Check Type Information

Type traits allow us access and verify the type categories for all our variables. We'll write some code to do that.

WE'LL COVER THE FOLLOWING

With the type traits library, you can check primary and composite type categories. The attribute `value` gives you the result.

Primary Type Categories

There are 14 different type categories. They are complete and don't overlap. So each type is only a member of one type category. If you check a type category for your type, the request is independent of the `const` or `volatile` qualifiers.

```
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;
```

The following code samples show all primary type categories.

```
// typeCategories.cpp
```

```
#include <iostream>
#include <type_traits>
using std::cout;

int main()
{
  //out put 1 means that the function returns true
  cout << "is_void: " << std::is_void<void>::value << "\n";             // 1
  cout << "is_integral: " << std::is_integral<short>::value << "\n";        // 1
  cout << "is_floating_point: " << std::is_floating_point<double>::value << "\n"; // 1
  cout << "is_array: " << std::is_array<int [] >::value << "\n";          // 1
  cout << "is_pointer: " << std::is_pointer<int*>::value << "\n";          // 1
  cout << "is_reference: " << std::is_reference<int&>::value << "\n";        // 1

  struct A{
    int a;
    int f(int){ return 2011; }
  };
  cout << "is_member_object_pointer: " << std::is_member_object_pointer<int A::*>::value << '
  cout << "is_member_function_pointer: " << std::is_member_function_pointer<int (A::*)(int)>:

  enum E{
    e= 1,
  };
  cout << "is_enum: " << std::is_enum<E>::value << "\n";                  // 1

  union U{
    int u;
  };
  cout << "is_union: " << std::is_union<U>::value << "\n";                 // 1

  cout << "is_class: " << std::is_class<std::string>::value << "\n";       // 1
  cout << "is_function: " << std::is_function<int * (double)>::value << "\n"; // 1
  cout << "is_lvalue_reference: " << std::is_lvalue_reference<int&>::value << "\n";   // 1
  cout << "is_rvalue_reference: " << std::is_rvalue_reference<int&&>::value << "\n";  // 1

  return 0;
}
```

All primary type categories

Composite Type Categories

Based on the 14 primary type categories, there are 6 composite type categories.

Composite type categories	Primary type category
`is_arithmetic`	`is_floating_point` or `is_integral`
`is_fundamental`	`is_arithmetic` or `is_void`

`is_object`	`is_arithmetic` or `is_enum` or `is_pointer` or `is_member_pointer`
`is_reference`	`is_lvalue_reference` or `is_rvalue_reference`
`is_compound`	complement of `is_fundamental`
`is_member_pointer`	`is_member_object_pointer` or `is_member_function_pointer`

Composite type categories

Type Properties

In addition to the primary and composite type categories, there are type properties.

```
template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_trivial;
template <class T> struct is_trivially_copyable;
template <class T> struct is_standard_layout;
template <class T> struct is_pod;
template <class T> struct is_literal_type;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;

template <class T> struct is_signed;
template <class T> struct is_unsigned;

template <class T, class... Args> struct is_constructible;
template <class T> struct is_default_constructible;
template <class T> struct is_copy_constructible;
template <class T> struct is_move_constructible;

template <class T, class U> struct is_assignable;
template <class T> struct is_copy_assignable;
template <class T> struct is_move_assignable;
template <class T> struct is_destructible;
template <class T, class... Args> struct is_trivially_constructible;
template <class T> struct is_trivially_default_constructible;
template <class T> struct is_trivially_copy_constructible;
template <class T> struct is_trivially_move_constructible;
template <class T, class U> struct is_trivially_assignable;
template <class T> struct is_trivially_copy_assignable;
```

```
template <class T> struct is_trivially_copy_assignable;
template <class T> struct is_trivially_move_assignable;

template <class T> struct is_trivially_destructible;

template <class T, class... Args> struct is_nothrow_constructible;
template <class T> struct is_nothrow_default_constructible;
template <class T> struct is_nothrow_copy_constructible;
template <class T> struct is_nothrow_move_constructible;

template <class T, class U> struct is_nothrow_assignable;
template <class T> struct is_nothrow_copy_assignable;
template <class T> struct is_nothrow_move_assignable;

template <class T> struct is_nothrow_destructible;
template <class T> struct has_virtual_destructor;
```

Now let's talk about type comparisons and modifying these types in compile-time.