Филипп Хандельянц

Лекция 9/12

Метапрограммирование в C++

Докладчик

Хандельянц

Филипп Александрович

- Ведущий разработчик в команде PVS-Studio (C++/C#)
- 3 года участвую в разработке ядра C++ анализатора
- Автор статей о проверке open source-проектов

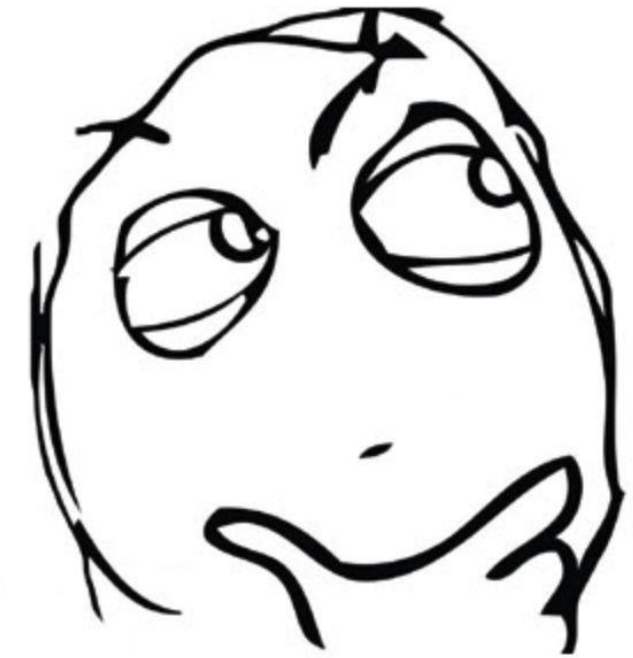
Metaprogramming

Метапрограммирование — вид программирования, связанный с созданием программ, которые порождают другие программы как результат своей работы.

© Википедия

Метапрограммирование — вид программирования, связанный с созданием программ, которые порождают другие программы как результат своей работы.

© Википедия



```
int max(int lhs, int rhs)
{
  return lhs > rhs ? lhs : rhs;
}
```

```
int max(int lhs, int rhs)
{
  return lhs > rhs ? lhs : rhs;
}

double max(double lhs, double rhs)
{
  return lhs > rhs ? lhs : rhs;
}
```

```
int max(int lhs, int rhs)
{
  return lhs > rhs ? lhs : rhs;
}

double max(double lhs, double rhs)
{
  return lhs > rhs ? lhs : rhs;
}

const SomeClass& max(const SomeClass &lhs, const SomeClass &rhs)
{
  return lhs > rhs ? lhs : rhs;
}
```

```
#define MAX(A, B) \
  ((A) > (B)) ? (A) : (B)

MAX(2, 5); // => ((2) > (5)) ? (2) : (5);
```

```
#define MAX(A, B) \
  ((A) > (B)) ? (A) : (B)

#define MAX(Type, A, B) \
  Type##MAX((A), (B))

#define DEF_MAX(Type) \
  Type Type##MAX(Type lhs, Type rhs) \
  { \
    return lhs > rhs ? lhs : rhs; \
  }

DEF_MAX(int); // => int intmax(int lhs, int rhs) { return .... }

MAX(int, 2, 5) // => intmax(2, 5);
```

Templates

```
template <typename T>
const T& max(const T &lhs, const T &rhs)
{
  return lhs > rhs ? lhs : rhs;
}

max(2, 5);       // => max<int>(2, 5);

max(2.0, 5.0)    // => max<double>(2.0, 5.0);

class SomeClass { .... };

SomeClass a, b;
max(a, b)        // => max<SomeClass>(a, b);
```

```
template <typename T, size_t N>
struct array
{
  T arr[N];

  T& operator[](size_t idx) noexcept { return arr[idx]; }
  const T& operator[](size_t idx) noexcept const { return arr[idx]; }
  ....
};

array<int, 3> arr { 0, 1, 2 };

// struct array_i_3
// {
//   int arr[3];
//
//   T& operator[](size_t idx) noexcept { return arr[idx]; }
//   const T& operator[](size_t idx) noexcept const { return arr[idx]; }
//   ....
// };
```

Non-type template parameters

- type name(optional)

```
template <size_t> // or template <size_t N>
struct int_array { .... };
```

- type name(optional) = default

```
template <size_t = 42> // or template <size_t N = 42>
struct array { .... };
```

- type ...name(optional) (since C++11)

```
template <size_t ...> // or template <size_t ...Ints>
class sizeT_sequence { .... };
```

- placeholder name (since C++17)

```
template <auto V> // or template <decltype(auto) V>
struct B { .... }
```

Non-type template parameters

- lvalue reference type
- std::nullptr_t (since C++11)
- integral type (bool, char, signed char, unsigned char, short, ...)
- pointer type
- pointer to member type
- enumeration type

Type template parameters

- type-parameter-key name$_{(optional)}$

```
template <class> // or template <typename T>
class FalseVector { .... };
```

- type-parameter-key name$_{(optional)}$ = default

```
template <class T, class Alloc = std::allocator<T>>
class TrueVector { .... };
```

- type-parameter-key ...name$_{(optional)}$ (**since C++11**)

```
template <class ...> // or template <typename ...Types>
class tuple { .... };
```

Template template parameters

- **template** ‹parameter-list› **typename** $_{(C++17)}$ | **class** name$_{(optional)}$

```
template <class K, class T, template <class> class Container>
class MyMap
{
  Container<K> keys;
  Container<T> values;
}
```

- **template** ‹parameter-list› **typename** $_{(C++17)}$ | **class** name$_{(optional)}$ = default

```
template <class T> class my_array { .... };

template <class K, class T, template <class> class Container = my_array>
class MyMap { .... }
```

- **template** ‹parameter-list› **typename** $_{(C++17)}$ | **class** ...name$_{(optional)}$

```
template <class K, class T, template <class, class> class ...Map>
class MyMap : Map<K, T>... { .... };
```

typename

```
template <typename T>
struct X : B<T> // "B<T>" is dependent on T
{
  typename T::A *pa; // "T::A" is dependent name from T

  void f(B<T> *pb)
  {
    static int i = B<T>::i; // "B<T>::i" is dependent variable on T
    pb->j++;                // "pb->j" is dependent variable from T
  }
};
```

Explicit (full) specialization

```
template <class T>
class vector        // class template
{
  ....
};

template <>
class vector<bool> // full specialization for vector<bool>
{
  ....
};
```

Explicit (full) specialization

```
template <class T>
void print(const T &obj) // function template
{
  std::cout << obj << '\n';
}

class SomeClass { .... };

template <>
void print<SomeClass>(const SomeClass &obj) // full specialization for print
{
  ....
}
```

Partial specialization

```
// class template
template <class T, class Deleter>
class unique_ptr
{
  ....
public:
  T* operator->() const noexcept;
  ....
};

// partial specialization for class template
template <class T, class Deleter>
class unique_ptr<T[], Deleter>
{
  ....
public:
  T& operator[](size_t idx) noexcept;
  const T& operator[](size_t idx) const noexcept;
  ....
};
```

Variadic template

```
template <class T1, class T2>
bool EqualsAnyOf(const T1 &t1, const T2 &t2)
{
  return t1 == t2;
}
```

Variadic template

```
template <class T1, class T2>
bool EqualsAnyOf(const T1 &t1, const T2 &t2)
{
  return t1 == t2;
}

template <class T1, class T2, class T3>
bool EqualsAnyOf(const T1 &t1, const T2 &t2, const T3 &t3)
{
  return t1 == t2 || t1 == t3;
}
```

Variadic template

```
template <class T1, class T2>
bool EqualsAnyOf(const T1 &t1, const T2 &t2)
{
  return t1 == t2;
}

template <class T1, class T2, class T3>
bool EqualsAnyOf(const T1 &t1, const T2 &t2, const T3 &t3)
{
  return t1 == t2 || t1 == t3;
}

template <class T1, class T2, class T3, class T4>
bool EqualsAnyOf(const T1 &t1, const T2 &t2, const T3 &t3, const T4 &t4)
{
  return t1 == t2 || t1 == t3 || t1 == t4;
}
```

Variadic template

```
template <class T1, class ...TN>
bool EqualsAnyOf(const T1 &t1, const TN &...tN);
```

Variadic template

```
template <class T1>
bool EqualsAnyOf(const T1 &t1) noexcept // end of the recursion
{
  return false;
}

template <class T1, class T2, class ...TN>
bool EqualsAnyOf(const T1 &t1, const T2 &t2, const TN &...tN) noexcept
{
  return t1 == t2 || EqualsAnyOf(t1, tN...);
}

void foo()
{
  EqualsAnyOf(0, 'a', 0.0, 42);
}
```

Variadic template

```
// EqualsAnyOf(0, 'a', 0.0, 42) ≡ return 0 == 'a' || Equals(0, 42, 0.0);
// EqualsAnyOf(0, 42, 0.0)      ≡ return 0 ==  42 || Equals(0, 0.0);
// EqualsAnyOf(0, 0.0)          ≡ return 0 == 0.0 || Equals(0);
// EqualsAnyOf(0)               ≡ return false;

// EqualsAnyOf(0, 'a', 0.0, 42) ≡ return 0 == 'a'
                                       || 0 == 42
                                       || 0 == 0.0
                                       || false;
```

Variadic template with fold expression

```
template <class T1, class T2, class ...TN>
bool EqualsAnyOf(const T1 &t1, const T2 &t2, const TN &...tN) noexcept
{
  return ((t1 == t2) || ... || (t1 == tN)); // fold expression since C++17

  // return ( ( ( ( (t1 == t2) || (t1 == t3) ) || ...) || t1 == tn) );
}

void foo()
{
  EqualsAnyOf(0, 'a', 0.0, 42);
}
```

Variadic template

```
template <class T, class Alloc = std::allocator<T>>
class vector
{
  ....
public:
  template <class ...Args>
  T& emplace_back(Args &&...args)
  {
    T *ptr = ....; // address of new object
    new (ptr) T { std::forward<Args>(args)... };

    // new (ptr) T { std::forward<Arg1>(arg1), std::forward<Arg2>(arg2), ... };

    return *ptr;
  }
  ....
};
```

Compile-time computations

```
template <size_t N>
struct Factorial;
```

```
template <size_t N>
struct Factorial;

template <>
struct Factorial<0>
{
  enum { value = 1 };
};

template <>
struct Factorial<1>
{
  enum { value = 1 };
};

template <>
struct Factorial<2>
{
  enum { value = 2 };
};
```

```
template <size_t N>
struct Factorial
{
  enum { value = N * Factorial<N – 1>::value };
};

void foo()
{
  const auto fac5 = Factorial<5>::value;
}
```

```
<source>:9:22: fatal error: recursive template instantiation exceeded
maximum depth of 1024

  enum { value = N * Factorial<N - 1>::value };
                     ^
<source>:9:22: note: in instantiation of template class
'Factorial<18446744073709550597>' requested here

<source>:9:22: note: in instantiation of template class
'Factorial<18446744073709550598>' requested here

<source>:9:22: note: in instantiation of template class
'Factorial<18446744073709550599>' requested here

<source>:9:22: note: in instantiation of template class
'Factorial<18446744073709550600>' requested here

<source>:9:22: note: in instantiation of template class
'Factorial<18446744073709550601>' requested here
```

```
<source>:9:22: fatal error: recursive template instantiation exceeded
maximum depth of 1024

  enum { value = N * Factorial<N - 1>::value };
                     ^
<source>:9:22: note: in instantiation of template class
'Factorial<18446744073709550597>' requested here

<source>:9:22: note: in instantiation of template class
'Factorial<18446744073709550598>' requested here

<source>:9:22: note: in instantiation of template class
'Factorial<18446744073709550599>' requested here

<source>:9:22: note: in instantiation of template class
'Factorial<18446744073709550600>' requested here

<source>:9:22: note: in instantiation of template class
'Factorial<18446744073709550601>' requested here
```

```
template <size_t N>
struct Factorial
{
  enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<0>
{
  enum { value = 1 };
};

template <>
struct Factorial<1>
{
  enum { value = 1 };
};

void foo()
{
  const size_t fac5 = Factorial<5>::value;
}
```

```
// Factorial<5> == 5 * Factorial<4>::value
// Factorial<4> == 4 * Factorial<3>::value
// Factorial<3> == 3 * Factorial<2>::value
// Factorial<2> == 2 * Factorial<1>::value
// Factorial<1> == 1

// Factorial<5> == 5 * (4 * (3 * (2 * 1))) == 120
```

```
template <size_t N>
struct Factorial
{
  static constexpr size_t value = N * Factorial<N - 1>::value;
};

template <>
struct Factorial<0>
{
 static constexpr size_t value = 1;
};

template <>
struct Factorial<1>
{
  static constexpr size_t value = 1;
};

void foo()
{
  const size_t fac5 = Factorial<5>::value;
}
```

```
constexpr size_t Factorial(size_t n) noexcept
{
  return n > 1 ? n * Factorial(n - 1) : 1; // since C++11
}

constexpr size_t Factorial(size_t n) noexcept
{
  size_t acc = 1;
  for (size_t i = 2; i <= n; ++i)
  {
    acc *= i;
  }

  return acc; // since C++14
}
```

```
template <size_t Exp>
struct pow1
{
  double operator()(double base) const noexcept
  {
    return base * pow1<Exp - 1>{}(base);
  }
};

template <>
struct pow1<0>
{
  double operator()(double base) const noexcept
  {
    return 1.0;
  }
};
```

```
template <size_t Exp>
struct pow2
{
  double operator()(double base) const noexcept
  {
    return (Exp % 2 != 0)
      ? base * pow2<(Exp - 1) / 2>{}(base) * pow2<(Exp - 1) / 2>{}(base)
      : pow2<Exp / 2>{}(base) * pow2<Exp / 2>{}(base);
  }
};

template <>
struct pow2<0>
{
  double operator()(double base) const noexcept
  {
    return 1.0;
  }
};
```

```
template <size_t Exp>
struct pow1
{
  double operator()(double base) const noexcept
  {
    if constexpr (Exp == 0) // since C++17
    {
      return 1.0;
    }
    else
    {
      return base * pow1<Exp - 1>{}(base);
    }
  }
};
```

```
template <size_t Exp>
struct pow2
{
  double operator()(double base) const noexcept
  {
    if constexpr (Exp == 0) // since C++17
    {
      return 1.0;
    }
    else if constexpr (Exp & 1 != 0) // since C++17
    {
      return base * pow2<(Exp - 1) / 2>{}(base)
                  * pow2<(Exp - 1) / 2>{}(base);
    }
    else
    {
      return pow2<Exp / 2>{}(base) * pow2<Exp / 2>{}(base);
    }
  }
};
```

Exp	MSVC (UCRT)	MSVC (pow1)	MSVC (pow2)	GCC (GlibC)	GCC (pow1)	GCC (pow2)	Clang (libc++)	Clang (pow1)	Clang (pow2)
10	53.486	13.67	14.5	21.9	21.42	21.02	21.45	21.51	21.56
20	52.64	13.79	15.53	21	21.8	20.75	21.52	21.51	21.42
50	52.99	14.1	20.68	21.09	20.79	21.21	21.43	21.41	21.47
70	52.46	13.76	22.61	20.94	20.68	20.68	21.38	21.41	21.18
100	52.61	101.37	27.144	21.02	20.96	20.73	23.32	21.24	21.28
150	52.49	150.87	33.42	20.91	20.69	21.59	21.87	25.34	21.79
200	54.04	198.45	41.92	20.93	20.69	20.7	21.42	26.14	21.32

```
base == 2.0

N == 10'000'000
```

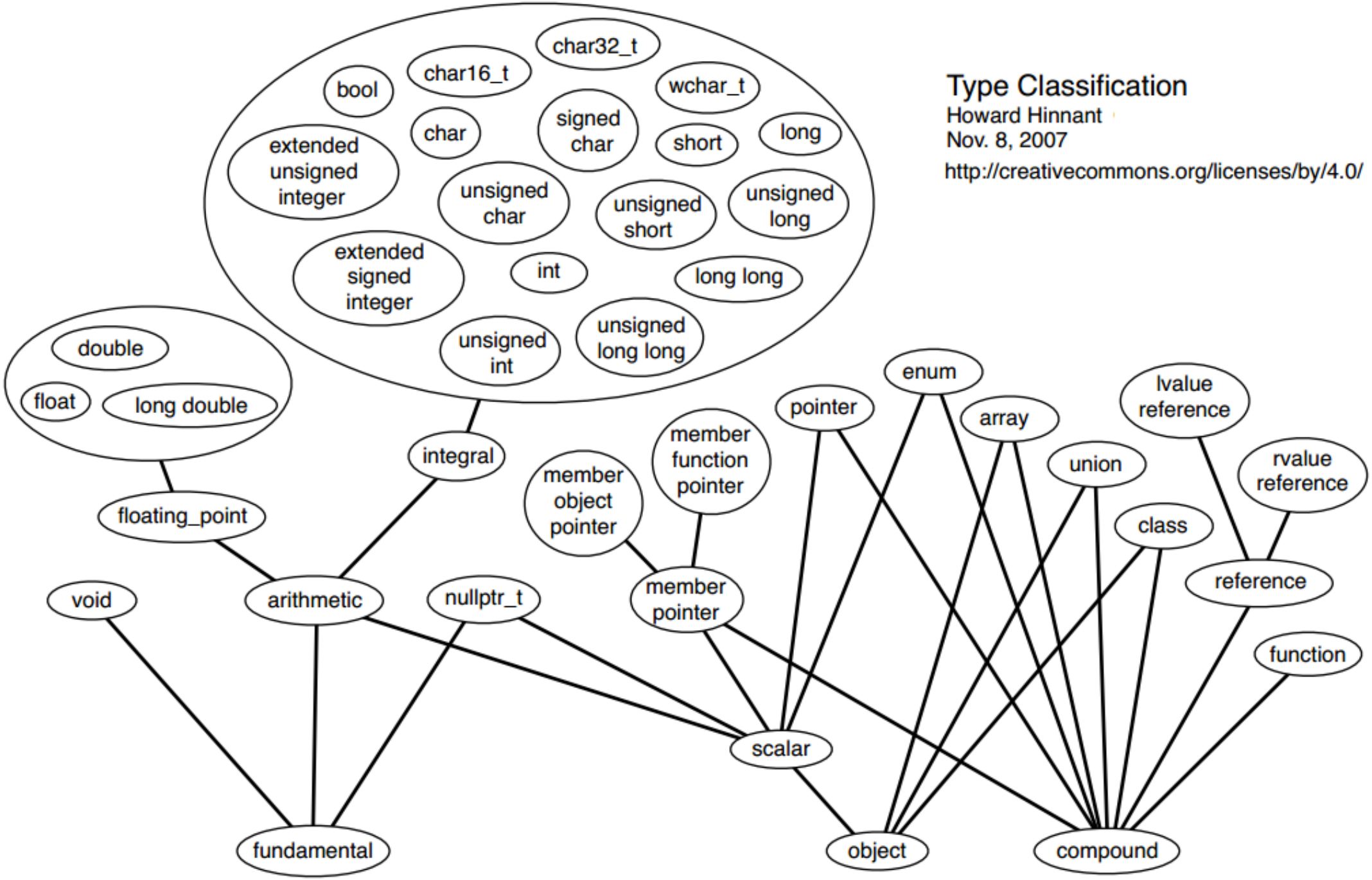
```
constexpr uint8_t popcount(uint64_t value) noexcept
{
  uint8_t res = 0;
  while (value != 0)
  {
    res += value & 1;
    value >>= 1;
  }

  return res;
}

void foo()
{
  constexpr uint8_t pop_count = popcount(0b10010010); // 3
}
```

```
constexpr uint64_t gcd(uint64_t a, uint64_t b) noexcept
{
  while (a != b)
  {
    if (a > b)
    {
      a -= b;
    }
    else
    {
      b -= a;
    }
  }

  return a;
}

void foo()
{
  constexpr auto gcd_a_b = gcd(15, 125); // 5
}
```

```
template <uint64_t N, size_t ...Idx>
constexpr std::array<bool, N + 1> eratosthenesSieve_impl(std::index_sequence<Idx...>) noexcept
{
  std::array<bool, N + 1> primes { (Idx, true)... };

  primes[0] = primes[1] = false;
  for (size_t i = 2; i * i <= N; ++i)
  {
    if (primes[i])
    {
      for (size_t j = i * i; j <= N; j += i)
      {
        primes[j] = false;
      }
    }
  }

  return primes;
}

template <uint64_t N, typename Idx = std::make_index_sequence<N + 1>>
constexpr std::array<bool, N + 1> eratosthenesSieve() noexcept
{
  return eratosthenesSieve_impl<N>(Idx {});
}
```

```
template <uint64_t N, typename Idx = std::make_index_sequence<N + 1>>
constexpr std::array<bool, N + 1> eratosthenesSieve() noexcept;

void foo()
{
  constexpr auto primes = eratosthenesSieve<5>();

  // primes[0] == false
  // primes[1] == false
  // primes[2] == true
  // primes[3] == true
  // primes[4] == false
  // primes[5] == true

  std::copy(primes.begin(),
            primes.end(),
            std::ostream_iterator<bool> { std::cout, " " });

  // prints: 0 0 1 1 0 1
}
```

Compile-time type manipulations


Type Classification
Howard Hinnant
Nov. 8, 2007
http://creativecommons.org/licenses/by/4.0/
char32_t
char16_t
bool
wchar_t
char
signed char
short
long
extended unsigned integer
unsigned char
unsigned short
unsigned long
extended signed integer
int
long long
unsigned int
unsigned long long
double
float
long double
integral
floating_point
member object pointer
member function pointer
pointer
enum
array
lvalue reference
union
rvalue reference
class
reference
void
arithmetic
nullptr_t
member pointer
function
scalar
fundamental
object
compound

```
template <class T>
struct is_reference
{
  static constexpr bool value = false;
};
```

```
template <class T>
struct is_reference
{
  static constexpr bool value = false;
};

template <class T>
struct is_reference<T&>
{
  static constexpr bool value = true;
};
```

```
template <class T>
struct is_reference
{
  static constexpr bool value = false;
};

template <class T>
struct is_reference<T&>
{
  static constexpr bool value = true;
};

template <class T>
struct is_reference<T&&>
{
  static constexpr bool value = true;
};
```

```cpp
template <class T>
struct is_reference
{
  static constexpr bool value = false;
};

template <class T>
struct is_reference<T&>
{
  static constexpr bool value = true;
};

template <class T>
struct is_reference<T&&>
{
  static constexpr bool value = true;
};

//   is_reference<int>::value == false
//  is_reference<int&>::value == true
// is_reference<int&&>::value == true
```

```
template <class T, T v>
struct integral_constant
{
  using value_type = T;
  using type = std::integral_constant<T, v>

  static constexpr value_type value = v;

  constexpr operator value_type() { return v; } const noexcept
  constexpr value_type operator()() { return v; } const noexcept
};

using  true_type = integral_constant<bool, true>;
using false_type = intergral_constant<bool, false>;
```

```
template <class T>
struct is_reference : false_type
{
  // static constexpr value_type value = false;
  // constexpr operator value_type() const noexcept { return value; }
  // constexpr operator()() const noexcept { return value; }
};

template <class T>
struct is_reference<T&> : true_type
{
  // static constexpr value_type value = true;
  // constexpr operator value_type() const noexcept { return value; }
  // constexpr operator()() const noexcept { return value; }
};

template <class T>
struct is_reference<T&&> : true_type
{
  // static constexpr value_type value = true;
  // constexpr operator value_type() const noexcept { return value; }
  // constexpr operator()() const noexcept { return value; }
};
```

```
template <class T>
struct is_reference : false_type
{
};

template <class T>
struct is_reference<T&> : true_type
{
};

template <class T>
struct is_reference<T&&> : true_type
{
};

template <class T>
inline constexpr bool is_reference_v = is_reference<T>::value;
```

Primary type categories

- is_void
- is_null_pointer
- is_integral
- is_floating_point
- is_array
- is_enum
- is_union
- is_class
- is_function
- is_pointer
- is_lvalue_reference
- is_rvalue_reference
- is_member_object_pointer
- is_member_function_pointer

Primary type categories

- is_void_v
- is_null_pointer_v
- is_integral_v
- is_floating_point_v
- is_array_v
- is_enum_v
- is_union_v
- is_class_v
- is_function_v
- is_pointer_v
- is_lvalue_reference_v
- is_rvalue_reference_v
- is_member_object_pointer_v
- is_member_function_pointer_v

Composite type categories

- is_fundamental
- is_arithmetic
- is_scalar
- is_object
- is_compound
- is_reference
- is_member_pointer

Composite type categories

- is_fundamental_v
- is_arithmetic_v
- is_scalar_v
- is_object_v
- is_compound_v
- is_reference_v
- is_member_pointer_v

Type properties

- is_const
- is_volatile
- is_trivial
- is_trivial_copyable
- is_standard_layout
- is_pod
- is_literal_type
- has_unique_object_representations
- is_empty
- is_polymorphic
- is_abstract
- is_final
- is_aggregate
- is_signed
- is_unsigned
- is_bounded_array
- is_unbounded_array

Type properties

- is_const_v
- is_volatile_v
- is_trivial_v
- is_trivial_copyable_v
- is_standard_layout_v
- is_pod_v
- is_literal_type_v
- has_unique_object_representations_v
- is_empty_v
- is_polymorphic_v
- is_abstract_v
- is_final_v
- is_aggregate_v
- is_signed_v
- is_unsigned_v
- is_bounded_array_v
- is_unbounded_array_v

Supported operations

- is_constructible
- is_trivially_constructible
- is_nothrow_constructible
- is_default_constructible
- is_trivially_default_constructible
- is_nothrow_default_constructible
- is_copy_constructible
- is_trivially_copy_constructible
- is_nothrow_copy_constructible
- is_move_constructible
- is_trivially_move_constructible
- is_nothrow_move_constructible
- is_assignable
- is_trivially_assignable
- is_nothrow_assignable
- is_copy_assignable
- is_trivially_copy_assignable
- is_nothrow_copy_assignable
- is_move_assignable
- is_trivially_move_assignable
- is_nothrow_move_assignable
- is_destructible
- is_trivially_destructible
- is_trivially_destructible
- is_nothrow_destructible
- has_virtual_destructor
- is_swappable_with
- is_swappable
- is_nothrow_swappable_with
- is_nothrow_swappable

Supported operations

- is_constructible_v
- is_trivially_constructible_v
- is_nothrow_constructible_v
- is_default_constructible_v
- is_trivially_default_constructible_v
- is_nothrow_default_constructible_v
- is_copy_constructible_v
- is_trivially_copy_constructible_v
- is_nothrow_copy_constructible_v
- is_move_constructible_v
- is_trivially_move_constructible_v
- is_nothrow_move_constructible_v
- is_assignable_v
- is_trivially_assignable_v
- is_nothrow_assignable_v
- is_copy_assignable_v
- is_trivially_copy_assignable_v
- is_nothrow_copy_assignable_v
- is_move_assignable_v
- is_trivially_move_assignable_v
- is_nothrow_move_assignable_v
- is_destructible_v
- is_trivially_destructible_v
- is_trivially_destructible_v
- is_nothrow_destructible_v
- has_virtual_destructor_v
- is_swappable_with_v
- is_swappable_v
- is_nothrow_swappable_with_v
- is_nothrow_swappable_v

Type relationships

- is_same
- is_base_of
- is_convertible
- is_nothrow_convertible
- is_invocable
- is_invocable_r
- is_nothrow_invocable

Type relationships

- is_same_v
- is_base_of_v
- is_convertible_v
- is_nothrow_convertible_v
- is_invocable_v
- is_invocable_r_v
- is_nothrow_invocable_v

Property queries

- alignment_of
- rank
- extent

Property queries

- alignment_of_v
- rank_v
- extent_v

```
template <class T>
struct remove_reference
{
  using type = T;
};

template <class T>
struct remove_reference<T&>
{
  using type = T;
};

template <class T>
struct remove_reference<T&&>
{
  using type = T;
};

template <class T>
using remove_reference_t = typename remove_reference<T>::type;
```

Type transformations

- remove_const
- remove_volatile
- remove_cv
- add_const
- add_volatile
- add_cv
- remove_reference
- add_lvalue_reference
- add_rvalue_reference
- remove_cvref
- remove_pointer
- add_pointer
- make_signed
- make_unsigned
- remove_extent
- remove_all_extents
- decay
- conditional
- underlying_type
- common_type
- result_of / invoke_result

Type transformations

- remove_const_t
- remove_volatile_t
- remove_cv_t
- add_const_t
- add_volatile_t
- add_cv_t
- remove_reference_t
- add_lvalue_reference_t
- add_rvalue_reference_t
- remove_cvref_t
- remove_pointer_t
- add_pointer_t
- make_signed_t
- make_unsigned_t
- remove_extent_t
- remove_all_extents_t
- decay_t
- conditional_t
- underlying_type_t
- common_type_t
- result_of_t / invoke_result_t

```
template <class T, class Alloc = std::allocator<T>>
class vector
{
  ....
public:
  template <class ...Args>
  T& emplace_back(Args ...args);
};
```

```
template <class ...Args>
T& emplace_back(Args ...args)
{
  if (size() == capacity()) // no free space
  {
    const auto oldCap = capacity();
    const auto newCap = computeGrowth(oldCap + 1);

    auto *newVec = allocator_traits<Alloc>::allocate(al, newCap);
    allocator_traits<Alloc>::construct(al, newVec, oldCap + 1,
                                       forward<Args>(args)...);

    if constexpr (   is_nothrow_move_constructible_v<T>
                  || !is_copy_constructible_v<T>)
    {
      uninitialized_move(begin(), end(), ptr);
    }
    else
    {
      uninitialized_copy(begin(), end(), ptr);
    }

    // deallocate old objects and release old memory region
    change_array(newVec, size() + 1, newCap);
  }
  ....
}
```

Curiously recurring template pattern

```
template <class T>
struct Base
{

};

struct Derived : Base<Derived>
{

};
```

```
template <class T = intmax_t>
class Rational
{
  T m_num = T(0), m_denom = T(1);
public:
  Rational() = default;
  explicit Rational(T num, T denom = T(1)) : m_num { num }
                                           , m_denom { denom } { .... }

  ....

  friend bool operator<(const Rational<T> &l, const Rational<T> &r) noexcept
  {
    const auto lcm = std::lcm(l.m_denom, r.m_denom);
    return l.m_num * (lcm / l.m_denom) < r.m_num * (lcm / r.m_denom);
  }
};
```

```
template <class T = intmax_t>
class Rational
{
  T m_num = T(0), m_denom = T(1);
public:
  Rational() = default;
  explicit Rational(T num, T denom = T(1)) : m_num { num }
                                           , m_denom { denom } { .... }

  ....

  friend bool operator<(const Rational<T> &l, const Rational<T> &r) noexcept
  {
    const auto lcm = std::lcm(l.m_denom, r.m_denom);
    return l.m_num * (lcm / l.m_denom) < r.m_num * (lcm / r.m_denom);
  }
};

bool foo(const Rational<> &a, const Rational<> &b)
{
  return a > b; // compile-time error
}
```

```
template <class T>
struct less_than_comparable
{
  friend bool operator> (const T &l, const T &r) noexcept { return r < l; }
  friend bool operator<=(const T &l, const T &r) noexcept { return !(r < l); }
  friend bool operator>=(const T &l, const T &r) noexcept { return !(l < r); }
};

template <class T = uint64_t>
class Rational : less_than_comparable<Rational<T>>
{
  ....
};

bool foo(const Rational<> &a, const Rational<> &b)
{
  return a > b; // ok
}
```

Substitution Failure Is Not An Error (SFINAE)

```
template <class Container>
void PrintContainer(const Container &cont)
{
  if (!cont.empty())
  {
    std::cout << '(';

    for (const auto &value : cont)
    {
      std::cout << ' ' << value;
    }

    std::cout << " )";
  }
}
```

```
void foo()
{
  std::vector<int> v { .... };
  std::deque<int>  d { .... };
  std::list<int>   l { .... };

  PrintContainer(v);
  PrintContainer(d);
  PrintContainer(l);
}
```

```
class SomeClass { .... };
std::ostream& operator<<(std::ostream &out, const SomeClass &obj) { .... }

void foo()
{
  std::vector<SomeClass> v { .... };
  std::deque<SomeClass>  d { .... };
  std::list<SomeClass>   l { .... };

  PrintContainer(v);
  PrintContainer(d);
  PrintContainer(l);
}
```

```
class SomeClass { .... };
std::ostream& operator<<(std::ostream &out, const SomeClass &obj) { .... }

template <typename T1, typename T2>
std::ostream& operator<<(std::ostream &out, const std::pair<T1, T2> &obj)
{
  std::out << '(' << obj.first << ", " << obj.second << ')';
  return out;
}

void foo()
{
  std::set<SomeClass>               s { .... };
  std::unordered_set<SomeClass>  us { .... };

  std::map<std::string, SomeClass>              m { .... };
  std::unordered_map<std::string, SomeClass> um { .... };

  PrintContainer(s);
  PrintContainer(us);
  PrintContainer(m);
  PrintContainer(um);
}
```

```
void foo()
{
  std::stack<int> st { std::deque<int> { .... } };

  PrintContainer(st);
}
```

```
<source>:22:28: error: invalid range expression of type 'const std::stack<int, std::deque<int, std::allocator<int> > >'; no viable
'begin' function available
    for (const auto &value : cont)
                           ^ ~~~~

<source>:34:3: note: in instantiation of function template specialization 'PrintContainer<std::stack<int, std::deque<int,
std::allocator<int> > > >' requested here
  PrintContainer(st);
  ^

/opt/compiler-explorer/gcc-8.3.0/lib/gcc/x86_64-linux-gnu/8.3.0/../../../../include/c++/8.3.0/initializer_list:89:5: note: candidate
template ignored: could not match 'initializer_list' against 'stack'
    begin(initializer_list<_Tp> __ils) noexcept
    ^

/opt/compiler-explorer/gcc-8.3.0/lib/gcc/x86_64-linux-gnu/8.3.0/../../../../include/c++/8.3.0/bits/range_access.h:48:5: note: candidate
template ignored: substitution failure [with _Container = const std::stack<int, std::deque<int, std::allocator<int> > >]: no member
named 'begin' in 'std::stack<int, std::deque<int, std::allocator<int> > >'
    begin(_Container& __cont) -> decltype(__cont.begin())
    ^                                     ~~~~~

/opt/compiler-explorer/gcc-8.3.0/lib/gcc/x86_64-linux-gnu/8.3.0/../../../../include/c++/8.3.0/bits/range_access.h:58:5: note: candidate
template ignored: substitution failure [with _Container = std::stack<int, std::deque<int, std::allocator<int> > >]: no member named
'begin' in 'std::stack<int, std::deque<int, std::allocator<int> > >'
    begin(const _Container& __cont) -> decltype(__cont.begin())
    ^                                           ~~~~~

/opt/compiler-explorer/gcc-8.3.0/lib/gcc/x86_64-linux-gnu/8.3.0/../../../../include/c++/8.3.0/bits/range_access.h:87:5: note: candidate
template ignored: could not match '_Tp [_Nm]' against 'const std::stack<int, std::deque<int, std::allocator<int> > >'
    begin(_Tp (&__arr)[_Nm])
    ^

/opt/compiler-explorer/gcc-8.3.0/lib/gcc/x86_64-linux-gnu/8.3.0/../../../../include/c++/8.3.0/bits/range_access.h:104:31: note:
candidate template ignored: could not match 'valarray' against 'stack'
  template<typename _Tp> _Tp* begin(valarray<_Tp>&);
                         ^

/opt/compiler-explorer/gcc-8.3.0/lib/gcc/x86_64-linux-gnu/8.3.0/../../../../include/c++/8.3.0/bits/range_access.h:105:37: note:
candidate template ignored: could not match 'valarray' against 'stack'
  template<typename _Tp> const _Tp* begin(const valarray<_Tp>&);
                                    ^
1 warning and 1 error generated.
```

```
template <class Container>
void PrintContainer(const Container &cont)
{
  if (!cont.empty())
  {
    std::cout << '(';

    for (const auto &value : cont)
    {
      std::cout << ' ' << value;
    }

    std::cout << " )";
  }
}

void foo()
{
  std::stack<int> st { std::deque<int> { .... } };

  PrintContainer(st);
}
```

```
template <class Container,
          class It = typename Container::const_iterator>
void PrintContainer(const Container &cont)
{
  if (!cont.empty())
  {
    std::cout << '(';

    for (const auto &value : cont)
    {
      std::cout << ' ' << value;
    }

    std::cout << " )";
  }
}

void foo()
{
  std::stack<int> st { std::vector<int> { .... } };

  PrintContainer(st);
}
```

```
<source>:35:3: error: no matching function for call to 'PrintContainer'
  PrintContainer(st);
  ^~~~~~~~~~~~~~

<source>:17:6: note: candidate template ignored: substitution failure
[with Container = std::stack<int, std::deque<int, std::allocator<int> >
>]: no type named 'const_iterator' in 'std::stack<int, std::deque<int,
std::allocator<int> > >'

void PrintContainer(const Container &cont)
     ^

1 warning and 1 error generated.
```

std::enable_if

```
template <bool Cond, class T = void>
struct enable_if
{
};

template <class T>
struct enable_if<true, T>
{
  using type = T;
};

template <bool Cond, class T = void>
using enable_if_t = typename enable_if<Cond, T>::type;
```

std::enable_if

```
template <typename It>
using ItTag = typename std::iterator_traits<It>::iterator_category;

template <class It, class Diff,
          std::enable_if_t<std::is_base_of_v<std::random_access_iterator_tag,
                                             ItTag<It>>,
                           int> = 0>
It next(It it, Diff n) noexcept
{
  return it + n;
}
```

std::enable_if

```
template <typename It>
using ItTag = typename std::iterator_traits<It>::iterator_category;

template <class It, class Diff,
          std::enable_if_t<std::is_same_v<std::bidirectional_iterator_tag,
                                          ItTag<It>>,
                           int> = 0>
It next(It it, Diff n) noexcept
{
  for (; n > Diff(0); --n)
  {
    ++it;
  }

  for (; n < Diff(0); ++n)
  {
    --it;
  }

  return it;
}
```

std::enable_if

```
template <typename It>
using ItTag = typename std::iterator_traits<It>::iterator_category;

template <class It, class Diff,
          std::enable_if_t<std::is_same<std::input_iterator_tag, ItTag<It>>,
                           || std::is_same<std::forward_iterator_tag, ItTag<It>>>,
          int> = 0>
It next(It it, Diff n) noexcept
{
  assert(n >= 0);
  for (; n != Diff(0); --n)
  {
    ++it;
  }

  return it;
}
```

std::enable_if

```
void foo()
{
  std::vector<int> v { 0, 1, 2, 3, 4, 5 };
  std::list<int> l { v.begin(), v.end() };
  std::forward_list<int> fl { v.begin(), v.end() };

  next(v.begin(), 6); // call 'next' for random access iterator
  next(v.end(),  -6); // call 'next' for random access iterator

  next(l.begin(), 6); // call 'next' for bidirectional iterator
  next(l.end(),  -6); // call 'next' for bidirectional iterator

  auto it = next(fl.begin(), 6); // call 'next' for input/forward iterator
       it = next(it,        -6); // compile-time error
}
```

Tag dispatch

```
template <typename It, class Diff>
It next_impl(It it, Diff n, std::input_iterator_tag)
{
  ....
}

template <typename It, class Diff>
It next_impl(It it, Diff n, std::bidirectional_iterator_tag)
{
  ....
}

template <typename It, class Diff>
It next_impl(It it, Diff n, std::random_access_iterator_tag)
{
  ....
}
```

Tag dispatch

```
template <typename It>
using ItTag = typename std::iterator_traits<It>::iterator_category;

template <typename It, class Diff>
It next(It it, Diff n)
{
  return next_impl(it, n, ItTag<It> {});
}
```

'if constexpr'

```
template <typename It>
using ItTag = typename std::iterator_traits<It>::iterator_category;

template <typename It, class Diff>
It next(It it, Diff n)
{
  // for random access iterators and others
  if constexpr(std::is_base_of_v<std::random_access_iterator_tag, ItTag<It>>)
  { .... }
  // for bidirectional iterators
  else if constexpr (std::is_same_v<std::bidirectional_iterator_tag, ItTag<It>>)
  { .... }
  // for input and forward iterators
  else
  { .... }
}
```

Real example based on SFINAE

```
#include <optional>
#include <functional>

template <class T>
using OptRefType = std::optional<std::reference_wrapper<std::remove_reference_t<T>>>;

template <class Func>
class Lazy
{
  using ResultType   = std::invoke_result_t<Func>;
  using OptionalType =
    std::conditional_t<std::is_void_v<ResultType>,
                       bool,
                       std::conditional_t<std::is_reference_v<ResultType>,
                                          OptRefType<ResultType>,
                                          std::optional<ResultType>>>;

  Func m_initializer;
  OptionalType m_value { };

public:

  Lazy(Func &&init) : m_initializer { std::forward<Func>(init) } { }
  ....
};
```

```
template <class Func>
class Lazy
{
  ....
public:

  template <class T = ResultType, std::enable_if_t<!std::is_void_v<T>, int> = 0>
  ResultType operator()()
  {
    if (m_value)
    {
      return *m_value;
    }

    return m_value.emplace(m_initializer());
  }
  ....
};
```

```
template <class Func>
class Lazy
{
  ....
public:

  template <class T = ResultType, std::enable_if_t<std::is_void_v<T>, int> = 0>
  void operator()()
  {
    if (m_value)
    {
      return;
    }

    m_value = true;
    m_initializer();
  }
  ....
};
```

```
template <class Func>
Lazy<Func> CreateLazy(Func &&f)
{
  return { std::forward<Func>(f); }; // before C++17
}

void foo()
{
  // before C++17
  auto idx = CreateLazy([]
                        {
                          auto i = ...; // some calculations
                          return i;
                        });

  // since C++17
  Lazy idx { []{ .... } };

  if (.... && idx()) // short-circuit evaluation
  {
    ....
  }
}
```

Special metafunctions

```
template <class T1, class T2, class ...TN>
bool EqualsAnyOf(const T1 &t1, const T2 &t2, const TN &...tN) noexcept
{
  ....
}
```

```
template <class T1, class T2, class ...TN>
bool EqualsAnyOf(const T1 &t1, const T2 &t2, const TN &...tN) noexcept
{
  ....
}

template <class T1, class T2, class ...TN,
          class = std::enable_if_t<  (std::is_same_v<T1, T2>
                                   && ...
                                   && std::is_same_v<T1, TN>>>
bool EqualsAnyOf(const T1 &t1, const T2 &t2, const TN &...tN) noexcept
{
  ....
}
```

```
template <class T1, class T2, class ...TN>
bool EqualsAnyOf(const T1 &t1, const T2 &t2, const TN &...tN) noexcept
{
  ....
}

template <class T1, class T2, class ...TN,
          class = std::enable_if_t<std::conjunction_v<std::is_same<T1, T2>,
                                                      std::is_same_v<T1, TN>...>>>
bool EqualsAnyOf(const T1 &t1, const T2 &t2, const TN &...tN) noexcept
{
  ....
}
```

```
template <class ...B>
struct conjunction;

template <class ...B>
struct disjunction;

template <class B>
struct negation;
```

```
template <class ...> struct conjunction : std::true_type { };

template <class B1> struct conjunction<B1> : B1 { };

template <class B1, class ...Bn>
struct conjunction<B1, Bn...> : std::conditional_t<bool(B1::value),
                                                   conjunction<Bn...>,
                                                   B1> { };
```

```
template <class ...> struct conjunction : std::true_type { };

template <class B1> struct conjunction<B1> : B1 { };

template <class B1, class ...Bn>
struct conjunction<B1, Bn...> : std::conditional_t<bool(B1::value),
                                                   conjunction<Bn...>,
                                                   B1> { };

template <class ...> struct disjunction : std::false_type { };

template <class B1> struct disjunction<B1> : B1 { };

template <class B1, class ...Bn>
struct disjunction<B1, Bn...> : std::conditional_t<bool(B1::value),
                                                   B1,
                                                   disjunction<Bn...>>  { };
```

```
template <class ...> struct conjunction : std::true_type { };

template <class B1> struct conjunction<B1> : B1 { };

template <class B1, class ...Bn>
struct conjunction<B1, Bn...> : std::conditional_t<bool(B1::value),
                                                   conjunction<Bn...>,
                                                   B1> { };

template <class ...> struct disjunction : std::false_type { };

template <class B1> struct disjunction<B1> : B1 { };

template <class B1, class ...Bn>
struct disjunction<B1, Bn...> : std::conditional_t<bool(B1::value),
                                                   B1,
                                                   disjunction<Bn...>>  { };

template <class B>
struct negation : std::bool_constant<!bool(B::value)> { };
```

void_t

```
template <class ...>
using void_t = void;
```

```
template <class ...>
using void_t = void;

void_t<int,
       unsigned long long
       float,
       double,
       SomeClass,
       std::vector<int>,
       std::stack<bool>>; // void
```

```
template <class T1, class T2, class ...TN>
bool EqualsAnyOf(const T1 &t1, const T2 &t2, const TN &...tN) noexcept
{
  return ( (t1 == t2) || ... || (t1 == tN) ); // since C++17
}

template <class T1, class T2, class ...TN>
bool EqualsAllOf(const T1 &t1, const T2 &t2, const TN &...tN) noexcept
{
  return ( (t1 == t2) && ... && (t1 == tN) ); // since C++17
}

template <class T1, class T2, class ...TN>
bool EqualsNoneOf(const T1 &t1, const T2 &t2, const TN &...tN) noexcept
{
  return ( (t1 != t2) && ... && (t1 != tN) ); // since C++17
}
```

```
template <class T1, class T2, class ...TN>
bool EqualsAnyOf(const T1 &t1, const T2 &t2, const TN &...tN) noexcept;

bool foo()
{
  int *pi = nullptr;
  float *pf = nullptr;

  std::vector<int*> vi;
  std::vector<float*> vf;

  return EqualsAnyOf(nullptr, pi, pf, vi, vf);
}
```

```
<source>:21:37: error: invalid operands to binary expression ('const nullptr_t' and 'const std::vector<int *, std::allocator<int *> >')
  return ( (t1 == t2) || ... || (t1 == tN) );
                                ~~ ^  ~~
<source>:32:10: note: in instantiation of function template specialization 'EqualsAnyOf<nullptr_t, int *, float *, std::vector<int *,
std::allocator<int *> >, std::vector<float *, std::allocator<float *> > >' requested here
  return EqualsAnyOf(nullptr, pi, pf, vi, vf);
         ^
/opt/compiler-explorer/gcc-8.3.0/lib/gcc/x86_64-linux-gnu/8.3.0/../../../../include/c++/8.3.0/system_error:292:3: note: candidate
function not viable: no known conversion from 'const nullptr_t' to 'const std::error_code' for 1st argument
  operator==(const error_code& __lhs, const error_code& __rhs) noexcept
  ^
/opt/compiler-explorer/gcc-8.3.0/lib/gcc/x86_64-linux-gnu/8.3.0/../../../../include/c++/8.3.0/system_error:297:3: note: candidate
function not viable: no known conversion from 'const nullptr_t' to 'const std::error_code' for 1st argument
  operator==(const error_code& __lhs, const error_condition& __rhs) noexcept
  ^
/opt/compiler-explorer/gcc-8.3.0/lib/gcc/x86_64-linux-gnu/8.3.0/../../../../include/c++/8.3.0/system_error:304:3: note: candidate
function not viable: no known conversion from 'const nullptr_t' to 'const std::error_condition' for 1st argument
  operator==(const error_condition& __lhs, const error_code& __rhs) noexcept
  ^
....
1 error generated.
Compiler returned: 1
```

```
template <class T1, class T2, class ...TN,
          class = void_t<decltype(std::declval<T1&>() == std::declval<T2&>()),
                         decltype(std::declval<T1&>() == std::declval<TN&>())...>>
bool EqualsAnyOf(const T1 &t1, const T2 &t2, const TN &...tN) noexcept
{
  return ( (t1 == t2) || ... || (t1 == tN) ); // since C++17
}

bool foo()
{
  int *pi = nullptr;
  float *pf = nullptr;

  std::vector<int*> vi;
  std::vector<float*> vf;

  return EqualsAnyOf(nullptr, pi, pf, vi, vf);
}
```

```
<source>:32:10: error: no matching function for call to 'EqualsAnyOf'

  return EqualsAnyOf(nullptr, pi, pf, vi, vf);

         ^~~~~~~~~~~

<source>:19:6: note: candidate template ignored: substitution failure
[with T1 = nullptr_t, T2 = int *, TN = <float *, std::vector<int *,
std::allocator<int *> >, std::vector<float *, std::allocator<float *> >>]:

invalid operands to binary expression ('nullptr_t' and 'std::vector<int *,
std::allocator<int *> >')

bool EqualsAnyOf(const T1 &t1, const T2 &t2, const TN &...tN) noexcept

     ^

1 error generated.

Compiler returned: 1
```

```
template <class T>
std::add_rvalue_reference<T>_t declval();
```

```
template <class T>
std::add_rvalue_reference<T>_t declval();

struct Default { int foo() const { return 1; } };

struct NonDefault
{
    NonDefault(const NonDefault &) { }
    int foo() const { return 1; }
};

int main()
{
  decltype(Default().foo())    i1 = 1;  // type of i1 is int
  decltype(NonDefault().foo()) i2 = n1; // compile-time error

  decltype(std::declval<NonDefault>().foo()) i2 = i1; // type of i2 is int
}
```

Detectors

```
template <class T, typename = void>
struct is_iterable : std::false_type
{
};

template <class T>
struct is_iterable<T, std::void_t<decltype(std::begin(std::declval<T&>())),
                                  decltype(  std::end(std::declval<T&>()))>>
  : std::true_type
{
};

template <class T>
inline constexpr auto is_iterable_v = is_iterable<T>::value;
```

```cpp
template <class T, typename = void>
struct is_iterable : std::false_type
{
};

template <class T>
struct is_iterable<T, std::void_t<decltype(std::begin(std::declval<T&>())),
                                  decltype(  std::end(std::declval<T&>()))>>
  : std::true_type
{
};

template <class T>
inline constexpr auto is_iterable_v = is_iterable<T>::value;

static_assert(is_iterable_v<std::vector<int>>); // ok

static_assert(is_iterable_v<std::list<int>>); // ok

static_assert(is_iterable_v<std::map<int, int>>); // ok

static_assert(is_iterable_v<std::stack<int>>); // compile-time error
```

```
template <class T, typename = void>
struct is_reverse_iterable : std::false_type
{
};

template <class T>
struct is_reverse_iterable<T, std::void_t<decltype(std::begin(std::declval<T&>())),
                                          decltype(  std::end(std::declval<T&>())),
                                          decltype(std::rbegin(std::declval<T&>())),
                                          decltype(  std::rend(std::declval<T&>()))>>
  : std::true_type
{
};

template <class T>
inline constexpr bool is_reverse_iterable_v = is_reverse_iterable<T>::value;
```

```
template <class T, typename = void>
struct is_reverse_iterable : std::false_type
{
};

template <class T>
struct is_reverse_iterable<T, std::void_t<decltype(std::begin(std::declval<T&>())),
                                          decltype(  std::end(std::declval<T&>())),
                                          decltype(std::rbegin(std::declval<T&>())),
                                          decltype(  std::rend(std::declval<T&>()))>>
  : std::true_type
{
};

template <class T>
inline constexpr bool is_reverse_iterable_v = is_reverse_iterable<T>::value;

static_assert(is_reverse_iterable_v<std::vector<int>>); // ok

static_assert(is_reverse_iterable_v<std::list<int>>); // ok

static_assert(is_reverse_iterable_v<std::map<int, int>>); // ok

static_assert(is_reverse_iterable_v<std::forward_list<int>>); // compile-time error
```

```
template <class Default, class AlwaysVoid, template <class ...> class Op, class ...Args>
struct detector
{
  using value_t = std::false_type;
  using type = Default;
};

template <class Default, template<class ...> class Op, class ...Args>
struct detector<Default, std::void_t<Op<Args...>>, Op, Args...>
{
  using value_t = std::true_type;
  using type = Op<Args...>;
};

struct nonesuch
{
  nonesuch() = delete;
  nonesuch(const nonesuch &) = delete;
  nonesuch(nonesuch &&) = delete;
  nonesuch& operator=(const nonesuch &) = delete;
  nonesuch& operator=(nonesuch &&) = delete;
};
```

```
template <template <class ...> class Op, class ...Args>
using is_detected = typename detector<nonesuch, void, Op, Args...>::value_t;

template <template<class ...> class Op, class ...Args >
inline constexpr bool is_detected_v = is_detected<Op, Args...>::value;

template <template<class...> class Op, class... Args>
using detected_t = typename detector<nonesuch, void, Op, Args...>::type;

template <class Default, template<class...> class Op, class... Args>
using detected_or = detector<Default, void, Op, Args...>;

template <class Default, template<class ...> class Op, class ...Args>
using detected_or_t = typename detected_or<Default, Op, Args...>::type;
```

```
template <class T>
using free_begin = decltype(std::begin(std::declval<T>()));

template <class T>
using free_end = decltype(std::end(std::declval<T>()));

template <class T>
using free_rbegin = decltype(std::rbegin(std::declval<T>()));

template <class T>
using free_rend = decltype(std::rend(std::declval<T>()));


template <class T>
inline constexpr auto is_iterable_v = is_detected_v<free_begin, T>
                                   && is_detected_v<free_end, T>;

template <class T>
inline constexpr auto is_reverse_iterable_v = is_iterable_v<T>
                                           && is_detected_v<free_rbegin, T>
                                           && is_detected_v<free_rend, T>;
```

```cpp
template <class T>
using v_foo_v = decltype(std::declval<T&>().foo());

template <class T>
using v_foo_i = decltype(std::declval<T&>().foo(std::declval<int>()));

template <class T>
inline constexpr bool has_foo = is_detected_v<v_foo_v, T>
                             && is_detected_v<v_foo_i, T>;
```

END

Q&A