Username: Pralay Patoria **Book:** The C++ Standard Library: A Tutorial and Reference, Second Edition. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

5.4. Type Traits and Type Utilities

Almost everything in the C++ standard library is template based. To support the programming of templates, sometimes called *metaprogramming*, template utilities are provided to help both programmers and library implementers.

Type traits, which were introduced with TR1 and extended with C++11, provide a mechanism to define behavior depending on types. They can be used to optimize code for types that provide special abilities.

Other utilities, such as reference and function wrappers, might also be helpful.

5.4.1. Purpose of Type Traits

A type trait provides a way to deal with the properties of a type. It is a template, which at compile time yields a specific type or value based on one or more passed template arguments, which are usually types.

Consider the following example:

[Click here to view code image]

```
template <typename T>
void foo (const T& val)
{
    if (std::is_pointer<T>::value) {
        std::cout << "foo() called for a pointer" << std::endl;
    }
    else {
        std::cout << "foo() called for a value" << std::endl;
    }
    ...
}
```

Here, the trait `std::is_pointer`, defined in `<type_traits>`, is used to check whether type `T` is a pointer type. In fact, `is_pointer<>` yields either a type `true_type` or a type `false_type`, for which `::value` either yields `true` or `false`. As a consequence, `foo()` will output

```
foo() called for a pointer
```

if the passed parameter `val` is a pointer.

Note, however, that you can't do something like:

```
template <typename T>
void foo (const T& val)
{
    std::cout << (std::is_pointer<T>::value ? *val : val)
              << std::endl;
}
```

The reason is that code is generated for both `*val` and `val`. Even when passing an `int` so that the expression `is_pointer<T>::value` yields `false` at compile time, the code expands to:

```
cout << (false ? *val : val) << endl;
```

And this won't compile, because `*val` is an invalid expression for `int`s.

But you can do the following:

[Click here to view code image]

```
// foo() implementation for pointer types:
template <typename T>
void foo_impl (const T& val, std::true_type)
{
    std::cout << "foo() called for pointer to " << *val
              << std::endl;
}

// foo() implementation for non-pointer types:
template <typename T>
void foo_impl (const T& val, std::false_type)
```

```
{
    std::cout << "foo() called for value " << val
              << std::endl;
}

template <typename T>
void foo (const T& val)
{
    foo_impl (val, std::is_pointer<T>());
}
```

Here, inside `foo()`, the expression

```
std::is_pointer<T>()
```

at compile time yields `std::true_type` or `std::false_type`, which defines which of the provided `foo_impl()` overloads gets instantiated.

Why is that better than providing two overloads of `foo()`: one for ordinary types and one for pointer types?

[Click here to view code image]

```
template <typename T>
void foo (const T& val);        // general implementation

template <typename T>
void foo<T*> (const T& val);    // partial specialization for pointers
```

One answer is that sometimes, too many overloads are necessary. In general, the power of type traits comes more from the fact that they are *building blocks* for generic code, which can be demonstrated by two examples.

Flexible Overloading for Integral Types

In [Becker:LibExt], Pete Becker gives a nice example, which I modified slightly here. Suppose that you have a function `foo()` that should be implemented differently for integral and floating-point type arguments. The usual approach would be to overload this function for all available integral and floating-point types:[21]

[21] According to the C++ standard, the term *integral type* includes `bool` and character types, but that's not meant here.

```
void foo (short);            // provide integral version
void foo (unsigned short);
void foo (int);
…
void foo (float);            // provide floating-point version
void foo (double);
void foo (long double);
```

This repetition is not only tedious but also introduces the problem that it might not work for new integral or floating-point types, either provided by the standard, such as `long long`, or provided as user-defined types.

With the type traits, you can provide the following instead:

[Click here to view code image]

```
template <typename T>
void foo_impl (T val, true_type);     // provide integral version

template <typename T>
void foo_impl (T val, false_type);    // provide floating-point version

template <typename T>
void foo (T val)
{
    foo_impl (val, std::is_integral<T>());
}
```

Thus, you provide two implementations — one for integral and one for floating-point types — and choose the right implementation according to what `std::is_integral<>` yields for the type.

Processing the Common Type

Another example for the usability of type traits is the need to process the "common type" of two or more types. This is a type I could use to deal with the values of two different types, provided there is a common type. For example, it would be an appropriate type of the minimum or the sum of two values of different type. Otherwise, if I want to implement a function that yields the minimum of two values of different types, which return type should it have:

```
template <typename T1, typename T2>
??? min (const T1& x, const T2& y);
```

Using the type traits, you can simply use the `std::common_type<>` to declare this type:

```
template <typename T1, typename T2>
typename std::common_type<T1,T2>::type min (const T1& x, const T2& y);
```

For example, the expression `std::common_type<T1,T2>::type` yields `int` if both arguments are `int`, `long`, if one is `int` and the other is `long`, or `std::string` if one is a string and the other is a string literal (type `const char*`).

How does it do that? Well, it simply uses the rules implemented for operator `?:`, which has to yield one result type based on the types of both operands. In fact, `std::common_type<>` is implemented as follows:

```
template <typename T1, typename T2>
struct common_type<T1,T2> {
    typedef decltype(true ? declval<T1>() : declval<T2>()) type;
};
```

where `decltype` is a new keyword in C++11 (see Section 3.1.11, page 32) to yield the type of an expression, and `declval<>` is an auxiliary trait to provide a declared value of the passed type without evaluating it (generating an rvalue reference for it).

Thus, when operator `?:` is able to find a common type, `common_type<>` will yield it. If not, you can still provide an overload of `common_type<>` (this, for example, is used by the chrono library to be able to combine durations; see Section 5.7.2, page 145).

5.4.2. Type Traits in Detail

The type traits are usually defined in `<type_traits>`.

(Unary) Type Predicates

As introduced in Section 5.4.1, page 122, the type predicates yield `std::true_type` if a specific property applies and `std::false_type` if not. These types are specialization of the helper `std::integral_constant`, so their corresponding `value` members yield `true` or `false`:

```
namespace std {
    template <typename T, T val>
    struct integral_constant {
        static constexpr T value = val;
        typedef T value_type;
        typedef integral_constant<T,val> type;
        constexpr operator value_type() {
            return value;
        }
    };
    typedef integral_constant<bool,true>  true_type;
    typedef integral_constant<bool,false> false_type;
}
```

Table 5.13 lists the type predicates provided for all types. Table 5.14 lists the traits that clarify details of class types.

Table 5.13. Traits to Check Type Properties

Trait	Effect
is_void<T>	Type void
is_integral<T>	Integral type (including bool, char, char16_t, char32_t, wchar_t)
is_floating_point<T>	Floating-point type (float, double, long double)
is_arithmetic<T>	Integral (including bool and characters) or floating-point type
is_signed<T>	Signed arithmetic type
is_unsigned<T>	Unsigned arithmetic type
is_const<T>	const qualified
is_volatile<T>	volatile qualified
is_array<T>	Ordinary array type (not type std::array)
is_enum<T>	Enumeration type
is_union<T>	Union type
is_class<T>	Class/struct type but not a union type
is_function<T>	Function type
is_reference<T>	Lvalue or rvalue reference
is_lvalue_reference<T>	Lvalue reference
is_rvalue_reference<T>	Rvalue reference
is_pointer<T>	Pointer type (including function pointer but not pointer to nonstatic member)
is_member_pointer<T>	Pointer to nonstatic member
is_member_object_pointer<T>	Pointer to a nonstatic data member
is_member_function_pointer<T>	Pointer to a nonstatic member function
is_fundamental<T>	void, integral (including bool and characters), floating-point, or std::nullptr_t
is_scalar<T>	Integral (including bool and characters), floating-point, enumeration, pointer, member pointer, std::nullptr_t
is_object<T>	Any type except void, function, or reference
is_compound<T>	Array, enumeration, union, class, function, reference, or pointer
is_trivial<T>	Scalar, trivial class, or arrays of these types
is_trivially_copyable<T>	Scalar, trivially copyable class, or arrays of these types
is_standard_layout<T>	Scalar, standard layout class, or arrays of these types
is_pod<T>	Plain old data type (type where memcpy() works to copy objects)
is_literal_type<T>	Scalar, reference, class, or arrays of these types

Table 5.14. Traits to Check Type Properties of Class Types

Trait	Effect
`is_empty<T>`	Class with no members, virtual member functions, or virtual base classes
`is_polymorphic<T>`	Class with a (derived) virtual member function
`is_abstract<T>`	Abstract class (at least one pure virtual function)
`has_virtual_destructor<T>`	Class with virtual destructor
`is_default_constructible<T>`	Class enables default construction
`is_copy_constructible<T>`	Class enables copy construction
`is_move_constructible<T>`	Class enables move construction
`is_copy_assignable<T>`	Class enables copy assignment
`is_move_assignable<T>`	Class enables move assignment
`is_destructible<T>`	Class with callable destructor (not deleted, protected, or private)
`is_trivially_default_constructible<T>`	Class enables trivial default construction
`is_trivially_copy_constructible<T>`	Class enables trivial copy construction
`is_trivially_move_constructible<T>`	Class enables trivial move construction
`is_trivially_copy_assignable<T>`	Class enables trivial copy assignment
`is_trivially_move_assignable<T>`	Class with trivial move assignment
`is_trivially_destructible<T>`	Class with trivial callable destructor
`is_nothrow_default_constructible<T>`	Class enables default construction that doesn't throw
`is_nothrow_copy_constructible<T>`	Class enables copy construction that doesn't throw
`is_nothrow_move_constructible<T>`	Class enables move construction that doesn't throw
`is_nothrow_copy_assignable<T>`	Class enables copy assignment that doesn't throw
`is_nothrow_move_assignable<T>`	Class enables move assignment that doesn't throw
`is_nothrow_destructible<T>`	Class with callable destructor that doesn't throw

Note that `bool` and all character types (`char`, `char16_t`, `char32_t`, and `wchar_t`) count as integral types and that type `std::nullptr_t` (see Section 3.1.1, page 14) counts as a fundamental data type.

Most, but not all, of these traits are unary. That is, they use one template argument. For example, `is_const<>` checks whether the passed type is `const`:

```
is_const<int>::value                   // false
is_const<const volatile int>::value    // true
is_const<int* const>::value            // true
is_const<const int*>::value            // false
is_const<const int&>::value            // false
is_const<int[3]>::value                // false
is_const<const int[3]>::value          // true
is_const<int[]>::value                 // false
is_const<const int[]>::value           // true
```

Note that a nonconstant pointer or reference to a constant type is not constant, whereas an ordinary array of constant elements is.[22]

[22] Whether this is correct is currently an issue to decide in the core language group.

Note that the traits checking for copy and move semantics only check whether the corresponding expressions are possible. For example, a type with a copy constructor with constant argument but no move constructor is still move constructible.

The `is_nothrow` ... type traits are especially used to formulate `noexcept` specifications (see Section 3.1.7, page 24).

Traits for Type Relations

Table 5.15 lists the type traits that allow checking relations between types. This includes checking which constructors and assignment operators are provided for class types.

Table 5.15. Traits to Check Type Relations

Trait	Effect
`is_same<T1,T2>`	*T1* and *T2* are the same types (including `const`/`volatile` qualifiers)
`is_base_of<T,D>`	Type *T* is base class of type *D*
`is_convertible<T,T2>`	Type *T* is convertible into type *T2*
`is_constructible<T,Args...>`	Can initialize type *T* with types *Args*
`is_trivially_constructible<T,Args...>`	Can trivially initialize type *T* with types *Args*
`is_nothrow_constructible<T,Args...>`	Initializing type *T* with types *Args* doesn't throw
`is_assignable<T,T2>`	Can assign type *T2* to type *T*
`is_trivially_assignable<T,T2>`	Can trivially assign type *T2* to type *T*
`is_nothrow_assignable<T,T2>`	Assigning type *T2* to type *T* doesn't throw
`uses_allocator<T,Alloc>`	*Alloc* is convertible into *T*`::allocator_type`

Note that a type like `int` represents an lvalue or an rvalue. Because you can't assign

```
42 = 77;
```

`is_assignable<>` for a nonclass type as first type always yields `false_type`. For class types, however, passing their ordinary type as first type is fine because there is a funny old rule that you can invoke member functions on rvalues of class types.[23] For example:

[23] Thanks to Daniel Krügler for pointing this out.

```
is_assignable<int,int>::value                      // false
is_assignable<int&,int>::value                     // true
is_assignable<int&&,int>::value                    // false
is_assignable<long&,int>::value                    // true
is_assignable<int&,void*>::value                   // false
is_assignable<void*,int>::value                    // false
is_assignable<const char*,std::string>::value      // false
is_assignable<std::string,const char*>::value      // true
```

Trait `is_constructible<>` yields, for example, the following:

Click here to view code image

```
is_constructible<int>::value                                  // true
is_constructible<int,int>::value                              // true
is_constructible<long,int>::value                             // true
is_constructible<int,void*>::value                            // false
is_constructible<void*,int>::value                            // false
is_constructible<const char*,std::string>::value              // false
is_constructible<std::string,const char*>::value              // true
is_constructible<std::string,const char*,int,int>::value      // true
```

`std::uses_allocator<>` is defined in `<memory>` (see Section 19.1, page 1024).

Type Modifiers

The traits listed in Table 5.16 allow you to modify types.

All modifying traits add a type property, provided it doesn't exist yet, or remove a property provided it exists already. For example, type `int` might only be extended:

```
typedef int T;
add_const<T>::type                   // const int
add_lvalue_reference<T>::type        // int&
add_rvalue_reference<T>::type        // int&&
add_pointer<T>::type                 // int*
make_signed<T>::type                 // int
make_unsigned<T>::type               // unsigned int
remove_const<T>::type                // int
remove_reference<T>::type            // int
remove_pointer<T>::type              // int
```

Table 5.16. Traits for Type Modifications

Trait	Effect
`remove_const<T>`	Corresponding type without `const`
`remove_volatile<T>`	Corresponding type without `volatile`
`remove_cv<T>`	Corresponding type without `const` and `volatile`
`add_const<T>`	Corresponding `const` type
`add_volatile<T>`	Corresponding `volatile` type
`add_cv<T>`	Corresponding `const volatile` type
`make_signed<T>`	Corresponding signed nonreference type
`make_unsigned<T>`	Corresponding unsigned nonreference type
`remove_reference<T>`	Corresponding nonreference type
`add_lvalue_reference<T>`	Corresponding lvalue reference type (rvalues become lvalues)
`add_rvalue_reference<T>`	Corresponding rvalue reference type (lvalues remain lvalues)
`remove_pointer<T>`	Referred type for pointers (same type otherwise)
`add_pointer<T>`	Type of pointer to corresponding nonreference type

whereas type `const int&` might be reduced and/or extended:

[Click here to view code image]

```
typedef const int& T;
add_const<T>::type                  // const int&
add_lvalue_reference<T>::type       // const int&
add_rvalue_reference<T>::type       // const int& (yes, lvalue remains lvalue)
add_pointer<T>::type                // const int*
make_signed<T>::type                // undefined behavior
make_unsigned<T>::type              // undefined behavior
remove_const<T>::type               // const int&
remove_reference<T>::type           // const int
remove_pointer<T>::type             // const int&
```

Note again that a reference to a constant type is not constant, so you can't remove `const`ness there. Note that `add_pointer<>` implies the application of `remove_reference<>`. However, `make_signed<>` and `make_unsigned<>` require that the arguments be either integral or enumeration types, except `bool`, so passing references results in undefined behavior.

Note that `add_lvalue_reference<>` converts an rvalue reference into an lvalue reference, whereas `add_rvalue_reference<>` does *not* convert an lvalue reference into an rvalue reference (the type remains as it was). Thus, to convert an lvalue into a rvalue reference, you have to call:

```
add_rvalue_reference<remove_reference<T>::type>::type
```

Other Type Traits

Table 5.17 lists all remaining type traits. They query special properties, check type relations, or provide more complicated type transformations.

Table 5.17. Other Type Traits

Trait	Effect
`rank<T>`	Number of dimensions of an array type (or 0)
`extent<T,I=0>`	Extent of dimension *I* (or 0)
`remove_extent<T>`	Element types for arrays (same type otherwise)
`remove_all_extents<T>`	Element type for multidimensional arrays (same type otherwise)
`underlying_type<T>`	Underlying type of an enumeration type (see Section 3.1.13, page 32)
`decay<T>`	Transfers to corresponding "by-value" type
`enable_if<B,T=void>`	Yields type *T* only if `bool` *B* is `true`
`conditional<B,T,F>`	Yields type *T* if `bool` *B* is `true` and type *F* otherwise
`common_type<T1,...>`	Common type of all passed types
`result_of<F,ArgTypes>`	Type of calling *F* with argument types *ArgTypes*
`alignment_of<T>`	Equivalent to `alignof(T)`
`aligned_storage<Len>`	Type of *Len* bytes with default alignment
`aligned_storage<Len,Align>`	Type of *Len* bytes aligned according to a divisor of `size_t` *Align*
`aligned_union<Len,Types...>`	Type of *Len* bytes aligned for a union of *Types...*

The traits that deal with `rank` s and `extent` s allow you to deal with (multidimensional) arrays. For example:

```
rank<int>::value                      // 0
rank<int[]>::value                    // 1
rank<int[5]>::value                   // 1
rank<int[][7]>::value                 // 2
rank<int[5][7]>::value                // 2
extent<int>::value                    // 0
extent<int[]>::value                  // 0
extent<int[5]>::value                 // 5
extent<int[][7]>::value               // 0
extent<int[5][7]>::value              // 5
extent<int[][7],1>::value             // 7
extent<int[5][7],1>::value            // 7
extent<int[5][7],2>::value            // 0
remove_extent<int>::type              // int
remove_extent<int[]>::type            // int
remove_extent<int[5]>::type           // int
remove_extent<int[][7]>::type         // int[7]
remove_extent<int[5][7]>::type        // int[7]
remove_all_extents<int>::type         // int
remove_all_extents<int[]>::type       // int
remove_all_extents<int[5]>::type      // int
remove_all_extents<int[][7]>::type    // int
remove_all_extents<int[5][7]>::type   // int
```

Trait `decay<>` provides the ability to convert a type `T` into its corresponding type when this type is passed by value. Thus, it converts array and function types into pointers as well as lvalues into rvalues, including removing `const` and `volatile` . See Section 5.1.1, page 65, for an example of its use.

As introduced in Section 5.4.1, page 124, `common_type<>` provides a common type for all passed types (may be one, two, or more type arguments).

5.4.3. Reference Wrappers

Class `std::reference_wrapper<>` , declared in `<functional>` , is used primarily to "feed" references to function templates that take their parameter by value. For a given type `T` , this class provides `ref()` for an implicit conversion to `T&` and `cref()` for an implicit conversion to `const T&` , which usually allows function templates to work on references without specialization.

For example, after a declaration such as

```
template <typename T>
void foo (T val);
```

by calling

```
int x;
foo (std::ref(x));
```

`T` becomes `int&`, whereas by calling

```
int x;
foo (std::cref(x));
```

`T` becomes `const int&`.

This feature is used by the C++ standard library at various places. For example:

- `make_pair()` uses this to be able to create a `pair<>` of references (see Section 5.1.1, page 66).
- `make_tuple()` uses this to be able to create a `tuple<>` of references (see Section 5.1.2, page 70).
- Binders use this to be able to bind references (see Section 10.2.2, page 491).
- Threads use this to pass arguments by reference (see Section 18.2.2, page 971).

Note also that class `reference_wrapper` allows you to use references as first-class objects, such as element type in arrays or STL containers:

Click here to view code image

```
std::vector<MyClass&> coll;                              // Error
std::vector<std::reference_wrapper<MyClass>> coll;      // OK
```

See Section 7.11, page 391, for details.

5.4.4. Function Type Wrappers

Class `std::function<>`, declared in `<functional>`, provides polymorphic wrappers that generalize the notion of a function pointer. This class allows you to use *callable objects* (functions, member functions, function objects, and lambdas; see Section 4.4, page 54) as first-class objects.

For example:

```
void func (int x, int y);

// initialize collections of tasks:
std::vector<std::function<void(int,int)>> tasks;
tasks.push_back(func);
tasks.push_back([] (int x, int y) {
                    ...
                });

// call each task:
for (std::function<void(int,int)> f : tasks) {
    f(33,66);
}
```

To call each task, you could also simply call:

```
// call each task:
for (auto f : tasks) {
    f(33,66);
}
```

When member functions are used, the object they are called for has to be the first argument:

```
class C {
  public:
    void memfunc (int x, int y) const;
};

std::function<void(const C&,int,int)> mf;
mf = &C::memfunc;
mf(C(),42,77);
```

Another application of this is to declare functions that return lambdas (see Section 3.1.10, page 31).

Note that performing a function call without having a *target* to call throws an exception of type `std::bad_function_call` (see Section 4.3.1, page 43):

```
std::function<void(int,int)> f;
f(33,66);   // throws std::bad_function_call
```