Type-Traits (Correctness and Optimization)

In this lesson, we'll study type-traits correctness and their optimization using a gcd (greatest common divisor) algorithm along with fill and equal (type-trait features).

WE'LL COVER THE FOLLOWING

Correctness

The reason we use type-traits is correctness and optimization. Let's start with correctness. The idea is to implement generic gcd algorithms, step by step, to make it more type-safe with the help of the type-traits library.

gcd - The First

Our starting point is the euclid algorithm to calculate the greatest common divisor of two numbers.

It's quite easy to implement the algorithm as a function template and feed it with various arguments. Let's start!

```
#include <iostream>
```

```
#include <iostream>

template<typename T>
T gcd(T a, T b){
if( b == 0 ) return a;
  else return gcd(b, a % b);
}

int main(){
  std::cout << gcd(100, 10) << std::endl;  // 10
  std::cout << gcd(100, 33) << std::endl;  // 1
  std::cout << gcd(100, 0) << std::endl;   // 100
  std::cout << gcd(3.5, 4.0) << std::endl; // ERROR
  std::cout << gcd("100", "10") << std::endl;  // ERROR
  std::cout << gcd(100, 10L) << std::endl;  // ERROR
}
```

The function template has two serious issues.

- First, it is too generic. The function template accepts doubles (line 13) and C strings (line 14). But it makes no sense to determine the greatest common divisor of both data types. The modulo operation (`%`) for the double and the C string values fails in line 6. But that’s not the only issue.
- Second, gcds depend on one type parameter, `T`. This shows the function template signature `gcd(T a, T b))`. `a` and `b` have to be of the same type `T`. There is no conversion for type parameters. Therefore, the instantiation of gcd with an int type and a long type (line 15) fails.

gcd - The Second

We can ignore the rest of the examples below where both arguments have to be positive numbers. The `static_assert` operator and the predicate `std::is_integral<T>::value` will help us to check at compile-time whether `T` is an integral type. A predicate always returns a boolean value.

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

template<typename T>
T gcd(T a, T b){
  static_assert(std::is_integral<T>::value, "T should be integral type!");
  if( b == 0 ) return a;
  else return gcd(b, a % b);
}
int main(){
  std::cout << gcd(3.5, 4.0) << std::endl;
  std::cout << gcd("100", "10") << std::endl;
```

```
}
```

Great. We have solved the first issue of the gcd algorithm. The compilation will not fail by accident because the modulo operator is not defined for a double value and a C string. The compilation fails because the assertion in line 6 will not hold true. The subtle difference is that we now get an exact error message and not a cryptic output of a failed template instantiation as in the first example.

But what about the second issue. The gcd algorithm should accept arguments of a different type.

gcd - The Third

That's no big deal. But wait, what should the type of result be?

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

template<typename T1, typename T2>
??? gcd(T1 a, T2 b){
  static_assert(std::is_integral<T1>::value, "T1 should be integral!");
  static_assert(std::is_integral<T2>::value, "T2 should be integral!");
  if( b == 0 )
    return a;
  else
    return gcd(b, a % b);
}

int main(){
  std::cout << gcd(100, 10L) << std::endl;
}
```

The three questions marks in line 5 show the core of the issue. Should the first type or the second type be the return type of the algorithm? Or should the algorithm derive a new type from the two arguments? The type-traits library comes to the rescue. We will present two variations.

The Smaller Type

A good choice for the return type is to use the smaller of both types. Therefore, we need a ternary operator at compile-time. Thanks to the type-traits library, we have it. The ternary function `std::conditional` operates on types and not on values. That's because we apply the function at compile-time.

So, we have to feed `std::conditional` with the right constant expression and we are done. `std::conditional<(sizeof(T1) < sizeof(T2)), T1, T2>::type` will return, at compile-time, `T1` if `T1` is smaller than `T2`; it will return `T2` if `T2` is not smaller than `T1`.

Let's apply the logic.

```
#include <iostream>
#include <type_traits>
#include <typeinfo>
template<typename T1, typename T2>
typename std::conditional<(sizeof(T1)<sizeof(T2)),T1,T2>::type gcd(T1 a, T2 b){
  static_assert(std::is_integral<T1>::value, "T1 should be integral!");
  static_assert(std::is_integral<T2>::value, "T2 should be integral!");
  if( b == 0 )
    return a;
  else
    return gcd(b, a % b);
}

int main(){
  std::cout << gcd(100,10LL) << std::endl;
  auto res= gcd(100,10LL);
  std::conditional<(sizeof(long long)<sizeof(long)), long long, long>::type res2=gcd(100LL,1
  std::cout << typeid(res).name() << std::endl;  // i
  std::cout << typeid(res2).name() << std::endl; // l
  std::cout << std::endl;
}
```

The critical line of the program is in line 5 with the return type of the gcd algorithm. Of course, the algorithm can also deal with template arguments of the same type. What about line 15? We used the number 100 of type int and the number 10 of type `long long int`. The result for the greatest common divisor is 10. Line 17 is extremely ugly. We have to repeat the expression `std::conditional <(sizeof(100) < sizeof(10LL)), long long, long>::type` to determine the right type of the variable `res2`. Automatic type deduction with `auto` comes to my rescue (line 16). The `typeid` operator in line 18 and 19 shows that the result type of the arguments of type `int` and `long long int` is `int`; that the result type of the types `long long int` and `long int` is `long int`.

The Common Type

Now to the second variation. Often it is not necessary to determine the smaller type at compile-time but to determine the type to which all types can implicitly be converted to. `std::common_type` can handle an arbitrary number of template arguments. To say it more formally. `std::common_type` is a variadic template.

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

template<typename T1, typename T2>
typename std::common_type<T1, T2>::type gcd(T1 a, T2 b){
  static_assert(std::is_integral<T1>::value, "T1 should be an integral type!");
  static_assert(std::is_integral<T2>::value, "T2 should be an integral type!");
  if( b == 0 ){
    return a;
  }
  else{
    return gcd(b, a % b);
  }
}
int main(){
  std::cout << typeid(gcd(100, 10)).name() << std::endl;  // i
  std::cout << typeid(gcd(100, 10L)).name() << std::endl;  // l
  std::cout << typeid(gcd(100, 10LL)).name() << std::endl;  // x
 }
```

The only difference to the last implementation is that `std::common_type` in line 6 determines the return type. We ignored the results of the `gcd` algorithm in this example because we're more interested in the types of results. With the argument types `int` and `int` we get `int`; with the argument types `int` and `long int` we get `long int`, and with `int` and `long long int` we get `long long int`.

gcd - The Fourth

But that's not all. `std::enable_if` from the type-traits library provides a very interesting variation. What the previous implementations have in common is that they will check in the function body if the arguments are of integral types or not. The key observation is that the compiler always tries to instantiate the function templates but sometimes fails. We know the result. If the expression `std::is_integral` returns `false`, the instantiation will fail. That is not the best way. It would be better if the function template is only available for the valid types. Therefore, we put the check of the function template from the template body to the template signature.

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

template<typename T1, typename T2,
  typename std::enable_if<std::is_integral<T1>::value,T1 >::type= 0,
  typename std::enable_if<std::is_integral<T2>::value,T2 >::type= 0,
  typename R = typename std::conditional<(sizeof(T1) < sizeof(T2)),T1,T2>::type>
R gcd(T1 a, T2 b){
  if( b == 0 ){
    return a;
  }
  else{
    return gcd(b, a % b);
  }
}

int main(){
  std::cout << "gcd(100, 10)= " << gcd(100, 10) << std::endl;
  std::cout << "gcd(100, 33)= " << gcd(100, 33) << std::endl;
  std::cout << "gcd(3.5, 4.0)= " << gcd(3.5, 4.0) << std::endl;
}
```

Lines 5 and 6 are the key lines of the new program. The expression `std::is_integral` determines whether the type parameter `T1` and `T2` are integrals. If `T1` and `T2` are not integrals, and therefore they return `false`, we will not get a template instantiation. This is the decisive observation.

If `std::enable_if` returns `true` as the first parameter, `std::enable_if` will have a public member `typedef` type. This `type` is used in lines 5 and 6. If `std::enable_if` returns `false` as first parameter, `std::enable_if` will have no member `type`. Therefore, lines 5 and 6 are not valid. This is not an error but a common technique in C++: SFINAE. SFINAE stands for **S**ubstitution **F**ailure **I**s **N**ot **A**n **E**rror. Only the template for exactly this type will not be instantiated and the compiler tries to instantiate the template in another way.

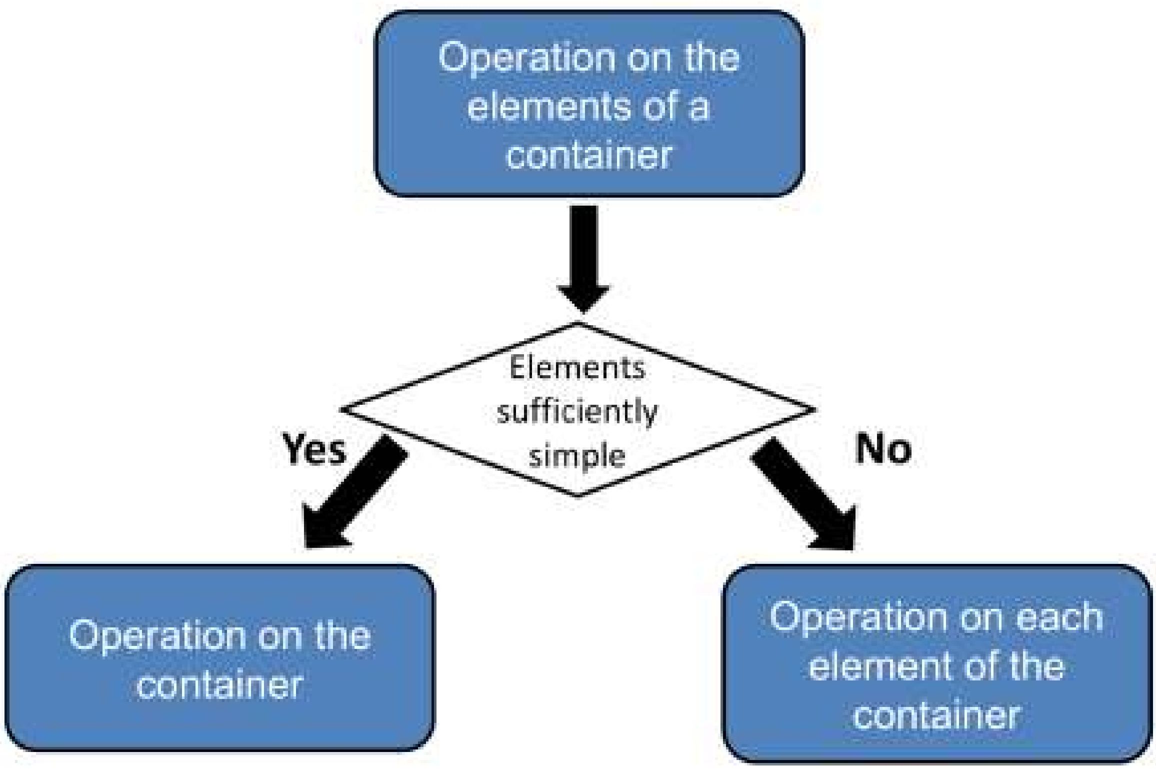
Type-Traits: Performance

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 like `std::copy`, `std::fill`, or `std::equal` will directly be applied on the memory area. Instead of using `std::copy` to copy the elements one by one, all is done in one step. Internally, C functions like `memcmp`, `memset`, `memcpy`, or `memmove` are used. The small difference

between `memcpy` and `memmove` is that `memmove` can deal with overlapping memory areas.

The implementations of the algorithm `std::copy`, `std::fill`, or `std::equal` use a simple strategy. `std::copy` is like a wrapper. This wrapper checks if the element is simple enough. If so, the wrapper will delegate the work to the optimized copy function. If not, the general copy algorithm will be used. This one copies each element after one another. To make the right decision, the functions of the type-traits library will be used if the elements are simple enough.

The graphic shows this strategy once more:



Type-Traits: Optimization

Type-Traits `fill`

`std::fill` assigns each element, in the range, a value. The listing shows a simple implementation which is based on the GCC implementation.

```
// 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;

}
```

`my::fill` make in line 27 the decision 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 should be 1 byte large: `sizeof(T) == 1`. The function `std::is_trivially_copy_assignable` is part of the type-traits library. The first call `my::fill(charArray1, charArray1 + arraySize,1);` has the

last parameter 1, which is an `int` that has a size of 4 bytes. That is why, the test on line 26 evaluates to false.

Our GCC calls the function `std::is_trivially_copy_assignable` instead of `std::has_trivial_copy_assign`. If we request with the keyword `default` from the compiler the copy assignment operator, the operator will be trivial.

Type-Traits `std::equal`

The following code snippet shows a part of the implementation of `std::equal` in the GCC:

```
template<typename _II1, typename _II2>
inline bool __equal_aux(_II1 __first1, _II1 __last1, _II2 __first2){
  typedef typename iterator_traits<_II1>::value_type _ValueType1;
  typedef typename iterator_traits<_II2>::value_type _ValueType2;
  const bool __simple = ((__is_integer<_ValueType1>::__value
                         || __is_pointer<_ValueType1>::__value )
                         && __is_pointer<_II1>::__value
                         && __is_pointer<_II2>::__value
                         &&__are_same<_ValueType1, _ValueType2>::__value
                        );
return std::__equal<__simple>::equal(__first1, __last1, __first2);
}
```

We have a different perception of `__simple`. To use the optimized variant of `std::equal`, the container elements have to fulfill some assurances. The elements of the container have to be of the same type (line 9) and have to be an integral or a pointer (lines 5 and 6). In addition, the iterators have to be pointers (lines 7 and 8).

To learn more about type-traits, click here.

In the next lesson, we'll look at a couple of examples of type traits.