- Examples

We'll discuss the examples for application of type traits in this lesson.

WE'LL COVER THE FOLLOWING ^ Example 1 Explanation Example 2 Explanation Example 3 Explanation

Example 1

Here is the application of the primary type categories:

```
// typeTraitsTypeCategories.cpp
#include <iostream>
#include <type_traits>
using namespace std;
int main(){
  cout << endl;</pre>
  cout << boolalpha;</pre>
  cout << "is_void<void>::value: " << is_void<void>::value << endl;</pre>
  cout << "is_integral<short>::value: " << is_integral<short>::value << endl;</pre>
  cout << "is_floating_point<double>::value: " << is_floating_point<double>::value << endl;</pre>
  cout << "is_array<int []>::value: " << is_array<int [] >::value << endl;</pre>
  cout << "is_pointer<int*>::value: " << is_pointer<int*>::value << endl;</pre>
  cout << "is_null_pointer<std::nullptr_t>::value: " << is_null_pointer<std::nullptr_t>::value
  struct A{
    int f(double){return 2011;}
  };
  cout << "is_member_object_pointer<int A::*>::value: " << is_member_object_pointer<int A::*>
  cout << "is_member_function_pointer<int (A::*)(double)>::value: " << is_member_function_poi</pre>
  enum E{
```

```
};
cout << "is_enum<E>::value: " << is_enum<E>::value << endl;
union U{
   int u;
};
cout << "is_union<U>::value: " << is_union<U>::value << endl;
cout << "is_class<string>::value: " << is_class<string>::value << endl;
cout << "is_function<int * (double)>::value: " << is_function<int * (double)>::value << endl;
cout << "is_lvalue_reference<int&>::value: " << is_lvalue_reference<int&>::value << endl;
cout << "is_rvalue_reference<int&&>::value: " << is_rvalue_reference<int&&>::value << endl;
cout << endl;
}
</pre>
```

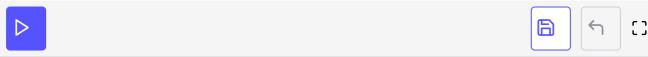
Explanation

Due to the flag boolalpha in line 10, the program displays either true or false instead of 1 or 0. Each call of the 14 primary type categories returns true.

Example 2

```
// typeTraitsCopy.cpp
                                                                                           G
#include <string.h>
#include <iostream>
#include <type_traits>
namespace my{
  template<typename I1, typename I2, bool b>
  I2 copy_imp(I1 first, I1 last, I2 out, const std::integral_constant<bool, b>&){
    while(first != last){
      *out = *first;
      ++out;
      ++first;
    std::cout << "elementwise." << std::endl;</pre>
    return out;
  }
  template<typename T>
  T* copy_imp(const T* first, const T* last, T* out, const std::true_type&){
    memcpy(out, first, (last-first)*sizeof(T));
    std::cout << "bitwise." << std::endl;</pre>
    return out+(last-first);
```

```
}
  template<typename I1, typename I2>
  I2 copy(I1 first, I1 last, I2 out){
    typedef typename std::iterator_traits<I1>::value_type value_type;
    return copy_imp(first, last, out, std::is_trivially_copy_assignable<value_type>());
const int arraySize = 1000;
// intialize all elements to 0
int intArray[arraySize] = {0, };
int intArray2[arraySize]={0, };
int* pArray = intArray;
const int* pArray2 = intArray2;
int main(){
  std::cout << std::endl;</pre>
  std::cout << "Copying pArray";</pre>
  my::copy(pArray2, pArray2 + arraySize, pArray);
  std::cout << "\n" << "Copying intArray ";</pre>
  my::copy(intArray2, intArray2 + arraySize, intArray);
```



Explanation

- my::copy, in line 36, makes the decision of which implementation of
 my::copy_imp is applied.
- In lines 13-16, we use a simple while loop to iterate through the array and copy elements.
- In line 27, we use std::memcpy to copy all the elements bitwise from the first to last and store it in out.

Example 3

```
#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 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 << std::endl;</pre>
  std::cout << "gcd(100,10)= " << gcd(100,10) << std::endl;
  std::cout << "gcd(100,33)= " << gcd(100,33) << std::endl;
  std::cout << "gcd(100,0)= " << gcd(100,0) << std::endl;
  std::cout << std::endl;</pre>
  std::cout << "gcd(100,10LL)= " << gcd(100,10LL) << std::endl;
  std::conditional <(sizeof(100) < sizeof(10LL)), long long, long>::type uglyRes= gcd(100,10L
  auto res= gcd(100,10LL);
  auto res2= gcd(100LL,10L);
  std::cout << "typeid(gcd(100,10LL)).name(): " << typeid(res).name() << std::endl;</pre>
  std::cout << "typeid(gcd(100LL,10L)).name(): " << typeid(res2).name() << std::endl;</pre>
  std::cout << std::endl;</pre>
```







נכ

Explanation

- The key line of the program is line 8 with the return type of the gcd algorithm. The algorithm can also handle template arguments of the same type. We can see this process both in lines 21 24 and in the output of the program.
- What about line 27?
- We use the number 100 of type int and the number 10 of type long long int. The result for the greatest common divisor is 10. We must repeat the expression std::conditional <(sizeof(100) < sizeof(10LL)), long long, long>::type to determine the right type of the variable uglyRes.

- Automatic type deduction with auto resolves this problem (line 30 and 31).
- The typeid operator in line 33 and 34 shows us two things. Firstly, the result type of the arguments of type int and long long int is int. Secondly, that the result type of the types long long int and long int is long int.

Let's test our understanding of this topic with an exercise in the next lesson.