# Tick Documentation

*Release 1.2*

**Paul Fultz II**

**Jan 29, 2017**

Contents

**Paul Fultz II**

CHAPTER 1

# Getting Started

Tick provides a mechanism for easily defining and using traits in C++11. For example, if we defined a generic `increment` function, like this:

```
template<class T>
void increment(T& x)
{
    x++;
}
```

If we pass something that does not have the `++` operator to `increment`, we will get an error inside of the `increment` function. This can make it unclear whether the error is due to a mistake by the user of the function or by the implementor of the function. Instead we want to check the type requirements of the function.

Using Tick we can create an `is_incrementable` trait, like this:

```
TICK_TRAIT(is_incrementable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x++),
        decltype(++x)
    >;
};
```

And then we can use a simple requires clause in our function to check the type requirements:

```
template<class T, TICK_REQUIRES(is_incrementable<T>())>
void increment(T& x)
{
    x++;
}
```

So, now, if we pass something that is not incrementable to `increment`:

```
struct foo {};

foo f;
increment(f);
```

Then we get an error like this in clang:

```
demo.cpp:25:2: error: no matching function for call to 'increment'
        increment(f);
```

```
        ^~~~~~~~~
demo.cpp:14:19: note: candidate template ignored: disabled by 'enable_if' [with T =␣
↪foo]
template<class T, TICK_REQUIRES(is_incrementable<T>())>
                  ^
```

This gives an error at the call to `increment` rather than inside the function, and then pointes to the type requirements of the function. This gives enough information for most commons cases, however, sometimes we may want more information. In that case the `TICK_TRAIT_CHECK` can be used. For example, say we had the `is_incrementable` trait defined like this:

```
TICK_TRAIT(is_incrementable, std::is_integral<_>)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x++),
        decltype(++x)
    >;
};
```

Then if we use `TICK_TRAIT_CHECK` , we can see why `int*` is not incrementable:

```
TICK_TRAIT_CHECK(is_incrementable<int*>);
```

Which will produce this error:

```
../tick/trait_check.h:95:38: error: implicit instantiation of undefined template
↪'tick::TRAIT_CHECK_FAILURE<std::is_integral<int *>, is_incrementable<int *> >'
```

Which shows the traits that failed including any refinements. So we can see that it failed because `std::is_integral<int *>` is not true.

CHAPTER 2

# Requirements

This requires a C++11 compiler. There a no third-party dependencies. This has been tested on clang 3.4, gcc 4.6-4.9, and Visual Studio 2015.

CHAPTER 3

# Building traits

## 3.1 TICK_TRAIT

This macro will build a boolean type trait for you. Each trait has a `require` member function of the form:

```
TICK_TRAIT(my_trait)
{
    template<class T>
    auto require(T&& x) -> valid<
        ...
    >;
};
```

This will essentially build a class that inherits from `integral_constant` , so the above is equivalent to this:

```
template<class... Ts>
struct my_trait
: integral_constant<bool, (...)>
{};
```

The parameters to the trait are based on the parameters passed to the `require` function.

The trait will be either true or false if the expressions given are valid. Each expression in `valid` needs a `decltype` around it. If one of the expressions is not valid, the the trait will return false. For example,

```
TICK_TRAIT(my_trait)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x++)
    >;
};
```

The trait above will check that `x++` is a valid expression.

## 3.2 Refinements

Refinements can be expressed after the name. Each refinement is a placeholder expression, where each placeholder(ie `_1` , `_2` , etc) are replaced by their corresponding type passed into the trait. In the case of traits that accept a single parameter the unnamed placeholder(`_` ) can be used, for example:

```
TICK_TRAIT(is_incrementable, std::is_default_constructible<_>)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x++),
        decltype(++x)
    >;
};
```

This trait will be true when `x++` and `++x` are valid expressions and `x` is default constructible.

When a trait has multiple parameters, its best to use named placeholders. For example:

```
TICK_TRAIT(is_equality_comparable,
    std::is_default_constructible<_1>,
    std::is_default_constructible<_2>)
{
    template<class T, class U>
    auto require(T&& x, U&& y) -> valid<
        decltype(x == y),
        decltype(x != y)
    >;
};
```

This trait will be true when `x == y` and `x != y` are valid expressions and both `x` and `y` are default constructible.

In addition `quote` can be used to pass all the args from the trait to the refinement:

```
TICK_TRAIT(is_comparable,
    quote<is_equality_comparable>)
{
    template<class T, class U>
    auto require(T&& x, U&& y) -> valid<
        decltype(x < y),
        decltype(x <= y),
        decltype(x >= y),
        decltype(x > y)
    >;
};
```

CHAPTER 4

# Query operations

These can be used to query more information about the types then just valid expressions.

When a type is matched in query operation, it can either be convertible to the type given, or the evaluated placeholder expression must be true. Placeholder expressions can be given so the type can be matched against other traits.

## 4.1 returns

The `returns` query can check if the result of the expressions matches the type. For example,

```
TICK_TRAIT(is_incrementable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(returns<int>(x++))
    >;
};
```

This trait will be true if the expressions `x++` is valid and is convertible to `int` .

Here's an example using placeholder expressions as well:

```
TICK_TRAIT(is_incrementable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(returns<std::is_integral<_>>(x++))
    >;
};
```

This trait will be true if the expressions `x++` is valid and returns a type that `is_integral` .

Note: The `TICK_RETURNS` macro can be used instead to improve compatability with older compilers(such as gcc 4.6):

```
TICK_TRAIT(is_incrementable)
{
    template<class T>
    auto require(T&& x) -> valid<
        TICK_RETURNS(x++, int)
    >;
};
```

Also, `returns<void>` is prohibited.

```
TICK_TRAIT(is_incrementable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(returns<void>(x++)) // Compiler error
    >;
};
```

Instead, use either `decltype` directly without `returns`, or if there is a possibility of `void` from a computed type, use `TICK_RETURNS` or `has_type` instead.

## 4.2 has_type

The `has_type` query can check if a type exist and if the type matches. For example:

```
TICK_TRAIT(has_nested_type)
{
    template<class T>
    auto require(const T& x) -> valid<
        has_type<typename T::type>
    >;
};
```

This trait will be true if `T` has a nested type called `type`.

Now `has_type` used as above is not quite as useful since the above example, can also be simply written without `has_type` like this:

```
TICK_TRAIT(has_nested_type)
{
    template<class T>
    auto require(const T& x) -> valid<
        typename T::type
    >;
};
```

So, an optional second parameter can be provided to check if the type matches. Here's an example:

```
TICK_TRAIT(has_nested_int_type)
{
    template<class T>
    auto require(const T& x) -> valid<
        has_type<typename T::type, std::is_integral<_>>
    >;
};
```

This trait will be true if `T` has a nested type called `type` which is an integral type.

## 4.3 has_template

The `has_template` query can check if a template exist. For example:

```
TICK_TRAIT(has_nested_result)
{
    template<class T>
    auto require(const T& x) -> valid<
        has_template<T::template result>
    >;
};
```

This trait will be true if `T` has a nested template called `result`.

## 4.4 is_true

The `is_true` query can check if a trait is true. Using *Refinements* is the preferred way of checking for additional traits, but this can be useful if the evaluation of some trait can't be used lazily with placeholder expression. So `is_true` can be used instead, for example:

```
TICK_TRAIT(is_2d_array)
{
    template<class T>
    auto require(const T& x) -> valid<
        is_true<std::is_same<std::rank<T>::type, std::integral_constant<std::size_t,␣
↪2>> >
    >;
};
```

## 4.5 is_false

The `is_false` query can check if a trait is false. Using *Refinements* is the preferred way of checking for additional traits, but this can be useful if the evaluation of some trait can't be used lazily with placeholder expression. So `is_false` can be used instead, for example:

```
TICK_TRAIT(is_multi_array)
{
    template<class T>
    auto require(const T& x) -> valid<
        is_false<std::is_same<std::rank<T>::type, std::integral_constant<std::size_t,␣
↪1>> >
    >;
};
```

## 4.6 as_const

The `as_const` function helps ensure that lvalues are `const`.

```
TICK_TRAIT(is_copy_assignable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x = as_const(x))
```

```
    >;
};
```

## 4.7 as_mutable

The `as_mutable` function helps ensure that lvalues are treated as `mutable` or non-const.

```
TICK_TRAIT(is_destructible)
{
    template<class T>
    auto require(const T& x) -> valid<
        decltype(as_mutable(x).~T())
    >;
};
```

CHAPTER 5

# Build traits without macros

The traits can be built without using the `TICK_TRAIT` macros. However, it may introduce problems with portability. So if only one platform is needed to be supported, then here's how to build them. First, build a class for the `require` functions and inherit from `tick::ops` to bring in all the query operations:

```
struct is_incrementable_r : tick::ops
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x++),
        decltype(++x)
    >;
};
```

Next, turn it into a trait using `tick::models` :

```
template<class... Ts>
struct is_incrementable
: tick::models<is_incrementable_r, Ts...>
{};
```

## 5.1 refines

Refinements can be used by using the `refines` class:

```
struct is_incrementable_r
: tick::ops, tick::refines<std::is_default_constructible<tick::_>>
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x++),
        decltype(++x)
    >;
};
```

Notice, the placeholders have to be fully qualified here.

CHAPTER 6

# Template constraints

Three macros are provided to help improve the readability of template constraints.

## 6.1 TICK_REQUIRES

The `TICK_REQUIRES` can be used on template parameters. For example,

```
template<class T, TICK_REQUIRES(is_incrementable<T>())>
void increment(T& x)
{
    x++;
}
```

## 6.2 TICK_CLASS_REQUIRES

The `TICK_CLASS_REQUIRES` can be used when template specialization is done on classes. For example,

```
template<class T, class=void>
struct foo
{
    ...
};

template<class T>
struct foo<T, TICK_CLASS_REQUIRES(is_incrementable<T>() and not std::is_integral<T>
↪())>
{
    ...
};

template<class T>
struct foo<T, TICK_CLASS_REQUIRES(std::is_integral<T>())>
{
    ...
};
```

## 6.3 TICK_MEMBER_REQUIRES

The `TICK_MEMBER_REQUIRES` can be used for member function inside of classes, that are not templated. For example,

```
template<class T>
struct foo
{
    T x;

    TICK_MEMBER_REQUIRES(is_incrementable<T>())
    void up()
    {
        x++;
    }
};
```

## 6.4 TICK_PARAM_REQUIRES

The `TICK_PARAM_REQUIRES` can be used in the paramater of the function. This is useful for lambdas:

```
auto increment = [](auto& x, TICK_PARAM_REQUIRES(is_incrementable<decltype(x)>()))
{
    x++;
};
```

Also, the `trait` function is provided which can be used to deduce the type of the parameters:

```
auto increment = [](auto& x, TICK_PARAM_REQUIRES(trait<is_incrementable>(x)))
{
    x++;
};
```

Note: The `trait` function always deduces the type without references. So `trait<std::is_lvalue_reference>(x)` will always be false.

## 6.5 TICK_FUNCTION_REQUIRES

The `TICK_FUNCTION_REQUIRES` can be used on functions. This requires placing parenthesis around the return type:

```
template<class T>
TICK_FUNCTION_REQUIRES(is_incrementable<T>())
(void) increment(T& x)
{
    x++;
}
```

Note: The `TICK_REQUIRES` should be preferred.

CHAPTER 7

# Tag dispatching

Tag dispatching allows for functions to be ordered by the refinements defined in the trait. For example, if we try to implement an `advance` function like `std::advance`. First, we can define the traits for the different traversals:

```
TICK_TRAIT(is_incrementable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x++),
        decltype(++x)
    >;
};

TICK_TRAIT(is_decrementable, is_incrementable<_>)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x--),
        decltype(--x)
    >;
};

TICK_TRAIT(is_advanceable, is_decrementable<_>)
{
    template<class T, class Number>
    auto require(T&& x, Number n) -> valid<
        decltype(x += n)
    >;
};
```

Then we can try to use template constraints for the different overloads:

```
template<class Iterator, TICK_REQUIRES(is_advanceable<Iterator>())>
void advance(Iterator& it, int n)
{
    it += n;
}

template<class Iterator, TICK_REQUIRES(is_decrementable<Iterator>())>
void advance(Iterator& it, int n)
{
    if (n > 0) while (n--) ++it;
    else
    {
```

```
        n *= -1;
        while (n--) --it;
    }
}

template<class Iterator, TICK_REQUIRES(is_incrementable<Iterator>())>
void advance(Iterator& it, int n)
{
    while (n--) ++it;
}
```

However, this leads to ambiguities when we try to use it with iterators to vectors. That is because those iterators are valid for all three overloads. So, tag dispatching allows us to pick the overload that is the most refined. First, we need to call `most_refined` which will retrieve the tags for each each trait. So then `advance` could be implemented like this:

```
template<class Iterator>
void advance_impl(Iterator& it, int n, tick::tag<is_advanceable>)
{
    it += n;
}

template<class Iterator>
void advance_impl(Iterator& it, int n, tick::tag<is_decrementable>)
{
    if (n > 0) while (n--) ++it;
    else
    {
        n *= -1;
        while (n--) --it;
    }
}

template<class Iterator>
void advance_impl(Iterator& it, int n, tick::tag<is_incrementable>)
{
    while (n--) ++it;
}

template<class Iterator>
void advance(Iterator& it, int n)
{
    advance_impl(it, n, tick::most_refined<is_advanceable<Iterator>>());
}
```

## 7.1 Specialization

Tag dispatching will still work with specialization. Say, for instance, someone implemented an iterator called `foo_iterator` that when the user called `+=` it would crash at runtime. So we would like to specialize `is_advanceable` to make it false, so the `+=` won't be called:

```
template<>
struct is_advanceable<foo_iterator>
: std::false_type
{};
```

So this will exclude the `is_advanceable` overload, but the `is_decrementable` and `is_incrementable` will still be called.

CHAPTER 8

# Traits

## 8.1 bare

### 8.1.1 Header

```
#include <tick/traits/bare.h>
```

### 8.1.2 Description

The `bare` trait remvoes both the reference and the `const` /`volatile` qualifiers.

### 8.1.3 Synopsis

```
template<class T>
struct bare;
```

## 8.2 is_allocator

### 8.2.1 Header

```
#include <tick/traits/is_allocator.h>
```

### 8.2.2 Description

An allocator encapsulates a memory allocation and deallocation strategy.

### 8.2.3 Requirements

Given:

- `T` , a cv-unqualified object type
- `A` , an Allocator type for type `T`

- a , an object of type A
- B , the corresponding allocator type for some cv-unqualified object type U (as obtained by rebinding A )
- ptr , a value of type allocator_traits<A>::pointer , obtained by calling allocator_traits<A>::allocate()
- cptr , a value of type allocator_traits<A>::const_pointer , obtained by conversion from ptr
- vptr , a value of type allocator_traits<A>::void_pointer , obtained by conversion from ptr
- cvptr , a value of type allocator_traits<A>::const_void_pointer , obtained by conversion from cptr or from vptr
- xptr , a dereferencable pointer to some cv-unqualified object type X
- n , a value of type allocator_traits<A>::size_type

| Expression | Requirements | Return type |
|---|---|---|
| `A::pointer` (optional) | Satisfies `is_nullable_pointer` and `is_random_access_iterator` | |
| `A::const_pointer` (optional) | `A::pointer` is convertible to `A::const_pointer`. Satisfies `is_nullable_pointer` and `is_random_access_iterator` | |
| `A::void_pointer` (optional) | `A::pointer` is convertible to `A::void_pointer` `B::void_pointer` and `A::void_pointer` are the same type. Satisfies `is_nullable_pointer` | |
| `A::const_void_pointer` (optional) | `A::pointer`, `A::const_pointer`, and `A::void_pointer` are convertible to `A::const_void_pointer` ‘‘ `B::const_void_pointer` ‘‘ and `A::const_void_pointer` ‘‘ are the same type. Satisfies `is_nullable_pointer` | |
| `A::value_type` | | the type `T` |
| `*ptr` | | `T&` |
| `*cptr` | `*cptr` and `*ptr` identify the same object | `const T&` |
| `ptr->m` | same as `(*ptr).m`, if `(*ptr).m` is well-defined | the type of `T::m` |
| `cptr->m` | same as `(*cptr).m`, if `(*cptr).m` is well-defined | the type of `T::m` |
| `static_cast <A::pointer> (vptr)` | `static_cast <A::pointer> (vptr) == ptr` | `A::pointer` |
| `static_cast <A::const_pointer>(cvptr)` | `static_cast <A::const_pointer>(vptr) == cptr` | `A::const_pointer` |
| `a.allocate(n)` | allocates storage suitable for `n` objects of type `T`, but does not construct them. May throw exceptions. | `A::pointer` |
| `a.deallocate(ptr,n)` | deallocates storage pointed to `ptr`, which must be a value returned by a previous call to allocate that has not been invalidated by an intervening call to `deallocate`. `n` must match the value previously passed to `allocate`. Does not throw exceptions. | (not used) |
| `a.max_size ()` (optional) | the largest value that can be passed to `A::allocate()` | `A::size_type` |
| `a1 == a2` | returns true only if the storage allocated by the allocator `a1` can be deallocated through `a2`. Establishes reflexive, symmetric, and transitive relationship. Does not throw exceptions. | `bool` |
| `a1 != a2` | same as `!(a1==a2)` | `bool` |
| `A a1(a) A a1 = a` | Copy-constructs `a1` such that `a1 == a`. Does not throw exceptions. (Note: every allocator also satisfies `is_copy_constructible` | |
| `A a(b)` | Constructs `a` such that `B(a)==b` and `A(b)==a`. Does not throw exceptions. (Note: this implies that all allocators related by rebind maintain each other’s resources, such as memory pools) | |
| `A a1(std::move(a)) A a1 = std::move(a)` | Constructs `a1` such that it equals the prior value of `a`. Does not throw exceptions. | |
| `A a(std::move(b))` | Constructs `a` such that it equals the prior value of `A(b)`. Does not throw exceptions. | |

### 8.2.4 Synopsis

```
TICK_TRAIT(is_allocator,
    is_copy_constructible<_>,
    is_equality_comparable<_>
)
{
    template<class A>
    auto require(const A& a)-> valid<
        decltype(returns<is_nullable_pointer<_>, is_random_access_iterator<_>>(as_
↪mutable(a).allocate(std::declval<std::size_t>()))),
        decltype(returns<typename A::value_type&>(*(as_mutable(a).allocate(std::
↪declval<std::size_t>()))))
    >;
};
```

## 8.3 is_associative_container

### 8.3.1 Header

```
#include <tick/traits/is_associative_container.h>
```

### 8.3.2 Description

An associative container is an ordered container that provides fast lookup of objects based on keys.

### 8.3.3 Synopsis

```
TICK_TRAIT(is_associative_container, is_container<_>)
{
    template<class T>
    auto require(const T& x) -> valid<
        has_type<typename T::key_type, is_destructible<_>>,
        has_type<typename T::key_compare, is_compare<_, typename T::key_type>>,
        has_type<typename T::value_compare, is_compare<_, typename T::value_type>>,
        decltype(returns<typename T::key_compare>(x.key_comp())),
        decltype(returns<typename T::value_compare>(x.value_comp()))
    >;
};
```

## 8.4 is_bidirectional_iterator

### 8.4.1 Header

```
#include <tick/traits/is_bidirectional_iterator.h>
```

### 8.4.2 Description

A bidirectional iterator is a forward iterator that can be moved in both directions (i.e. incremented and decremented).

### 8.4.3 Requirements

The type `It` satisfies `is_bidirectional_iterator` if

- The type It satisfies *is_forward_iterator*

And, given

- `a` and `b`, iterators of type `It`
- `reference`, the type denoted by `std::iterator_traits<It>::reference`

The following expressions must be valid and have their specified effects

| Expression | Return | Equivalent expression |
|---|---|---|
| `--a` | `It&` | |
| `a--` | convertible to `It&` | `It temp = a; --a; return temp;` |
| `*a--` | `reference` | |

### 8.4.4 Synopsis

```
TICK_TRAIT(is_bidirectional_iterator, is_forward_iterator<_>)
{
    template<class I>
    auto require(I&& i) -> valid<
        decltype(returns<typename std::add_lvalue_reference<I>::type>(--i)),
        decltype(returns<I>(i--)),
        decltype(returns<typename iterator_traits<I>::reference>(*i--))
    >;
};
```

## 8.5 is_compare

### 8.5.1 Header

```
#include <tick/traits/is_compare.h>
```

### 8.5.2 Description

A compare function is a function that is used to establish a relationship.

### 8.5.3 Synopsis

```
TICK_TRAIT(is_compare)
{
    template<class F, class T>
    auto require(F&& f, T&& x) -> valid<
        decltype(returns<bool>(f(std::forward<T>(x), std::forward<T>(x)))
    >;

    template<class F, class T, class U>
    auto require(F&& f, T&& x, U&& y) -> valid<
        decltype(require(std::forward<F>(f), std::forward<T>(x))),
        decltype(require(std::forward<F>(f), std::forward<U>(y))),
        decltype(returns<bool>(f(std::forward<T>(x), std::forward<U>(y)))),
        decltype(returns<bool>(f(std::forward<U>(y), std::forward<T>(x)))
    >;
};
```

## 8.6 is_container

### 8.6.1 Header

```
#include <tick/traits/is_container.h>
```

### 8.6.2 Description

A container is an object used to store other objects and takes care of the management of the memory used by the objects it contains.

### 8.6.3 Synopsis

```
TICK_TRAIT(is_container,
    is_equality_comparable<_>,
    is_default_constructible<_>,
    is_copy_constructible<_>,
    is_copy_assignable<_>,
    is_destructible<_>,
    is_swappable<_>,
    is_range<_>,
    /* is_copy_insertable_container<_> */
)
{
    template<class T>
    auto require(const T& x) -> valid<
        has_type<typename T::iterator, is_iterator<_>>,
        has_type<typename T::const_iterator, is_iterator<_>>,
        has_type<typename T::value_type>,
        has_type<typename T::reference, typename T::value_type&>,
        has_type<typename T::const_reference, const typename T::value_type&>,
        has_type<typename T::difference_type, int>,
        has_type<typename T::size_type, unsigned>,
        decltype(returns<typename T::size_type>(x.size())),
        decltype(returns<typename T::size_type>(x.max_size())),
```

```
        decltype(returns<bool>(x.empty()))
    >;
};
```

## 8.7 is_copy_assignable

### 8.7.1 Header

```
#include <tick/traits/is_copy_assignable.h>
```

### 8.7.2 Description

Checks if type `T` can be copy-assigned from an lvalue.

### 8.7.3 Synopsis

```
TICK_TRAIT(is_copy_assignable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x = as_const(x))
    >;
};
```

## 8.8 is_copy_constructible

### 8.8.1 Header

```
#include <tick/traits/is_copy_constructible.h>
```

### 8.8.2 Description

Checks if type `T` can be copy-constructed from an lvalue.

### 8.8.3 Synopsis

```
TICK_TRAIT(is_copy_constructible)
{
    template<class T>
    auto require(const T& x) -> valid<
        decltype(T(x))
    >;
};
```

## 8.9 is_copy_insertable

### 8.9.1 Header

```
#include <tick/traits/is_copy_insertable.h>
```

### 8.9.2 Description

If a type can be copy-constructed in-place by a given allocator.

### 8.9.3 Synopsis

```
TICK_TRAIT(is_copy_insertable,
    is_move_insertable<_1, _2>
)
{
    template<class C, class T>
    auto require(const C&, const T&) -> valid<
        is_true<is_emplace_constructible<C, T, T>>
    >;
};
```

## 8.10 is_default_constructible

### 8.10.1 Header

```
#include <tick/traits/is_default_constructible.h>
```

### 8.10.2 Description

Checks if a type `T` is default constructible.

### 8.10.3 Synopsis

```
TICK_TRAIT(is_default_constructible)
{
    template<class T>
    auto require(const T&) -> valid<
        decltype(T())
    >;
};
```

## 8.11 is_destructible

### 8.11.1 Header

```
#include <tick/traits/is_destructible.h>
```

### 8.11.2 Description

Checks if type `T` can be destructed.

### 8.11.3 Synopsis

```
TICK_TRAIT(is_destructible)
{
    template<class T>
    auto require(const T& x) -> valid<
        decltype(as_mutable(x).~T())
    >;
};
```

## 8.12 is_emplace_constructible

### 8.12.1 Header

```
#include <tick/traits/is_emplace_constructible.h>
```

### 8.12.2 Description

If a type can be constructed from a given set of arguments in uninitialized storage by a given allocator.

### 8.12.3 Synopsis

```
TICK_TRAIT(is_emplace_constructible)
{
    template<class C, class T, class... Ts>
    auto require(const C& c, const T&, Ts&&...) -> valid<
        has_type<typename C::allocator_type, is_allocator<_>>,
        decltype(returns<typename C::allocator_type>(c.get_allocator())),
        decltype(
            std::allocator_traits<typename C::allocator_type>::construct(
                as_mutable(c.get_allocator()),
                std::declval<T*>(),
                std::declval<Ts>()...
            )
        )
    >;
};
```

## 8.13 is_equality_comparable

### 8.13.1 Header

```
#include <tick/traits/is_equality_comparable.h>
```

### 8.13.2 Description

Checks if type `T` has operator `==` and `!=` and is convertible to `bool` .

### 8.13.3 Requirements

The type `T` satisfies `is_equality_comparable` if

Given:

- a, and b expressions of type `T` or `const T`

The following expressions must be valid:

| Expression | Return type |
|---|---|
| `a == b` | implicitly convertible to `bool` |
| `a != b` | implicitly convertible to `bool` |

### 8.13.4 Synopsis

```
TICK_TRAIT(is_equality_comparable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(returns<bool>(x == x)),
        decltype(returns<bool>(x != x))
    >;

    template<class T, class U>
    auto require(T&& x, U&& y) -> valid<
        decltype(require(std::forward<T>(x))),
        decltype(require(std::forward<U>(y))),
        decltype(returns<bool>(x == y)),
        decltype(returns<bool>(x != y)),
        decltype(returns<bool>(y == x)),
        decltype(returns<bool>(y != x))
    >;
};
```

## 8.14 is_erasable

### 8.14.1 Header

```
#include <tick/traits/is_erasable.h>
```

### 8.14.2 Description

If a type can be destroyed by an allocator.

### 8.14.3 Synopsis

```
TICK_TRAIT(is_erasable)
{
    template<class C, class T>
    auto require(const C&, const T&) -> valid<
        has_type<typename C::allocator_type, is_allocator<_>>,
        decltype(returns<typename C::allocator_type&>(C::get_allocator())),
        decltype(
            std::allocator_traits<typename C::allocator_type>::destroy(
                C::get_allocator(),
                std::declval<T*>()
            )
        )
    >;
};
```

## 8.15 is_forward_iterator

### 8.15.1 Header

```
#include <tick/traits/is_forward_iterator.h>
```

### 8.15.2 Description

A forward iterator is an iterator that can read data from the pointed-to element.

### 8.15.3 Requirements

The type `It` satisfies `is_forward_iterator` if:

- The type `It` *is_input_iterator*
- The type `It` *is_default_constructible*

The type `std::iterator_traits<It>::reference` must be exactly

- `T&` if `It` satisfies OutputIterator (`It` is mutable)

- `const T&` otherwise (`It` is constant),

(where `T` is the type denoted by `std::iterator_traits<It>::value_type`)

And, given

- `i`, dereferenceable iterator of type `It`
- `reference`, the type denoted by `std::iterator_traits<It>::reference`

The following expressions must be valid and have their specified effects

| Expression | Return | Equivalent expression |
|---|---|---|
| `i++` | `It` | `It ip=i; ++i; return ip;` |
| `*i++` | `reference` | |

### 8.15.4 Synopsis

```
TICK_TRAIT(is_forward_iterator, is_input_iterator<_>, is_default_constructible<_>)
{
    template<class I>
    auto require(I&& i) -> valid<
        decltype(returns<typename std::add_lvalue_reference<I>::type>(++i)),
        decltype(returns<I>(i++)),
        is_true<std::is_reference<typename iterator_traits<I>::reference>>,
        decltype(returns<typename iterator_traits<I>::value_type&>(*i)),
        decltype(returns<typename iterator_traits<I>::reference>(*i++))
    >;
};
```

## 8.16 is_input_iterator

### 8.16.1 Header

```
#include <tick/traits/is_input_iterator.h>
```

### 8.16.2 Description

An input iterator is an iterator that can read from the pointed-to element. Input iterators only guarantee validity for single pass algorithms: once an inputIterator `i` has been incremented, all copies of its previous value may be invalidated.

### 8.16.3 Requirements

The type `It` satisfies InputIterator if

- The type `It` satisfies *is_iterator*
- The type `It` satisfies *is_equality_comparable*

And, given

- `i` and `j`, values of type `It` or const It

- `reference` , the type denoted by `std::iterator_traits<It>::reference`
- `value_type` , the type denoted by `std::iterator_traits<It>::value_type`

The following expressions must be valid and have their specified effects

| Expression | Return | Equivalent expression | Notes |
|---|---|---|---|
| `i != j` | contextually convertible to `bool` | `!(i == j)` | **Precondition**: `(i, j)` is in the domain of `==` . |
| `*i` | reference, convertible to `value_type` | If `i == j` and `(i, j)` is in the domain of `==` then this is equivalent to `*j` . | **Precondition**: `i` is dereferenceable. The expression `(void)*i` , `*i` is equivalent to `*i` . |
| `i->m` | | `(*i).m` | **Precondition**: `i` is dereferenceable. |
| `++i` | `It&` | | **Precondition**: `i` is dereferenceable. **Postcondition**: `i` is dereferenceable or `i` is past-the-end. **Postcondition**: Any copies of the previous value of `i` are no longer required to be either dereferenceable or to be in the domain of `==` . |
| `(void)i++` | | `(void)++i` | |
| `*i++` | convertible to `value_type` | `value_type x = *i; ++i; return x;` | |

### 8.16.4 Synopsis

```
TICK_TRAIT(is_input_iterator,
    is_iterator<_>,
    is_equality_comparable<_>
)
{
    template<class I>
    auto require(I&& i) -> valid<
        decltype(returns<typename iterator_traits<I>::value_type>(*i)),
        decltype(returns<typename iterator_traits<I>::value_type>(*i++))
    >;
};
```

## 8.17 is_iterator

### 8.17.1 Header

```
#include <tick/traits/is_iterator.h>
```

### 8.17.2 Description

An iterator can be used to identify and traverse the elements of a container.

### 8.17.3 Requirements

The type `It` satisfies `is_iterator` if

- The type `It` satisfies *is_copy_constructible*
- The type `It` satisfies *is_copy_constructible*
- The type `It` satisfies *is_destructible*
- lvalues of type `It` satisfy *is_destructible*

and given:

- `r`, an lvalue of type `It`.

The following expressions must be valid and have their specified effects:

| Expression | Return Type | Precondition |
|---|---|---|
| `*r` | unspecified | `r` is *dereferenceable* |
| `++r` | `It&` | `r` is *incrementable* (the behavior of the expression ++r is defined) |

### 8.17.4 Synopsis

```
TICK_TRAIT(is_iterator,
    is_copy_constructible<_>,
    is_copy_assignable<_>,
    is_destructible<_>,
    is_swappable<_>
)
{
    template<class I>
    auto require(I&& i) -> valid<
        decltype(returns<typename std::iterator_traits<I>::reference>(*i))
        decltype(returns<typename std::add_lvalue_reference<I>::type>(++i))
    >;
};
```

## 8.18 is_less_than_comparable

### 8.18.1 Header

```
#include <tick/traits/is_less_than_comparable.h>
```

### 8.18.2 Description

Checks if type `T` has operator `<` and is convertible to `bool`.

### 8.18.3 Requirements

The type `T` satisfies `is_less_than_comparable` if

Given:

- a, and b expressions of type T or const T

The following expressions must be valid:

| Expression | Return type |
|---|---|
| `a < b` | implicitly convertible to `bool` |

### 8.18.4 Synopsis

```
TICK_TRAIT(is_less_than_comparable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(returns<bool>(x < x))
    >;

    template<class T, class U>
    auto require(T&& x, U&& y) -> valid<
        decltype(returns<bool>(x < x)),
        decltype(returns<bool>(y < y)),
        decltype(returns<bool>(x < y)),
        decltype(returns<bool>(y < x))
    >;
};
```

## 8.19 is_move_assignable

### 8.19.1 Header

```
#include <tick/traits/is_move_assignable.h>
```

### 8.19.2 Description

Checks if type `T` can be assigned from an rvalue.

### 8.19.3 Synopsis

```
TICK_TRAIT(is_move_assignable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x = std::move(x))
    >;
};
```

## 8.20 is_move_constructible

### 8.20.1 Header

```
#include <tick/traits/is_move_constructible.h>
```

### 8.20.2 Description

Checks if type `T` can be constructed from an rvalue.

### 8.20.3 Synopsis

```
TICK_TRAIT(is_move_constructible)
{
    template<class T>
    auto require(const T& x) -> valid<
        decltype(T(std::move(x)))
    >;
};
```

## 8.21 is_move_insertable

### 8.21.1 Header

```
#include <tick/traits/is_move_insertable.h>
```

### 8.21.2 Description

If a type can be constructed into uninitialized storage from an rvalue of that type by a given allocator.

### 8.21.3 Synopsis

```
template<class C, class T>
struct is_move_insertable
: is_emplace_constructible<C, T, T&&>
{};
```

## 8.22 is_nullable_pointer

### 8.22.1 Header

```
#include <tick/traits/is_nullable_pointer.h>
```

### 8.22.2 Description

Checks if type `T` is a pointer-like object which can be compared to `std::nullptr_t` objects.

### 8.22.3 Requirements

The type must meet all of the following requirements:

- *is_equality_comparable*
- *is_default_constructible*
- *is_copy_constructible*
- *is_copy_assignable*
- *is_destructible*

The type `T` must satisfy the following additional expressions, given two values `p` and `q` that are of type `T`, and that `np` is a value of `std::nullptr_t` type:

| Expression | Effects |
|---|---|
| `T p(np);` | Afterwards, p is equivalent to `nullptr` |
| `T(np)` | a temporary object that is equivalent to `nullptr` |
| `p = np` | Must return a `T&`, and afterwards, `p` is equivalent to `nullptr` |
| `p != q` | Must return a value that is contextually convertible to `bool`, The effect is `!(p == q)` |
| `p == np` | Must return a value that is contextually convertible to `bool`. The effect is `(p == T())` |
| `np == p` | Must return a value that is contextually convertible to `bool`. The effect is `(p == T())` |
| `p != np` | Must return a value that is contextually convertible to `bool`. The effect is `!(p == np)` |
| `np != p` | Must return a value that is contextually convertible to `bool`. The effect is `!(p == np)` |

### 8.22.4 Synopsis

```
TICK_TRAIT(is_nullable_pointer,
    is_equality_comparable<_>,
    is_default_constructible<_>,
    is_copy_constructible<_>,
    is_copy_assignable<_>,
    is_destructible<_>,
    std::is_constructible<_, std::nullptr_t>,
    is_equality_comparable<_, std::nullptr_t>
)
{};
```

## 8.23 is_output_iterator

### 8.23.1 Header

```
#include <tick/traits/is_output_iterator.h>
```

### 8.23.2 Description

An output iterator is an iterator that can write to the pointed-to element.

### 8.23.3 Requirements

The type `X` satisfies `is_output_itearator` if

- The type `X` satisfies *is_iterator*
- `X` is a class type or a pointer type

And, given

- `o`, a value of some type that is writable to the output iterator (there may be multiple types that are writable, e.g. if `operator=` may be a template. There is no notion of `value_type` as for the input iterators)
- r, an lvalue of type `X`,

The following expressions must be valid and have their specified effects:

| Expression | Return | Equivalent expression | Pre-condition | Post-conditions | Notes |
|---|---|---|---|---|---|
| `*r = o` | (not used) | | `r` is dereferencable | `r` is incrementable | After this operation `r` is not required to be dereferenceable and any copies of the previous value of `r` are no longer required to be dereferenceable or incrementable. |
| `++r` | `X&` | | `r` is incrementable | `&r == &++r`, `r` is dereferencable or past-the-end | After this operation `r` is not required to be incrementable and any copies of the previous value of `r` are no longer required to be dereferenceable or incrementable. |
| `r++` | convertible to `const X&` | `X temp = r; ++r; return temp;` | | | |
| `*r++ = o` | (not used) | `*r = o; ++r;` | | | |

### 8.23.4 Synopsis

```
TICK_TRAIT(is_output_iterator, is_iterator<_>)
{
    template<class I, class T>
    static auto require(I&& i, T&& x) -> valid<
        decltype(*i = x),
        decltype(*i++ = x),
        decltype(returns<typename std::add_const<typename std::add_lvalue_reference<I>
↪::type>::type>(++i))
    >;

    template<class I>
    auto require(I&& i) -> valid<
        decltype(require(std::forward<I>(i), std::declval<typename iterator_traits<I>:
↪:value_type>()))
```

```
    >;
};
```

## 8.24 is_pod

### 8.24.1 Header

```
#include <tick/traits/is_pod.h>
```

### 8.24.2 Description

If a type is a POD (Plain Old Data) type. This means the type is compatible with the types used in the C programming language, can be manipulated using C library functions: it can be created with std::malloc, it can be copied with std::memmove, etc, and can be exchanged with C libraries directly, in its binary form.

## 8.25 is_predicate

### 8.25.1 Header

```
#include <tick/traits/is_predicate.h>
```

### 8.25.2 Description

A predicate is a function object that returns a boolean.

### 8.25.3 Synopsis

```
TICK_TRAIT(is_predicate)
{
    template<class F, class... Ts>
    auto require(F&& f, Ts&&... xs) -> valid<
        decltype(returns<bool>(f(std::forward<Ts>(xs)...)))
    >;
};
```

## 8.26 is_random_access_iterator

### 8.26.1 Header

```
#include <tick/traits/is_random_access_iterator.h>
```

### 8.26.2 Description

A random access iterator is a bidirectional iterator that can be moved to point to any element in constant time.

### 8.26.3 Requirements

The type `It` satisfies `is_random_access_iterator` if

- The type `It` satisfies *is_bidirectional_iterator*
- The type `It` satisfies *is_totally_ordered*

And, given

- `value_type` , the type denoted by `std::iterator_traits<It>::value_type`
- `difference_type` , the type denoted by `std::iterator_traits<It>::difference_type`
- `reference` , the type denoted by `std::iterator_traits<It>::reference`
- `i` , `a` , `b` , objects of type `It` or `const It`
- `r` , a value of type `It&`
- `n` , an integer of type `difference_type`

The following expressions must be valid and have their specified effects

| Expression | Return type |
|---|---|
| `r += n` | `It&` |
| `a + n,n + a` | `It` |
| `r -= n` | `It&` |
| `i -n` | `It` |
| `b -a` | `difference_type` |
| `i[n]` | convertible to `reference` |

### 8.26.4 Synopsis

```
TICK_TRAIT(is_random_access_iterator,
    is_bidirectional_iterator<_>,
    is_totally_ordered<_>
)
{
    template<class I, class Number>
    auto require(I&& i, Number n) -> valid<
        decltype(returns<typename std::add_lvalue_reference<I>::type>(i += n)),
        decltype(returns<typename std::add_lvalue_reference<I>::type>(i -= n)),
        decltype(returns<I>(i + n)),
        decltype(returns<I>(i - n)),
        decltype(returns<I>(n + i)),
        decltype(returns<typename iterator_traits<I>::difference_type>(i - i)),
        decltype(returns<typename iterator_traits<I>::reference>(i[n])),
        decltype(returns<typename iterator_traits<I>::reference>(*(i + n)))
    >;

    template<class I>
    auto require(I&& i) -> valid<
        decltype(require(std::forward<I>(i), 0))
```

```
    >;
};
```

## 8.27 is_range

### 8.27.1 Header

```
#include <tick/traits/is_range.h>
```

### 8.27.2 Description

Checks if the type provides begin and end iterators that can be used with a for-range loop.

### 8.27.3 Synopsis

```
TICK_TRAIT(is_range)
{
    using std::begin;
    using std::end;
    template<class T>
    auto require(T&& x) -> valid<
        decltype(returns<is_iterator<_>>(begin(std::forward<T>(x)))),
        decltype(returns<is_iterator<_>>(end(std::forward<T>(x))))
    >;
};
```

## 8.28 is_standard_layout

### 8.28.1 Header

```
#include <tick/traits/is_standard_layout.h>
```

### 8.28.2 Description

A standard layout types are useful for communicating with code written in other programming languages.

## 8.29 is_swappable

### 8.29.1 Header

```
#include <tick/traits/is_swappable.h>
```

### 8.29.2 Description

Checks if type `T` can be called with either `std::swap` or an ADL overload of `swap`.

### 8.29.3 Requirements

Type `U` is swappable with type `T` if, for any object `u` of type `U` and any object `t` of type `T`

| Expression | Requirements |
|---|---|
| `using std::swap; swap(t,u);` | After the call, the value of t is the value held by u before the call, and the value of u is the value held by t before the call. |
| `using std::swap; swap(u,t);` | After the call, the value of t is the value held by u before the call, and the value of u is the value held by t before the call. |

### 8.29.4 Synopsis

```
TICK_TRAIT(is_swappable)
{
    using std::swap;
    template<class T>
    auto require(T&& x) -> valid<
        decltype(swap(x, x))
    >;

    template<class T, class U>
    auto require(T&& x, U&& y) -> valid<
        decltype(swap(x, y))
        decltype(swap(y, x))
    >;
};
```

## 8.30 is_totally_ordered

### 8.30.1 Header

```
#include <tick/traits/is_totally_ordered.h>
```

### 8.30.2 Description

Checks if type `T` has operators for ordering and equality, and they are convertible to `bool`.

### 8.30.3 Requirements

The type `T` satisfies `is_totally_ordered` if

- The type `T` satisfies *is_weakly_ordered*
- The type `T` satisfies *is_equality_comparable*

### 8.30.4 Synopsis

```
TICK_TRAIT(is_totally_ordered,
    quote<is_weakly_ordered>,
    quote<is_equality_comparable>
)
{};
```

## 8.31 is_trivial

### 8.31.1 Header

```
#include <tick/traits/is_trivial.h>
```

### 8.31.2 Description

If a type is trivial.

## 8.32 is_trivially_copyable

### 8.32.1 Header

```
#include <tick/traits/is_trivially_copyable.h>
```

### 8.32.2 Description

If a type is trivially copyable.

## 8.33 is_value_swappable

### 8.33.1 Header

```
#include <tick/traits/is_value_swappable.h>
```

### 8.33.2 Description

Checks if `T` two objects of type `T` can be dereferenced and the resulting values satisify `is_swappable`.

### 8.33.3 Requirements

Type T satisfies `is_value_swappable` if

- Type T satisfies the requirements for `is_iterator`
- For any dereferencable object `x` of type `T`, `*x` satisfies the requirements for `is_swappable`

### 8.33.4 Synopsis

```
TICK_TRAIT(is_value_swappable, is_iterator<_>)
{
    template<class T>
    auto require(T&&) -> valid<
        is_true<is_swappable<typename iterator_traits<T>::value_type>>
    >;
};
```

## 8.34 is_weakly_ordered

### 8.34.1 Header

```
#include <tick/traits/is_weakly_ordered.h>
```

### 8.34.2 Description

Checks if type `T` has order operators such as `>`, `<`, `>=` and `<=` and they are convertible to `bool`.

### 8.34.3 Requirements

The type `T` satisfies `is_weakly_ordered` if

Given:

- a, and b expressions of type `T` or `const T`

The following expressions must be valid:

| Expression | Return type |
|---|---|
| `a > b` | implicitly convertible to `bool` |
| `a > b` | implicitly convertible to `bool` |
| `a >= b` | implicitly convertible to `bool` |
| `a <= b` | implicitly convertible to `bool` |

### 8.34.4 Synopsis

```
TICK_TRAIT(is_weakly_ordered)
{
    template<class T>
    auto require(T&& x) -> valid<
```

```
        decltype(returns<bool>(x < x)),
        decltype(returns<bool>(x > x)),
        decltype(returns<bool>(x <= x)),
        decltype(returns<bool>(x >= x))
    >;

    template<class T, class U>
    auto require(T&& x, U&& y) -> valid<
        decltype(require(std::forward<T>(x))),
        decltype(require(std::forward<U>(y))),
        decltype(returns<bool>(x < y)),
        decltype(returns<bool>(y < x)),

        decltype(returns<bool>(x > y)),
        decltype(returns<bool>(y > x)),

        decltype(returns<bool>(x <= y)),
        decltype(returns<bool>(y <= x)),

        decltype(returns<bool>(x >= y)),
        decltype(returns<bool>(y >= x))
    >;
};
```

CHAPTER 9

# Design Notes

## 9.1 Using template class

Tick uses the `valid` template class to place valid expressions, because it provides a more robust solution. Ideally, using tick we could define the traits like this:

```
TICK_TRAIT(is_incrementable)
{
    template<class T>
    auto require(T&& x) -> decltype(
        x++,
        ++x
    );
};
```

However, if one of the expressions returns a type that overloads the comma operator in a strange way(rare but still possible), then the trait could fail(ie return false when it should be true). To fix it, we could add `void` casts like this:

```
TICK_TRAIT(is_incrementable)
{
    template<class T>
    auto require(T&& x) -> decltype(
        (void)x++,
        (void)++x
    );
};
```

However, the `void` casts can be easy to forget. Another solution to the problem could be to pass it to a function:

```
template<class... Ts>
void valid_expr(T&&...);

TICK_TRAIT(is_incrementable)
{
    template<class T>
    auto require(T&& x) -> decltype(valid_expr(
        x++,
        ++x
    ));
};
```

However, if one of the expressions returns `void` , then this will fail as well(ie return false when it should be true). So this could be fixed in a similiar way as well:

```
TICK_TRAIT(is_incrementable)
{
    template<class T>
    auto require(T&& x) -> decltype(valid_expr(
        (x++, 1),
        (++x, 1)
    ));
};
```

However, it can be easy to forget to put the `1` in there. So instead we use a `valid` template class, like this:

```
TICK_TRAIT(is_incrementable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x++),
        decltype(++x)
    >;
};
```

This requires placing each expression in a decltype, but if this was forgotten there would be a compile error pointing to the incorrect expression.

## 9.2 Trait-based

The concept predicates in Tick are defined as regular type traits(ie they are integrel constants) instead of a `constexpr bool` function. They are almost functionally the same in use. However, as a trait, it allows for better flexibility and expressiveness, through higher-order programming. This is what enables passing the traits to other functions which can be used to match return types to traits as well as other types.

## 9.3 Specializations

All the traits created can be specialized by the user. This is very important. Since the definition of traits relies on duck typing, there are times that even though it may quack like a duck it is not a duck. So with specialization the user can clarify the type's capabilities.

CHAPTER 10

# ZLang support

ZLang is supported for some of the macros. The macros are in the `tick` namespace. For example,

```
$(trait is_incrementable)
{
    template<class T>
    auto require(T&& x) -> valid<
        decltype(x++),
        decltype(++x)
    >;
};
```

CHAPTER 11

# Acknowledgments


- Eric Niebler for the idea of using a `requires` member function to check concept traits.[http://ericniebler.com/2013/11/23/concept-checking-in-c11/]
- Jamboree for the idea of using a template class to place expressions.[https://github.com/ericniebler/range-v3/issues/29#issuecomment-51016277]

CHAPTER 12

# Search

- *Search*