# 2023.10.25

### Template Value Type

```
template <typename T, T v>
struct IntegralConstant
{
    static constexpr T value = v; // sabit
    using value_type = T; // bir tür
    using type = IntegralConstant; // oluşturalan struct kendisi
    constexpr operator value_type()const noexcept {return value;} // tür dönüşüm
operator
    constexpr value_type operator()()const noexcept {return value;}
};

using TrueType = IntegralConstant<bool, true> // true

int main()
{
    IntegralConstant<int, 5>::value; // sabit
    IntegralConstant<int, 5>::value_type; // bir tür
    IntegralConstant<int, 4>::type  // IntegralConstant<int, 4>

    constexpr auto val = IntegralConstant<int, 4>{} + IntegralConstant<int, 3>:{};
// 7
    constexpr auto val = IntegralConstant<int, 5>{}(); // val = 5
}
```

```
template <typename T>
void func(T x)
{
    static_assert(sizeof(T) > 2, "sizeof value must be greater than 2");
}

int main()
{
    func('A');
}
```

```
template<typename T>
struct IsPointer : FalseType {};

template<typename T>
struct IsPointer<T*> : TrueType {};

template <typename T>
void func(T x)
{
    static_assert(IsPointer<T>::value, "only for pointer type");
}

int main()
{

}
```

```cpp
#include <type_traits>

template<typename T>
void func(T)
{
    static_assert(std::is_pointer_v<T>);
    // pointer olmayan türden arguman gönderirsek syntax hatası verir
}

int main()
{

}
```

```cpp
template<typename T>
struct RemoveReference
{
    using type = T;
};

template<typename T>
struct RemoveReference<T&>
{
    using type = T;
};

template<typename T>
struct RemoveReference<T&&>
{
    using type = T;
};

template<typename T>
using RemoveReference_t = typename RemoveReference<T>::type;

int main()
{
    RemoveReference<int>::type // int
    RemoveReference<int&>::type // int
    RemoveReference<int&& >::type // int
}
```

**Tag-Dispatch**

```cpp
// r value gelince ayrı bir kod l value gelince ayrı kod

// tag-dispatch
#include <type_traits>
#include <iostream>

void bar(std::true_type)
{
    std::cout << "implementaion for l values\n";
}

void bar(std::false_type)
{
    std::cout << "implementaion for r values\n";
}


template<typename T>
void func(T&&)
{
    bar(std::is_lvalue_reference<T>{});
}

int main()
{
    func(12); // implementaion for r values

    int x{ 21 };

    func(x); // implementaion for l values

}
```

```cpp
// tam sayı türleri için ayrı implementasyon olmayanlar için ayrı

template<typename T>
void func_impl(T, std::true_type)
{
    std::cout << "tam sayi türleri için\n";
}
template<typename T>
void func_impl(T, std::false_type)
{
    std::cout << "tam sayi olmayan türler icin\n";
}
template <typename T>
void func(T x)
{
    func_impl(x, std::is_integeral<T>{});
}

int main()
{
    func(12); // tam sayi türleri için
    func(12.2); // tam sayi olmayan türler icin
}
```

### Static If (cpp 17)

```cpp
template <typename T>
auto get_value(T x)
{
    if constexpr(std::is_pointer_v<T>)
    {
        return  *x;
    }
    else
    {
        return x;
    }
}

int main()
{
    int x = 5;
    double dval = 4.971;

    int* ip {&x};
    double* dp{&dval};

    std::cout << get_value(x) << "\n";
    std::cout << get_value(dval) << "\n";
    std::cout << get_value(ip) << "\n";
    std::cout << get_value(dp) << "\n";
}
```

### Standart Template Library

### Iterators

```cpp
template<typename Iter>
void print_array(Iter beg, Iter end)
{
    while(beg != end)
    {
        std::cout << *beg << '  ';
        ++beg;
    }

    std::cout << "\n"
}

int main()
{
    int a[5] = {1 , 3 , 4, 5};
    print_array(a ,a + 5);

    std::vector<double> dvec{1.2, 4.5, 7.3, 9.3, 2.45};

    print_array(dvec.begin(), dvec.end());

    std::list<std::string> names{"melike", "tamer", "serhat", "burak"};
    print_array(names.begin(), names.end());
}
```

```
    template<typename T, typename A>
    class Vector
    {
        class iterator
        {
            public:
            T& operator*();
            bool operator!=(const Iterator&)const;
            operator++();
            operator++(int);
        };

        iterator begin();
        iterator end();
    }
```

```
int main()
{
    vector<int> ivec(1000);

    auto bg = ivec.begin();
    auto iter = begin(ivec); // global

}
```

**Iterator Category**

iterator category:

- input_iterator
- output_iterator
- forward_iterator
- bidirectional_iterator
- random_access_iterator

STL Container'ların iterator category'si belirlidir.

```
template<typename Iter>
void func(Iter beg, Iter end)
{
    // compile time
}

//std::output_iterator_tag
//std::input_iterator_tag
//std::forward_iterator_tag
//std::bidirectional_iterator_tag
//std::random_access_iterator_tag


int main()
{
    vector<string>::iterator::iterator_category // iterator_category söyler
}
```

```cpp
template<typename T, typename U>
struct IsSame : std::false_type {};

template<typename T>
struct IsSame<T, T> : std::true_type {};

template<typename T, typename U>
constexpr bool IsSame_v = IsSame<T, U>::value;

int main()
{
    IsSame_v<int, double> // false
    IsSame_v<int, int> // true
}
```

```cpp
template<typename Iter>
void algo(Iter beg, Iter end)
{
    if constexpr(std::is_same_v<Iter::iterator_category,
std::random_access_iterator_tag>)
    {
        ///
    }
}
```