

```

// namespace integer_sequence_internals
template<typename T, T N>
using make_integer_sequence = typename integer_sequence_internals::make_integer_sequence<T, N>::type;

```

```

template<class T1, class T2 = T1>
inline T1 exchange(T1 & obj, T2 && new_value)
{
    T1 old_value = std::move(obj);
    obj = std::forward<T2>(new_value);
    return old_value;
}

struct increment_me
{
    template<typename T>
    auto operator()(T t) const
        -> decltype(++t)
    {
        return ++t;
    }
};

template<typename T=double>
constexpr T pi = T(3.1415926535897932385);
auto area = [](auto c)
{
    using T = typename decltype(c).value_type;
    return pi<T> * c.radius * c.radius;
};

template<typename Array, size_t... I>
auto array_to_tuple_(const Array& a, std::index_sequence<I...>)
-> decltype(std::make_tuple(a[I]...))
{
    return std::make_tuple(a[I]...);
}

template<typename T, std::size_t N, typename Indices = std::make_index_sequence<N>>
auto array_to_tuple_(const std::array<T, N>& a)
-> decltype(array_to_tuple_(a, Indices()))
{
    return array_to_tuple_(a, Indices());
}

template<class T>
struct C
{
    T t_;
    template<class U, class V = typename std::enable_if
        <
        !std::is_lvalue_reference<U>::value
        >::type>
    C(U&& u) : t_(std::forward<T>(std::move(u).get())) {}
};

int x;
struct A
{
    // constexpr constructor
    constexpr A(bool b) : m(b?42:x) {}
    int m;
};

constexpr int v = A(true).m;
constexpr int w = A(false).m;

#include <utility>
#include <type_traits>
// generates std::size_t : 0, 1, 2, 3
using indices =
    std::index_sequence_for<char, int, std::size_t, unsigned long>;
static_assert(std::is_same<
    indices,
    std::make_index_sequence<4>
>::value, "");

struct A{};
struct B{};
static_assert(std::is_same<
    std::index_sequence_for<A, B>,
    std::index_sequence<0, 1>
>::value, "");

template<class T
using void_t = void;
//using void_t = std::enable_if_t<true>;>;
template<class, class T = void>
struct has_type_member : std::false_type
{
};

template<class T>
struct has_type_member<T, void_t<typename T::type>> : std::true_type
{
};

template<typename Array, size_t... I>
auto array_to_tuple_(const Array& a, std::index_sequence<I...>)
-> decltype(std::make_tuple(a[I]...))
{
    return std::make_tuple(a[I]...);
}

template<typename T, std::size_t N, typename Indices = std::make_index_sequence<N>>
auto array_to_tuple_(const std::array<T, N>& a)
-> decltype(array_to_tuple_(a, Indices()))

```

C++14 FAQs

Chandra Shekhar Kumar