

Type Traits: Categories and Transformations

This lesson is an extension of the previous lesson where we'll study categories and transformations of type traits.

WE'LL COVER THE FOLLOWING

- Primary type categories
- Composite type categories
- Performance - working on the entire memory area
- Example - `std::fill`
 - Explanation
- Type comparisons
- Type transformations
- Further information

Primary type categories

C++ has 14 primary type categories. They are complete and orthogonal, meaning that each type is exactly a 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;
```



Composite type categories

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

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

Performance - working on the entire memory area

This idea is quite straightforward and is used in current implementations of the 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 be directly 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 `memcpy`, `memset`, `strcpy`, or `memmove` are used. The small difference between `memcpy` and `memmove` is that only `memmove` can handle overlapping memory areas.

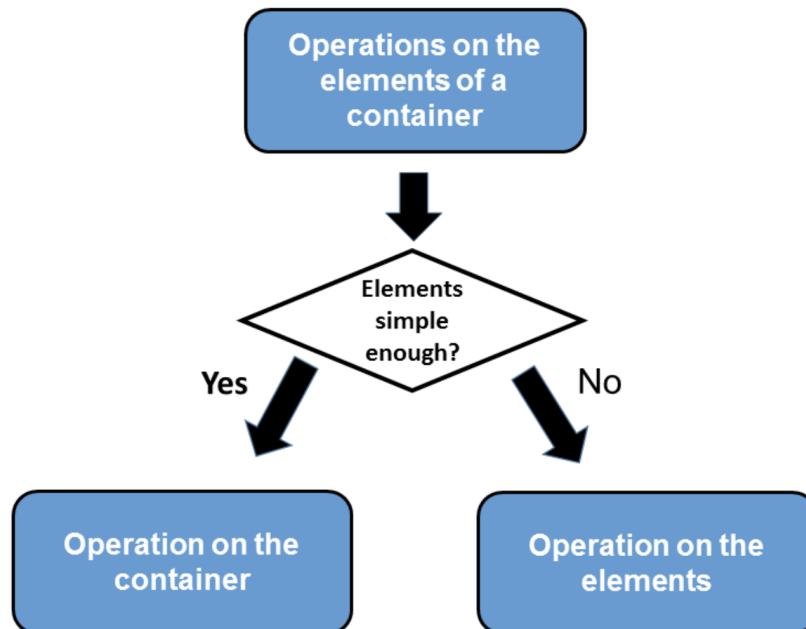
The implementations of the algorithms `std::copy`, `std::fill`, or `std::equal` use, is a simple strategy as described below.

`std::copy` functions as a wrapper. This wrapper checks if the element is simple enough to perform the function. 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, one after

another. 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:



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 && (sizeof(
        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

- The code in line 27 makes the decision, to which the 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` (line 10 -16), this variant fills the entire memory area - consisting of 100 million entries - with the value 1. `sizeof(char)` is 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. We can only generate new types upon 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, we can remove `const` or `volatile` properties from a type or add a new one to it. There is one more thing to note: We 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, we have to use the member `type` of the class template. In C++14, this is a lot easier. We 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 we can write `std::remove_reference<int &>::type` with C++14, in the form `std::remove_reference_t<int &>`. Thanks to `::value`, we return the result of the comparison `std::is_same`.

Further information

- [template metaprogramming](#)
- [purely functional language](#)

In the next lesson, we'll discuss examples of type traits.