Type support (basic types, RTTI, type traits)

(See also type for type system overview)

Basic types

Fundamental types defined by the language

Additional basic types and macros

Defined in header <cstddef>

size_t	unsigned integer type returned by the sizeof operator (typedef)
ptrdiff_t	signed integer type returned when subtracting two pointers (typedef)
nullptr_t (C++11)	the type of the null pointer literal nullptr (typedef)
NULL	implementation-defined null pointer constant (macro constant)
max_align_t (C++11)	trivial type with alignment requirement as great as any other scalar type (typedef)
offsetof	byte offset from the beginning of a standard-layout type to specified member (function macro)
byte (C++17)	the byte type (enum)

Defined in header <cstdbool> (until C++20)
Defined in header <stdbool.h>

__bool_true_false_are_defined (C++11)(deprecated in C++17)	C compatibility macro constant, expands to integer literal 1 (macro constant)

Defined in header <cstdalign> (until C++20)
Defined in header <stdalign.h>

__alignas_is_defined (C++11)(deprecated in C++17)	C compatibility macro constant, expands to integer literal 1 (macro constant)

Fixed width integer types (since C++11)

Numeric limits

Defined in header <limits>

numeric_limits	provides an interface to query properties of all fundamental numeric types. (class template)

C numeric limits interface

Runtime type identification

Defined in header <typeinfo>

type_info	contains some type's information, generated by the implementation. This is the class returned by the typeid operator. (class)
bad_typeid	exception that is thrown if an argument in a typeid expression is null (class)
bad_cast	exception that is thrown by an invalid dynamic_cast expression, i.e. a cast of reference type fails (class)

Defined in header <typeindex>

type_index (C++11)	wrapper around a type_info object, that can be used as index in associative and unordered associative containers (class)

Type traits (since C++11)

Type traits defines a compile-time template-based interface to query or modify the properties of types.

Attempting to specialize a template defined in the <type_traits> header results in undefined behavior, except that std::common_type may be specialized as described in its description.

A template defined in the <type_traits> header may be instantiated with an incomplete type unless otherwise specified, notwithstanding the general prohibition against instantiating standard library templates with incomplete types.

Type properties

Defined in header <type_traits>

Primary type categories

is_void (C++11)	checks if a type is `void` (class template)
is_null_pointer (C++14)	checks if a type is `std::nullptr_t` (class template)
is_integral (C++11)	checks if a type is an integral type (class template)
is_floating_point (C++11)	checks if a type is a floating-point type (class template)
is_array (C++11)	checks if a type is an array type (class template)
is_enum (C++11)	checks if a type is an enumeration type (class template)
is_union (C++11)	checks if a type is an union type (class template)
is_class (C++11)	checks if a type is a non-union class type (class template)
is_function (C++11)	checks if a type is a function type (class template)
is_pointer (C++11)	checks if a type is a pointer type (class template)
is_lvalue_reference (C++11)	checks if a type is a *lvalue reference* (class template)
is_rvalue_reference (C++11)	checks if a type is a *rvalue reference* (class template)
is_member_object_pointer (C++11)	checks if a type is a pointer to a non-static member object (class template)
is_member_function_pointer (C++11)	checks if a type is a pointer to a non-static member function (class template)

Composite type categories

is_fundamental (C++11)	checks if a type is a fundamental type (class template)
is_arithmetic (C++11)	checks if a type is an arithmetic type (class template)
is_scalar (C++11)	checks if a type is a scalar type (class template)
is_object (C++11)	checks if a type is an object type (class template)
is_compound (C++11)	checks if a type is a compound type (class template)
is_reference (C++11)	checks if a type is either a *lvalue reference* or *rvalue reference* (class template)
is_member_pointer (C++11)	checks if a type is a pointer to an non-static member function or object (class template)

Type properties

is_const (C++11)	checks if a type is const-qualified (class template)
is_volatile (C++11)	checks if a type is volatile-qualified (class template)
is_trivial (C++11)	checks if a type is trivial (class template)
is_trivially_copyable (C++11)	checks if a type is trivially copyable (class template)
is_standard_layout (C++11)	checks if a type is a standard-layout type (class template)
is_pod (C++11)(deprecated in C++20)	checks if a type is a plain-old data (POD) type (class template)
is_literal_type (C++11) (deprecated in C++17) (removed in C++20)	checks if a type is a literal type (class template)
has_unique_object_representations (C++17)	checks if every bit in the type's object representation contributes to its value (class template)
is_empty (C++11)	checks if a type is a class (but not union) type and has no non-static data members (class template)
is_polymorphic (C++11)	checks if a type is a polymorphic class type (class template)
is_abstract (C++11)	checks if a type is an abstract class type (class template)
is_final (C++14)	checks if a type is a final class type (class template)
is_aggregate (C++17)	checks if a type is an aggregate type (class template)
is_signed (C++11)	checks if a type is a signed arithmetic type

(class template)

is_unsigned (C++11)	checks if a type is an unsigned arithmetic type (class template)
is_bounded_array (C++20)	checks if a type is an array type of known bound (class template)
is_unbounded_array (C++20)	checks if a type is an array type of unknown bound (class template)

Supported operations

is_constructible (C++11) **is_trivially_constructible** (C++11) **is_nothrow_constructible** (C++11)	checks if a type has a constructor for specific arguments (class template)
is_default_constructible (C++11) **is_trivially_default_constructible** (C++11) **is_nothrow_default_constructible** (C++11)	checks if a type has a default constructor (class template)
is_copy_constructible (C++11) **is_trivially_copy_constructible** (C++11) **is_nothrow_copy_constructible** (C++11)	checks if a type has a copy constructor (class template)
is_move_constructible (C++11) **is_trivially_move_constructible** (C++11) **is_nothrow_move_constructible** (C++11)	checks if a type can be constructed from an rvalue reference (class template)
is_assignable (C++11) **is_trivially_assignable** (C++11) **is_nothrow_assignable** (C++11)	checks if a type has a assignment operator for a specific argument (class template)
is_copy_assignable (C++11) **is_trivially_copy_assignable** (C++11) **is_nothrow_copy_assignable** (C++11)	checks if a type has a copy assignment operator (class template)
is_move_assignable (C++11) **is_trivially_move_assignable** (C++11) **is_nothrow_move_assignable** (C++11)	checks if a type has a move assignment operator (class template)
is_destructible (C++11) **is_trivially_destructible** (C++11) **is_nothrow_destructible** (C++11)	checks if a type has a non-deleted destructor (class template)
has_virtual_destructor (C++11)	checks if a type has a virtual destructor (class template)
is_swappable_with (C++17) **is_swappable** (C++17) **is_nothrow_swappable_with** (C++17) **is_nothrow_swappable** (C++17)	checks if objects of a type can be swapped with objects of same or different type (class template)

Property queries

alignment_of (C++11)	obtains the type's alignment requirements (class template)
rank (C++11)	obtains the number of dimensions of an array type (class template)
extent (C++11)	obtains the size of an array type along a specified dimension (class template)

Type relationships

is_same (C++11)	checks if two types are the same (class template)
is_base_of (C++11)	checks if a type is derived from the other type (class template)
is_convertible (C++11) **is_nothrow_convertible** (C++20)	checks if a type can be converted to the other type (class template)
is_invocable **is_invocable_r** **is_nothrow_invocable** (C++17) **is_nothrow_invocable_r**	checks if a type can be invoked (as if by `std::invoke`) with the given argument types (class template)
is_layout_compatible (C++20)	checks if two types are *layout-compatible* (class template)
is_pointer_interconvertible_base_of (C++20)	checks if a type is a *pointer-interconvertible* (initial) base of another type (class template)
is_pointer_interconvertible_with_class (C++20)	checks if objects of a type are pointer-interconvertible with the specified subobject of that type (function template)
is_corresponding_member (C++20)	checks if two specified members correspond to each other in the common initial subsequence of two specified types (function template)

Type modifications

Type modification templates create new type definitions by applying modifications on a template parameter. The resulting type can then be accessed through `type` member typedef.

Defined in header <type_traits>

Const-volatility specifiers

remove_cv (C++11) **remove_const** (C++11) **remove_volatile** (C++11)	removes `const` or/and `volatile` specifiers from the given type (class template)
add_cv (C++11) **add_const** (C++11) **add_volatile** (C++11)	adds `const` or/and `volatile` specifiers to the given type (class template)

References

remove_reference (C++11)	removes a reference from the given type (class template)
add_lvalue_reference (C++11) **add_rvalue_reference** (C++11)	adds a *lvalue* or *rvalue* reference to the given type (class template)

Pointers

remove_pointer (C++11)	removes a pointer from the given type (class template)
add_pointer (C++11)	adds a pointer to the given type (class template)

Sign modifiers

make_signed (C++11)	makes the given integral type signed (class template)
make_unsigned (C++11)	makes the given integral type unsigned (class template)

Arrays

remove_extent (C++11)	removes one extent from the given array type (class template)
remove_all_extents (C++11)	removes all extents from the given array type (class template)

Miscellaneous transformations

Defined in header <type_traits>

aligned_storage (C++11)	defines the type suitable for use as uninitialized storage for types of given size (class template)
aligned_union (C++11)	defines the type suitable for use as uninitialized storage for all given types (class template)
decay (C++11)	applies type transformations as when passing a function argument by value (class template)
remove_cvref (C++20)	combines `std::remove_cv` and `std::remove_reference` (class template)
enable_if (C++11)	hides a function overload or template specialization based on compile-time boolean (class template)
conditional (C++11)	chooses one type or another based on compile-time boolean (class template)
common_type (C++11)	determines the common type of a group of types (class template)
common_reference **basic_common_reference** (C++20)	determines the common reference type of a group of types (class template)
underlying_type (C++11)	obtains the underlying integer type for a given enumeration type (class template)
result_of (C++11)(removed in C++20) **invoke_result** (C++17)	deduces the result type of invoking a callable object with a set of arguments (class template)
void_t (C++17)	void variadic alias template (alias template)
type_identity (C++20)	returns the type argument unchanged (class template)

Operations on traits

Defined in header <type_traits>

conjunction (C++17)	variadic logical AND metafunction (class template)
disjunction (C++17)	variadic logical OR metafunction (class template)
negation (C++17)	logical NOT metafunction (class template)

Helper classes

Defined in header <type_traits>

integral_constant (C++11) **bool_constant** (C++17)	compile-time constant of specified type with specified value (class template)

Two specializations of `std::integral_constant` for the type `bool` are provided:

Defined in header <type_traits>

Type	Definition
`true_type`	`std::integral_constant<bool, true>`
`false_type`	`std::integral_constant<bool, false>`

Constant evaluation context (since C++20)

Defined in header <type_traits>

is_constant_evaluated (C++20)	detects whether the call occurs within a constant-evaluated context (function)

See also

C documentation for **Type support library**