && (sizeo
    fill_impl(first, last, val, boolType());
  }
}

const int arraySize = 100000000;
char charArray1[arraySize]= {0,};
char charArray2[arraySize]= {0,};

int main(){

  std::cout << std::endl;

  auto begin= std::chrono::system_clock::now();
  my::fill(charArray1, charArray1 + arraySize,1);
  auto last=  std::chrono::system_clock::now() - begin;
  std::cout <<  "charArray1: " << std::chrono::duration<double>(last).count() << " seconds" <

  begin= std::chrono::system_clock::now();
  my::fill(charArray2, charArray2 + arraySize, static_cast<char>(1));
  last=  std::chrono::system_clock::now() - begin;
  std::cout <<  "charArray2: " << std::chrono::duration<double>(last).count() << " seconds" <

  std::cout << std::endl;

}
```

Explanation

- Code in line 27 makes the decision, to which implementation of `my::fill_impl` is applied. To use the optimized variant, the elements should have a compiler-generated copy assignment operator `std::is_trivially_copy_assignable<T>`, and it should be 1 byte large: `sizeof(T) == 1`. The function `std::is_trivially_copy_assignable` is part of the type-traits library.

- If the expression `boolType()` in line 27 is `true`, the optimized version of `my::fill_impl` in the lines 18 - 21 will be used. As opposed to the generic variant `my::fill_impl`, this variant (line 10 -16) fills the entire memory area - consisting of 100 million entries - with the value 1. `sizeof(char)` is 1.

1.

What about the performance of the program? We compiled the program without optimization. The execution of the optimized variant is approximately 3 times faster on Windows and 20 times faster on Linux.

Type Comparisons

The type-traits library support three kinds of comparisons:

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

Due to its member value, each class template returns `true` or `false` and is, therefore, the optimal fit for `static_assert`.

Type Transformations

Although the C++ standard speaks about the modification or transformation of types that are not accurate, there is no state at compile time, meaning that there is nothing to modify. You can only generate new types upon specific request. The type-traits library is template metaprogramming in a very beautiful robe. Template metaprogramming is a purely functional language that is embedded in C++. Purely functional languages have no state. With that in mind, let's continue discussing the modification of types in this lesson.

The type-traits library has many functions to modify types at compile time. Therefore, you can remove `const` or `volatile` properties from a type or add a new one to it. There is one more thing to note: You must remove the sign of a type or the dimension of an array.

Here is the overview of that technique:

```
// const-volatile modifications
template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;
template <class T> struct add_const;
template <class T> struct add_volatile;
template <class T> struct add_cv;
```

```
// reference modifications
template <class T> struct remove_reference;
template <class T> struct add_lvalue_reference;
template <class T> struct add_rvalue_reference;

// sign modifications
template <class T> struct make_signed;
template <class T> struct make_unsigned;

// array modifications
template <class T> struct remove_extent;
template <class T> struct remove_all_extents;

// pointer modifications
template <class T> struct remove_pointer;
template <class T> struct add_pointer;
```

In order to get from a reference `int&` at compile time, you have to use the member `type` of the class template. In C++14, this is a lot easier. You must only add `_t` to the function. This holds for all invocated functions of this section.

```
std::cout << std::is_same<int, std::remove_reference<int &>::type>::value
<< std::endl; // true
std::cout << std::is_same<int, std::remove_reference_t<int &>>::value << s
td::endl; // true
```

The key of the code snippet states that you can write with C++14 `std::remove_reference<int &>::type` in the form `std::remove_reference_t<int &>`. Thanks to `::value`, you return the result of the comparison `std::is_same`.

The examples in the next lesson will build on your understanding of this topic.