

Writing Regexprs 2021-22 / Utils

Generated by Doxygen 1.9.3

1 Namespace Index	1
1.1 Namespace List	1
2 Hierarchical Index	3
2.1 Class Hierarchy	3
3 Class Index	5
3.1 Class List	5
4 File Index	7
4.1 File List	7
5 Namespace Documentation	9
5.1 wr22 Namespace Reference	9
5.2 wr22::utils Namespace Reference	9
5.2.1 Function Documentation	10
5.2.1.1 Box() [1/2]	10
5.2.1.2 Box() [2/2]	10
5.2.1.3 operator!=(()) [1/2]	10
5.2.1.4 operator!=(()) [2/2]	10
5.2.1.5 operator==(()) [1/2]	10
5.2.1.6 operator==(()) [2/2]	11
5.3 wr22::utils::detail Namespace Reference	11
5.4 wr22::utils::detail::adt Namespace Reference	11
6 Class Documentation	13
6.1 wr22::utils::Adt< Variants > Class Template Reference	13
6.1.1 Detailed Description	14
6.1.2 Member Typedef Documentation	14
6.1.2.1 VariantType	14
6.1.3 Constructor & Destructor Documentation	14
6.1.3.1 Adt()	14
6.1.4 Member Function Documentation	14
6.1.4.1 as_variant() [1/2]	15
6.1.4.2 as_variant() [2/2]	15
6.1.4.3 visit() [1/2]	15
6.1.4.4 visit() [2/2]	15
6.1.5 Member Data Documentation	16
6.1.5.1 m_variant	16
6.2 wr22::utils::Box< T > Class Template Reference	16
6.2.1 Detailed Description	16
6.2.2 Constructor & Destructor Documentation	17
6.2.2.1 Box() [1/3]	17
6.2.2.2 Box() [2/3]	17

6.2.2.3 Box() [3/3]	17
6.2.3 Member Function Documentation	18
6.2.3.1 construct_in_place()	18
6.2.3.2 operator*() [1/2]	18
6.2.3.3 operator*() [2/2]	18
6.3 wr22::utils::BoxIsEmpty Struct Reference	18
6.3.1 Member Function Documentation	19
6.3.1.1 what()	19
6.4 wr22::utils::detail::adt::MultiCallable< Fs > Struct Template Reference	19
6.4.1 Constructor & Destructor Documentation	19
6.4.1.1 MultiCallable()	20
6.5 wr22::utils::NonconcurrentSemaphore Class Reference	20
6.5.1 Constructor & Destructor Documentation	20
6.5.1.1 NonconcurrentSemaphore()	20
6.5.2 Member Function Documentation	20
6.5.2.1 enter()	20
6.5.3 Friends And Related Function Documentation	20
6.5.3.1 NonconcurrentSemaphoreGuard	21
6.6 wr22::utils::NonconcurrentSemaphoreGuard Class Reference	21
6.6.1 Constructor & Destructor Documentation	21
6.6.1.1 NonconcurrentSemaphoreGuard() [1/2]	21
6.6.1.2 NonconcurrentSemaphoreGuard() [2/2]	21
6.6.1.3 ~NonconcurrentSemaphoreGuard()	22
6.6.2 Member Function Documentation	22
6.6.2.1 operator=() [1/2]	22
6.6.2.2 operator=() [2/2]	22
6.6.3 Friends And Related Function Documentation	22
6.6.3.1 NonconcurrentSemaphore	22
7 File Documentation	23
7.1 include/wr22/utils/adt.hpp File Reference	23
7.2 adt.hpp	24
7.3 include/wr22/utils/box.hpp File Reference	24
7.4 box.hpp	25
7.5 include/wr22/utils/nonconcurrent_semaphore.hpp File Reference	26
7.6 nonconcurrent_semaphore.hpp	26
7.7 src/box.cpp File Reference	27
7.8 src/nonconcurrent_semaphore.cpp File Reference	27
Index	29

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

wr22	9
wr22::utils	9
wr22::utils::detail	11
wr22::utils::detail::adt	11

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

wr22::utils::Adt< Variants >	13
wr22::utils::Box< T >	16
std::exception	
wr22::utils::BoxIsEmpty	18
wr22::utils::NonconcurrentSemaphore	20
wr22::utils::NonconcurrentSemaphoreGuard	21
wr22::utils::detail::adt::Fs	
wr22::utils::detail::adt::MultiCallable< Fs >	19

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

wr22::utils::Adt< Variants >	
A helper class that simplifies creation of algebraic data types	13
wr22::utils::Box< T >	
A copyable and equality-comparable wrapper around <code>std::unique_ptr</code>	16
wr22::utils::BoxIsEmpty	18
wr22::utils::detail::adt::MultiCallable< Fs >	19
wr22::utils::NonconcurrentSemaphore	20
wr22::utils::NonconcurrentSemaphoreGuard	21

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

include/wr22/utils/ adt.hpp	23
include/wr22/utils/ box.hpp	24
include/wr22/utils/ nonconcurrent_semaphore.hpp	26
src/ box.cpp	27
src/ nonconcurrent_semaphore.cpp	27

Chapter 5

Namespace Documentation

5.1 wr22 Namespace Reference

Namespaces

- namespace [utils](#)

5.2 wr22::utils Namespace Reference

Namespaces

- namespace [detail](#)

Classes

- class [Adt](#)
A helper class that simplifies creation of algebraic data types.
- class [Box](#)
A copyable and equality-comparable wrapper around `std::unique_ptr`.
- struct [BoxIsEmpty](#)
- class [NonconcurrentSemaphore](#)
- class [NonconcurrentSemaphoreGuard](#)

Functions

- template<typename... Variants>
bool [operator==](#) (const [Adt](#)< Variants... > &lhs, const [Adt](#)< Variants... > &rhs)
Compare two compatible ADTs for equality.
- template<typename... Variants>
bool [operator!=](#) (const [Adt](#)< Variants... > &lhs, const [Adt](#)< Variants... > &rhs)
Compare two compatible ADTs for non-equality.
- template<typename T>
[Box](#) (T &&value) -> [Box](#)< T >
Type deduction guideline for [Box](#) (value initialization).
- template<typename T>
[Box](#) (std::unique_ptr< T > ptr) -> [Box](#)< T >
Type deduction guideline for [Box](#) (std::unique_ptr adoption).
- template<typename T, typename U>
bool [operator==](#) (const [Box](#)< T > &lhs, const [Box](#)< U > &rhs)
- template<typename T, typename U>
bool [operator!=](#) (const [Box](#)< T > &lhs, const [Box](#)< U > &rhs)

5.2.1 Function Documentation

5.2.1.1 `Box()` [1/2]

```
template<typename T >
wr22::utils::Box (
    std::unique_ptr< T > ptr ) -> Box< T >
```

Type deduction guideline for `Box` (`std::unique_ptr` adoption).

5.2.1.2 `Box()` [2/2]

```
template<typename T >
wr22::utils::Box (
    T && value ) -> Box< T >
```

Type deduction guideline for `Box` (value initialization).

5.2.1.3 `operator!=()` [1/2]

```
template<typename... Variants>
bool wr22::utils::operator!= (
    const Adt< Variants... > & lhs,
    const Adt< Variants... > & rhs )
```

Compare two compatible ADTs for non-equality.

5.2.1.4 `operator!=()` [2/2]

```
template<typename T , typename U >
bool wr22::utils::operator!= (
    const Box< T > & lhs,
    const Box< U > & rhs )
```

5.2.1.5 `operator==()` [1/2]

```
template<typename... Variants>
bool wr22::utils::operator== (
    const Adt< Variants... > & lhs,
    const Adt< Variants... > & rhs )
```

Compare two compatible ADTs for equality.

5.2.1.6 operator==([2/2]

```
template<typename T , typename U >
bool wr22::utils::operator==(
    const Box< T > & lhs,
    const Box< U > & rhs )
```

5.3 wr22::utils::detail Namespace Reference

Namespaces

- namespace [adt](#)

5.4 wr22::utils::detail::adt Namespace Reference

Classes

- struct [MultiCallable](#)

Chapter 6

Class Documentation

6.1 wr22::utils::Adt< Variants > Class Template Reference

A helper class that simplifies creation of algebraic data types.

```
#include <adt.hpp>
```

Public Types

- using [VariantType](#) = std::variant< Variants... >
A convenience type alias for the concrete `std::variant` type used.

Public Member Functions

- template<typename V >
[Adt](#) (V variant)
Constructor for each of the variants.
- template<typename... Fs>
decltype(auto) [visit](#) (Fs &&... visitors) const
Visit the ADT, applying the suitable function from the list of visitors on the variant held.
- template<typename... Fs>
decltype(auto) [visit](#) (Fs &&... visitors)
Visit the ADT, applying the suitable function from the list of visitors on the variant held.
- const [VariantType](#) & [as_variant](#) () const
Access the underlying `std::variant` type (constant version).
- [VariantType](#) & [as_variant](#) ()
Access the underlying `std::variant` type (non-constant version).

Protected Attributes

- [VariantType](#) m_variant

6.1.1 Detailed Description

```
template<typename... Variants>
class wr22::utils::Adt< Variants >
```

A helper class that simplifies creation of algebraic data types.

Algebraic data types are data types that can have one type of a predefined set of variants, but be stored and represented as values of one common type. In C++, `std::variant` serves exactly this purpose. It is, however, not very convenient to work with or build upon, so this class is designed to simplify building new algebraic data types. It still uses `std::variant` under the hood.

The template type parameters are the types that the variants may hold (must be distinct types).

6.1.2 Member Typedef Documentation

6.1.2.1 VariantType

```
template<typename... Variants>
using wr22::utils::Adt< Variants >::VariantType = std::variant<Variants...>
```

A convenience type alias for the concrete `std::variant` type used.

6.1.3 Constructor & Destructor Documentation

6.1.3.1 Adt()

```
template<typename... Variants>
template<typename V >
wr22::utils::Adt< Variants >::Adt (
    V variant ) [inline]
```

Constructor for each of the variants.

Construct an instance holding a specified variant. The type `V` of the variant provided must be one of the types from `Variants`. Note that this constructor is purposefully implicit, so that the variants as separate types are transparently converted to this common type when necessary.

The variant is taken by value and moved thereafter, so that, when constructing the common type, the variant may be either copied or moved, depending on the user's intentions.

6.1.4 Member Function Documentation

6.1.4.1 as_variant() [1/2]

```
template<typename... Variants>
VariantType & wr22::utils::Adt< Variants >::as_variant ( ) [inline]
```

Access the underlying `std::variant` type (non-constant version).

6.1.4.2 as_variant() [2/2]

```
template<typename... Variants>
const VariantType & wr22::utils::Adt< Variants >::as_variant ( ) const [inline]
```

Access the underlying `std::variant` type (constant version).

6.1.4.3 visit() [1/2]

```
template<typename... Variants>
template<typename... Fs>
decltype(auto) wr22::utils::Adt< Variants >::visit (
    Fs &&... visitors ) [inline]
```

Visit the ADT, applying the suitable function from the list of visitors on the variant held.

This is the non-constant version of the method. See the docs for the constant version for a detailed description and code examples. The only thing different in this version of the method is that the visitors get called with a non-const lvalue reference to the variants instead of a const reference.

6.1.4.4 visit() [2/2]

```
template<typename... Variants>
template<typename... Fs>
decltype(auto) wr22::utils::Adt< Variants >::visit (
    Fs &&... visitors ) const [inline]
```

Visit the ADT, applying the suitable function from the list of visitors on the variant held.

Using this method is essentially the same as using `std::visit` on the variant, except that, for convenience, multiple visitors are joined into one big visitor. That is, a typical `Adt` usage might look like this:

```
struct MyAdt : public Adt<int, double> {
    // Make the constructor available in the derived class.
    using Adt<int, double>::Adt;
};
// <...>
void func() {
    // Variant type: double.
    MyAdt my_adt = 3.14;
    // Prints "Double: 3.14".
    my_adt.visit(
        [] (int x) { std::cout << "Int: " << x << std::endl; },
        [] (double x) { std::cout << "Double: " << x << std::endl; }
    );
}
```

This is the constant version of the method. Visitors must be callable with the const reference to variant types.

6.1.5 Member Data Documentation

6.1.5.1 m_variant

```
template<typename... Variants>
VariantType wr22::utils::Adt< Variants >::m_variant [protected]
```

The documentation for this class was generated from the following file:

- include/wr22/utlis/[adt.hpp](#)

6.2 wr22::utils::Box< T > Class Template Reference

A copyable and equality-comparable wrapper around `std::unique_ptr`.

```
#include <box.hpp>
```

Public Member Functions

- [Box](#) (T &&value)
Constructor that places a value inside the wrapped `std::unique_ptr`.
- [Box](#) (std::unique_ptr< T > ptr)
Constructor that adopts an existing `std::unique_ptr`.
- template<typename Dummy = T>
[Box](#) (const [Box](#) &other)
Copy constructor.
- const T & [operator*](#) () const
Dereferencing operator: obtain a const reference to the stored value.
- T & [operator*](#) ()
Dereferencing operator: obtain a reference to the stored value.

Static Public Member Functions

- template<typename... Args>
static [Box](#)< T > [construct_in_place](#) (Args &&... args)
Construct a value on the heap in place.

6.2.1 Detailed Description

```
template<typename T>
class wr22::utils::Box< T >
```

A copyable and equality-comparable wrapper around `std::unique_ptr`.

The behavior of this wrapper regarding copying and equality comparison are akin to that of Rust's `std::boxed::Box`, and hence the class's name. Namely, when testing for (in)equality, the wrapped values are compared instead of raw pointers, and, when wrapped values are copyable, copying a [Box](#) creates another `std::unique_ptr` with a copy of the wrapped value.

A [Box](#) usually contains a value. However, it may become empty when it is moved from. To ensure safety, most operations on an empty box will throw a [BoxIsEmpty](#) exception instead of causing undefined behavior.

6.2.2 Constructor & Destructor Documentation

6.2.2.1 Box() [1/3]

```
template<typename T >
wr22::utils::Box< T >::Box (
    T && value ) [inline], [explicit]
```

Constructor that places a value inside the wrapped `std::unique_ptr`.

Takes the value by a universal reference and, due to perfect forwarding, both copy and move initialization is possible.

6.2.2.2 Box() [2/3]

```
template<typename T >
wr22::utils::Box< T >::Box (
    std::unique_ptr< T > ptr ) [inline], [explicit]
```

Constructor that adopts an existing `std::unique_ptr`.

Takes the `std::unique_ptr` by value, so the latter must be either passed directly as an rvalue or `std::move()` d into the argument. However, please note that if your code snippet looks like this:

```
Box(std::make_unique<T>(args...))
```

Then you should take a look at the `construct_in_place` method:

```
Box<T>::construct_in_place(args...)
```

6.2.2.3 Box() [3/3]

```
template<typename T >
template<typename Dummy = T>
wr22::utils::Box< T >::Box (
    const Box< T > & other ) [inline]
```

Copy constructor.

Creates another `std::unique_ptr` with a copy of the currently wrapped value.

Parameters

<code>`other`</code>	the <code>Box</code> from which to copy.
----------------------	--

Exceptions

<code>BoxIsEmpty</code>	if <code>other</code> is empty.
-------------------------	---------------------------------

6.2.3 Member Function Documentation

6.2.3.1 `construct_in_place()`

```
template<typename T >
template<typename... Args>
static Box< T > wr22::utils::Box< T >::construct_in_place (
    Args &&... args ) [inline], [static]
```

Construct a value on the heap in place.

Forwards the arguments to `std::make_unique` and wraps the resulting `std::unique_ptr`.

6.2.3.2 `operator*()` [1/2]

```
template<typename T >
T & wr22::utils::Box< T >::operator* ( ) [inline]
```

Dereferencing operator: obtain a reference to the stored value.

Exceptions

<i>BoxIsEmpty</i>	if this <code>Box</code> does not contain a value at the moment.
-----------------------------------	--

6.2.3.3 `operator*()` [2/2]

```
template<typename T >
const T & wr22::utils::Box< T >::operator* ( ) const [inline]
```

Dereferencing operator: obtain a const reference to the stored value.

Exceptions

<i>BoxIsEmpty</i>	if this <code>Box</code> does not contain a value at the moment.
-----------------------------------	--

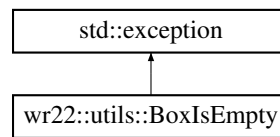
The documentation for this class was generated from the following file:

- `include/wr22/utils/box.hpp`

6.3 `wr22::utils::BoxIsEmpty` Struct Reference

```
#include <box.hpp>
```

Inheritance diagram for wr22::utils::BoxIsEmpty:



Public Member Functions

- `const char * what () const` noexcept override

6.3.1 Member Function Documentation

6.3.1.1 what()

```
const char * wr22::utils::BoxIsEmpty::what ( ) const [override], [noexcept]
```

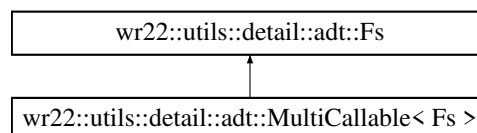
The documentation for this struct was generated from the following files:

- `include/wr22/utils/box.hpp`
- `src/box.cpp`

6.4 wr22::utils::detail::adt::MultiCallable< Fs > Struct Template Reference

```
#include <adt.hpp>
```

Inheritance diagram for wr22::utils::detail::adt::MultiCallable< Fs >:



Public Member Functions

- `MultiCallable (Fs &&... fs)`

6.4.1 Constructor & Destructor Documentation

6.4.1.1 MultiCallable()

```
template<typename... Fs>
wr22::utils::detail::adt::MultiCallable< Fs >::MultiCallable (
    Fs &&... fs ) [inline]
```

The documentation for this struct was generated from the following file:

- include/wr22/utlis/[adt.hpp](#)

6.5 wr22::utils::NonconcurrentSemaphore Class Reference

```
#include <nonconcurrent_semaphore.hpp>
```

Public Member Functions

- [NonconcurrentSemaphore](#) (size_t max_capacity)
- std::optional< [NonconcurrentSemaphoreGuard](#) > [enter](#) ()

Friends

- class [NonconcurrentSemaphoreGuard](#)

6.5.1 Constructor & Destructor Documentation

6.5.1.1 NonconcurrentSemaphore()

```
wr22::utils::NonconcurrentSemaphore::NonconcurrentSemaphore (
    size_t max_capacity ) [explicit]
```

6.5.2 Member Function Documentation

6.5.2.1 enter()

```
std::optional< NonconcurrentSemaphoreGuard > wr22::utils::NonconcurrentSemaphore::enter ( )
```

6.5.3 Friends And Related Function Documentation

6.5.3.1 NonconcurrentSemaphoreGuard

```
friend class NonconcurrentSemaphoreGuard [friend]
```

The documentation for this class was generated from the following files:

- include/wr22/utils/nonconcurrent_semaphore.hpp
- src/nonconcurrent_semaphore.cpp

6.6 wr22::utils::NonconcurrentSemaphoreGuard Class Reference

```
#include <nonconcurrent_semaphore.hpp>
```

Public Member Functions

- [NonconcurrentSemaphoreGuard](#) (const [NonconcurrentSemaphoreGuard](#) &)=delete
- [NonconcurrentSemaphoreGuard](#) ([NonconcurrentSemaphoreGuard](#) &&other)
- [NonconcurrentSemaphoreGuard](#) & operator= (const [NonconcurrentSemaphoreGuard](#) &)=delete
- [NonconcurrentSemaphoreGuard](#) & operator= ([NonconcurrentSemaphoreGuard](#) &&other)
- [~NonconcurrentSemaphoreGuard](#) ()

Friends

- class [NonconcurrentSemaphore](#)

6.6.1 Constructor & Destructor Documentation

6.6.1.1 NonconcurrentSemaphoreGuard() [1/2]

```
wr22::utils::NonconcurrentSemaphoreGuard::NonconcurrentSemaphoreGuard (  
    const NonconcurrentSemaphoreGuard & ) [delete]
```

6.6.1.2 NonconcurrentSemaphoreGuard() [2/2]

```
wr22::utils::NonconcurrentSemaphoreGuard::NonconcurrentSemaphoreGuard (  
    NonconcurrentSemaphoreGuard && other )
```

6.6.1.3 ~NonconcurrentSemaphoreGuard()

```
wr22::utils::NonconcurrentSemaphoreGuard::~~NonconcurrentSemaphoreGuard ( )
```

6.6.2 Member Function Documentation

6.6.2.1 operator=() [1/2]

```
NonconcurrentSemaphoreGuard & wr22::utils::NonconcurrentSemaphoreGuard::operator= (
    const NonconcurrentSemaphoreGuard & ) [delete]
```

6.6.2.2 operator=() [2/2]

```
NonconcurrentSemaphoreGuard & wr22::utils::NonconcurrentSemaphoreGuard::operator= (
    NonconcurrentSemaphoreGuard && other )
```

6.6.3 Friends And Related Function Documentation

6.6.3.1 NonconcurrentSemaphore

```
friend class NonconcurrentSemaphore [friend]
```

The documentation for this class was generated from the following files:

- include/wr22/Utils/nonconcurrent_semaphore.hpp
- src/nonconcurrent_semaphore.cpp

Chapter 7

File Documentation

7.1 include/wr22/utils/adt.hpp File Reference

```
#include <utility>
#include <variant>
```

Classes

- struct [wr22::utils::detail::adt::MultiCallable< Fs >](#)
- class [wr22::utils::Adt< Variants >](#)

A helper class that simplifies creation of algebraic data types.

Namespaces

- namespace [wr22](#)
- namespace [wr22::utils](#)
- namespace [wr22::utils::detail](#)
- namespace [wr22::utils::detail::adt](#)

Functions

- template<typename... Variants>
bool [wr22::utils::operator==](#) (const Adt< Variants... > &lhs, const Adt< Variants... > &rhs)
Compare two compatible ADTs for equality.
- template<typename... Variants>
bool [wr22::utils::operator!=](#) (const Adt< Variants... > &lhs, const Adt< Variants... > &rhs)
Compare two compatible ADTs for non-equality.

7.2 adt.hpp

[Go to the documentation of this file.](#)

```

1 #pragma once
2
3 // stl
4 #include <utility>
5 #include <variant>
6
7 namespace wr22::utils {
8
9 namespace detail::adt {
10     // https://en.cppreference.com/w/cpp/utility/variant/visit#Example provides a very similar
11     // example of C++ template black magic.
12     template <typename... Fs>
13     struct MultiCallable : public Fs... {
14         MultiCallable(Fs&&... fs) : Fs(fs)... {}
15         using Fs::operator()...;
16     };
17 } // namespace detail::adt
18
19 template <typename... Variants>
20 class Adt {
21 public:
22     using VariantType = std::variant<Variants...>;
23
24     template <typename V>
25     Adt(V variant) : m_variant(std::move(variant)) {}
26
27     template <typename... Fs>
28     decltype(auto) visit(Fs&&... visitors) const {
29         return std::visit(
30             detail::adt::MultiCallable<Fs...>(std::forward<Fs>(visitors)...),
31             m_variant);
32     }
33
34     template <typename... Fs>
35     decltype(auto) visit(Fs&&... visitors) {
36         return std::visit(
37             detail::adt::MultiCallable<Fs...>(std::forward<Fs>(visitors)...),
38             m_variant);
39     }
40
41     const VariantType& as_variant() const {
42         return m_variant;
43     }
44
45     VariantType& as_variant() {
46         return m_variant;
47     }
48
49 protected:
50     VariantType m_variant;
51 };
52
53 template <typename... Variants>
54 bool operator==(const Adt<Variants...>& lhs, const Adt<Variants...>& rhs) {
55     return lhs.as_variant() == rhs.as_variant();
56 }
57
58 template <typename... Variants>
59 bool operator!=(const Adt<Variants...>& lhs, const Adt<Variants...>& rhs) {
60     return !(lhs == rhs);
61 }
62
63 } // namespace wr22::utils

```

7.3 include/wr22/utils/box.hpp File Reference

```

#include <exception>
#include <memory>
#include <utility>

```

Classes

- struct [wr22::utils::BoxIsEmpty](#)
- class [wr22::utils::Box< T >](#)

A copyable and equality-comparable wrapper around `std::unique_ptr`.

Namespaces

- namespace [wr22](#)
- namespace [wr22::utils](#)

Functions

- template<typename T >
[wr22::utils::Box](#) (T &&value) -> Box< T >
Type deduction guideline for [Box](#) (value initialization).
- template<typename T >
[wr22::utils::Box](#) (std::unique_ptr< T > ptr) -> Box< T >
Type deduction guideline for [Box](#) (std::unique_ptr adoption).
- template<typename T, typename U >
bool [wr22::utils::operator==](#) (const Box< T > &lhs, const Box< U > &rhs)
- template<typename T, typename U >
bool [wr22::utils::operator!=](#) (const Box< T > &lhs, const Box< U > &rhs)

7.4 box.hpp

[Go to the documentation of this file.](#)

```

1 #pragma once
2
3 // stl
4 #include <exception>
5 #include <memory>
6 #include <utility>
7
8 namespace wr22::utils {
9
10 struct BoxIsEmpty : public std::exception {
11     const char* what() const noexcept override;
12 };
13
14 template <typename T>
15 class Box {
16 public:
17     explicit Box(T&& value) : m_ptr(std::make_unique<T>(std::forward<T>(value))) {}
18
19     explicit Box(std::unique_ptr<T> ptr) : m_ptr(std::move(ptr)) {}
20
21     template <typename Dummy = T>
22     Box(const Box& other) : m_ptr(std::make_unique<T>(*other)) {}
23
24     template <typename... Args>
25     static Box<T> construct_in_place(Args&&... args) {
26         return Box(std::make_unique<T>(std::forward<Args>(args)...));
27     }
28
29     const T& operator*() const {
30         if (m_ptr == nullptr) {
31             throw BoxIsEmpty{};
32         }
33         return *m_ptr;
34     }
35
36     T& operator*() {
37         if (m_ptr == nullptr) {
38             throw BoxIsEmpty{};
39         }
40         return *m_ptr;
41     }
42
43     const T& operator*() const {
44         if (m_ptr == nullptr) {
45             throw BoxIsEmpty{};
46         }
47         return *m_ptr;
48     }
49
50     T& operator*() {
51         if (m_ptr == nullptr) {
52             throw BoxIsEmpty{};
53         }
54         return *m_ptr;
55     }
56
57     const T& operator*() const {
58         if (m_ptr == nullptr) {
59             throw BoxIsEmpty{};
60         }
61         return *m_ptr;
62     }
63
64     T& operator*() {
65         if (m_ptr == nullptr) {
66             throw BoxIsEmpty{};
67         }
68         return *m_ptr;
69     }
70
71     const T& operator*() const {
72         if (m_ptr == nullptr) {
73             throw BoxIsEmpty{};
74         }
75         return *m_ptr;
76     }
77
78     T& operator*() {
79         if (m_ptr == nullptr) {
80             throw BoxIsEmpty{};
81         }
82         return *m_ptr;
83     }
84
85     const T& operator*() const {
86         if (m_ptr == nullptr) {
87             throw BoxIsEmpty{};
88         }
89         return *m_ptr;
90     }
91
92     T& operator*() {
93         if (m_ptr == nullptr) {
94             throw BoxIsEmpty{};
95         }
96         return *m_ptr;
97     }
98
99     const T& operator*() const {
100         if (m_ptr == nullptr) {
101             throw BoxIsEmpty{};
102         }
103         return *m_ptr;
104     }
105
106     T& operator*() {
107         if (m_ptr == nullptr) {
108             throw BoxIsEmpty{};
109         }
110         return *m_ptr;
111     }
112
113     const T& operator*() const {
114         if (m_ptr == nullptr) {
115             throw BoxIsEmpty{};
116         }
117         return *m_ptr;
118     }
119
120     T& operator*() {
121         if (m_ptr == nullptr) {
122             throw BoxIsEmpty{};
123         }
124         return *m_ptr;
125     }
126
127     const T& operator*() const {
128         if (m_ptr == nullptr) {
129             throw BoxIsEmpty{};
130         }
131         return *m_ptr;
132     }
133
134     T& operator*() {
135         if (m_ptr == nullptr) {
136             throw BoxIsEmpty{};
137         }
138         return *m_ptr;
139     }
140
141     const T& operator*() const {
142         if (m_ptr == nullptr) {
143             throw BoxIsEmpty{};
144         }
145         return *m_ptr;
146     }
147
148     T& operator*() {
149         if (m_ptr == nullptr) {
150             throw BoxIsEmpty{};
151         }
152         return *m_ptr;
153     }
154
155     const T& operator*() const {
156         if (m_ptr == nullptr) {
157             throw BoxIsEmpty{};
158         }
159         return *m_ptr;
160     }
161
162     T& operator*() {
163         if (m_ptr == nullptr) {
164             throw BoxIsEmpty{};
165         }
166         return *m_ptr;
167     }
168
169     const T& operator*() const {
170         if (m_ptr == nullptr) {
171             throw BoxIsEmpty{};
172         }
173         return *m_ptr;
174     }
175
176     T& operator*() {
177         if (m_ptr == nullptr) {
178             throw BoxIsEmpty{};
179         }
180         return *m_ptr;
181     }
182
183     const T& operator*() const {
184         if (m_ptr == nullptr) {
185             throw BoxIsEmpty{};
186         }
187         return *m_ptr;
188     }
189
190     T& operator*() {
191         if (m_ptr == nullptr) {
192             throw BoxIsEmpty{};
193         }
194         return *m_ptr;
195     }
196
197     const T& operator*() const {
198         if (m_ptr == nullptr) {
199             throw BoxIsEmpty{};
200         }
201         return *m_ptr;
202     }
203
204     T& operator*() {
205         if (m_ptr == nullptr) {
206             throw BoxIsEmpty{};
207         }
208         return *m_ptr;
209     }
210
211     const T& operator*() const {
212         if (m_ptr == nullptr) {
213             throw BoxIsEmpty{};
214         }
215         return *m_ptr;
216     }
217
218     T& operator*() {
219         if (m_ptr == nullptr) {
220             throw BoxIsEmpty{};
221         }
222         return *m_ptr;
223     }
224
225     const T& operator*() const {
226         if (m_ptr == nullptr) {
227             throw BoxIsEmpty{};
228         }
229         return *m_ptr;
230     }
231
232     T& operator*() {
233         if (m_ptr == nullptr) {
234             throw BoxIsEmpty{};
235         }
236         return *m_ptr;
237     }
238
239     const T& operator*() const {
240         if (m_ptr == nullptr) {
241             throw BoxIsEmpty{};
242         }
243         return *m_ptr;
244     }
245
246     T& operator*() {
247         if (m_ptr == nullptr) {
248             throw BoxIsEmpty{};
249         }
250         return *m_ptr;
251     }
252
253     const T& operator*() const {
254         if (m_ptr == nullptr) {
255             throw BoxIsEmpty{};
256         }
257         return *m_ptr;
258     }
259
260     T& operator*() {
261         if (m_ptr == nullptr) {
262             throw BoxIsEmpty{};
263         }
264         return *m_ptr;
265     }
266
267     const T& operator*() const {
268         if (m_ptr == nullptr) {
269             throw BoxIsEmpty{};
270         }
271         return *m_ptr;
272     }
273
274     T& operator*() {
275         if (m_ptr == nullptr) {
276             throw BoxIsEmpty{};
277         }
278         return *m_ptr;
279     }
280
281     const T& operator*() const {
282         if (m_ptr == nullptr) {
283             throw BoxIsEmpty{};
284         }
285         return *m_ptr;
286     }
287
288     T& operator*() {
289         if (m_ptr == nullptr) {
290             throw BoxIsEmpty{};
291         }
292         return *m_ptr;
293     }
294
295     const T& operator*() const {
296         if (m_ptr == nullptr) {
297             throw BoxIsEmpty{};
298         }
299         return *m_ptr;
300     }
301
302     T& operator*() {
303         if (m_ptr == nullptr) {
304             throw BoxIsEmpty{};
305         }
306         return *m_ptr;
307     }
308
309     const T& operator*() const {
310         if (m_ptr == nullptr) {
311             throw BoxIsEmpty{};
312         }
313         return *m_ptr;
314     }
315
316     T& operator*() {
317         if (m_ptr == nullptr) {
318             throw BoxIsEmpty{};
319         }
320         return *m_ptr;
321     }
322
323     const T& operator*() const {
324         if (m_ptr == nullptr) {
325             throw BoxIsEmpty{};
326         }
327         return *m_ptr;
328     }
329
330     T& operator*() {
331         if (m_ptr == nullptr) {
332             throw BoxIsEmpty{};
333         }
334         return *m_ptr;
335     }
336
337     const T& operator*() const {
338         if (m_ptr == nullptr) {
339             throw BoxIsEmpty{};
340         }
341         return *m_ptr;
342     }
343
344     T& operator*() {
345         if (m_ptr == nullptr) {
346             throw BoxIsEmpty{};
347         }
348         return *m_ptr;
349     }
350
351     const T& operator*() const {
352         if (m_ptr == nullptr) {
353             throw BoxIsEmpty{};
354         }
355         return *m_ptr;
356     }
357
358     T& operator*() {
359         if (m_ptr == nullptr) {
360             throw BoxIsEmpty{};
361         }
362         return *m_ptr;
363     }
364
365     const T& operator*() const {
366         if (m_ptr == nullptr) {
367             throw BoxIsEmpty{};
368         }
369         return *m_ptr;
370     }
371
372     T& operator*() {
373         if (m_ptr == nullptr) {
374             throw BoxIsEmpty{};
375         }
376         return *m_ptr;
377     }
378
379     const T& operator*() const {
380         if (m_ptr == nullptr) {
381             throw BoxIsEmpty{};
382         }
383         return *m_ptr;
384     }
385
386     T& operator*() {
387         if (m_ptr == nullptr) {
388             throw BoxIsEmpty{};
389         }
390         return *m_ptr;
391     }
392
393     const T& operator*() const {
394         if (m_ptr == nullptr) {
395             throw BoxIsEmpty{};
396         }
397         return *m_ptr;
398     }
399
400     T& operator*() {
401         if (m_ptr == nullptr) {
402             throw BoxIsEmpty{};
403         }
404         return *m_ptr;
405     }
406
407     const T& operator*() const {
408         if (m_ptr == nullptr) {
409             throw BoxIsEmpty{};
410         }
411         return *m_ptr;
412     }
413
414     T& operator*() {
415         if (m_ptr == nullptr) {
416             throw BoxIsEmpty{};
417         }
418         return *m_ptr;
419     }
420
421     const T& operator*() const {
422         if (m_ptr == nullptr) {
423             throw BoxIsEmpty{};
424         }
425         return *m_ptr;
426     }
427
428     T& operator*() {
429         if (m_ptr == nullptr) {
430             throw BoxIsEmpty{};
431         }
432         return *m_ptr;
433     }
434
435     const T& operator*() const {
436         if (m_ptr == nullptr) {
437             throw BoxIsEmpty{};
438         }
439         return *m_ptr;
440     }
441
442     T& operator*() {
443         if (m_ptr == nullptr) {
444             throw BoxIsEmpty{};
445         }
446         return *m_ptr;
447     }
448
449     const T& operator*() const {
450         if (m_ptr == nullptr) {
451             throw BoxIsEmpty{};
452         }
453         return *m_ptr;
454     }
455
456     T& operator*() {
457         if (m_ptr == nullptr) {
458             throw BoxIsEmpty{};
459         }
460         return *m_ptr;
461     }
462
463     const T& operator*() const {
464         if (m_ptr == nullptr) {
465             throw BoxIsEmpty{};
466         }
467         return *m_ptr;
468     }
469
470     T& operator*() {
471         if (m_ptr == nullptr) {
472             throw BoxIsEmpty{};
473         }
474         return *m_ptr;
475     }
476
477     const T& operator*() const {
478         if (m_ptr == nullptr) {
479             throw BoxIsEmpty{};
480         }
481         return *m_ptr;
482     }
483
484     T& operator*() {
485         if (m_ptr == nullptr) {
486             throw BoxIsEmpty{};
487         }
488         return *m_ptr;
489     }
490
491     const T& operator*() const {
492         if (m_ptr == nullptr) {
493             throw BoxIsEmpty{};
494         }
495         return *m_ptr;
496     }
497
498     T& operator*() {
499         if (m_ptr == nullptr) {
500             throw BoxIsEmpty{};
501         }
502         return *m_ptr;
503     }
504
505     const T& operator*() const {
506         if (m_ptr == nullptr) {
507             throw BoxIsEmpty{};
508         }
509         return *m_ptr;
510     }
511
512     T& operator*() {
513         if (m_ptr == nullptr) {
514             throw BoxIsEmpty{};
515         }
516         return *m_ptr;
517     }
518
519     const T& operator*() const {
520         if (m_ptr == nullptr) {
521             throw BoxIsEmpty{};
522         }
523         return *m_ptr;
524     }
525
526     T& operator*() {
527         if (m_ptr == nullptr) {
528             throw BoxIsEmpty{};
529         }
530         return *m_ptr;
531     }
532
533     const T& operator*() const {
534         if (m_ptr == nullptr) {
535             throw BoxIsEmpty{};
536         }
537         return *m_ptr;
538     }
539
540     T& operator*() {
541         if (m_ptr == nullptr) {
542             throw BoxIsEmpty{};
543         }
544         return *m_ptr;
545     }
546
547     const T& operator*() const {
548         if (m_ptr == nullptr) {
549             throw BoxIsEmpty{};
550         }
551         return *m_ptr;
552     }
553
554     T& operator*() {
555         if (m_ptr == nullptr) {
556             throw BoxIsEmpty{};
557         }
558         return *m_ptr;
559     }
560
561     const T& operator*() const {
562         if (m_ptr == nullptr) {
563             throw BoxIsEmpty{};
564         }
565         return *m_ptr;
566     }
567
568     T& operator*() {
569         if (m_ptr == nullptr) {
570             throw BoxIsEmpty{};
571         }
572         return *m_ptr;
573     }
574
575     const T& operator*() const {
576         if (m_ptr == nullptr) {
577             throw BoxIsEmpty{};
578         }
579         return *m_ptr;
580     }
581
582     T& operator*() {
583         if (m_ptr == nullptr) {
584             throw BoxIsEmpty{};
585         }
586         return *m_ptr;
587     }
588
589     const T& operator*() const {
590         if (m_ptr == nullptr) {
591             throw BoxIsEmpty{};
592         }
593         return *m_ptr;
594     }
595
596     T& operator*() {
597         if (m_ptr == nullptr) {
598             throw BoxIsEmpty{};
599         }
600         return *m_ptr;
601     }
602
603     const T& operator*() const {
604         if (m_ptr == nullptr) {
605             throw BoxIsEmpty{};
606         }
607         return *m_ptr;
608     }
609
610     T& operator*() {
611         if (m_ptr == nullptr) {
612             throw BoxIsEmpty{};
613         }
614         return *m_ptr;
615     }
616
617     const T& operator*() const {
618         if (m_ptr == nullptr) {
619             throw BoxIsEmpty{};
620         }
621         return *m_ptr;
622     }
623
624     T& operator*() {
625         if (m_ptr == nullptr) {
626             throw BoxIsEmpty{};
627         }
628         return *m_ptr;
629     }
630
631     const T& operator*() const {
632         if (m_ptr == nullptr) {
633             throw BoxIsEmpty{};
634         }
635         return *m_ptr;
636     }
637
638     T& operator*() {
639         if (m_ptr == nullptr) {
640             throw BoxIsEmpty{};
641         }
642         return *m_ptr;
643     }
644
645     const T& operator*() const {
646         if (m_ptr == nullptr) {
647             throw BoxIsEmpty{};
648         }
649         return *m_ptr;
650     }
651
652     T& operator*() {
653         if (m_ptr == nullptr) {
654             throw BoxIsEmpty{};
655         }
656         return *m_ptr;
657     }
658
659     const T& operator*() const {
660         if (m_ptr == nullptr) {
661             throw BoxIsEmpty{};
662         }
663         return *m_ptr;
664     }
665
666     T& operator*() {
667         if (m_ptr == nullptr) {
668             throw BoxIsEmpty{};
669         }
670         return *m_ptr;
671     }
672
673     const T& operator*() const {
674         if (m_ptr == nullptr) {
675             throw BoxIsEmpty{};
676         }
677         return *m_ptr;
678     }
679
680     T& operator*() {
681         if (m_ptr == nullptr) {
682             throw BoxIsEmpty{};
683         }
684         return *m_ptr;
685     }
686
687     const T& operator*() const {
688         if (m_ptr == nullptr) {
689             throw BoxIsEmpty{};
690         }
691         return *m_ptr;
692     }
693
694     T& operator*() {
695         if (m_ptr == nullptr) {
696             throw BoxIsEmpty{};
697         }
698         return *m_ptr;
699     }
700
701     const T& operator*() const {
702         if (m_ptr == nullptr) {
703             throw BoxIsEmpty{};
704         }
705         return *m_ptr;
706     }
707
708     T& operator*() {
709         if (m_ptr == nullptr) {
710             throw BoxIsEmpty{};
711         }
712         return *m_ptr;
713     }
714
715     const T& operator*() const {
716         if (m_ptr == nullptr) {
717             throw BoxIsEmpty{};
718         }
719         return *m_ptr;
720     }
721
722     T& operator*() {
723         if (m_ptr == nullptr) {
724             throw BoxIsEmpty{};
725         }
726         return *m_ptr;
727     }
728
729     const T& operator*() const {
730         if (m_ptr == nullptr) {
731             throw BoxIsEmpty{};
732         }
733         return *m_ptr;
734     }
735
736     T& operator*() {
737         if (m_ptr == nullptr) {
738             throw BoxIsEmpty{};
739         }
740         return *m_ptr;
741     }
742
743     const T& operator*() const {
744         if (m_ptr == nullptr) {
745             throw BoxIsEmpty{};
746         }
747         return *m_ptr;
748     }
749
750     T& operator*() {
751         if (m_ptr == nullptr) {
752             throw BoxIsEmpty{};
753         }
754         return *m_ptr;
755     }
756
757     const T& operator*() const {
758         if (m_ptr == nullptr) {
759             throw BoxIsEmpty{};
760         }
761         return *m_ptr;
762     }
763
764     T& operator*() {
765         if (m_ptr == nullptr) {
766             throw BoxIsEmpty{};
767         }
768         return *m_ptr;
769     }
770
771     const T& operator*() const {
772         if (m_ptr == nullptr) {
773             throw BoxIsEmpty{};
774         }
775         return *m_ptr;
776     }
777
778     T& operator*() {
779         if (m_ptr == nullptr) {
780             throw BoxIsEmpty{};
781         }
782         return *m_ptr;
783     }
784
785     const T& operator*() const {
786         if (m_ptr == nullptr) {
787             throw BoxIsEmpty{};
788         }
789         return *m_ptr;
790     }
791
792     T& operator*() {
793         if (m_ptr == nullptr) {
794             throw BoxIsEmpty{};
795         }
796         return *m_ptr;
797     }
798
799     const T& operator*() const {
800         if (m_ptr == nullptr) {
801             throw BoxIsEmpty{};
802         }
803         return *m_ptr;
804     }
805
806     T& operator*() {
807         if (m_ptr == nullptr) {
808             throw BoxIsEmpty{};
809         }
810         return *m_ptr;
811     }
812
813     const T& operator*() const {
814         if (m_ptr == nullptr) {
815             throw BoxIsEmpty{};
816         }
817         return *m_ptr;
818     }
819
820     T& operator*() {
821         if (m_ptr == nullptr) {
822             throw BoxIsEmpty{};
823         }
824         return *m_ptr;
825     }
826
827     const T& operator*() const {
828         if (m_ptr == nullptr) {
829             throw BoxIsEmpty{};
830         }
831         return *m_ptr;
832     }
833
834     T& operator*() {
835         if (m_ptr == nullptr) {
836             throw BoxIsEmpty{};
837         }
838         return *m_ptr;
839     }
840
841     const T& operator*() const {
842         if (m_ptr == nullptr) {
843             throw BoxIsEmpty{};
844         }
845         return *m_ptr;
846     }
847
848     T& operator*() {
849         if (m_ptr == nullptr) {
850             throw BoxIsEmpty{};
851         }
852         return *m_ptr;
853     }
854
855     const T& operator*() const {
856         if (m_ptr == nullptr) {
857             throw BoxIsEmpty{};
858         }
859         return *m_ptr;
860     }
861
862     T& operator*() {
863         if (m_ptr == nullptr) {
864             throw BoxIsEmpty{};
865         }
866         return *m_ptr;
867     }
868
869     const T& operator*() const {
870         if (m_ptr == nullptr) {
871             throw BoxIsEmpty{};
872         }
873         return *m_ptr;
874     }
875
876     T& operator*() {
877         if (m_ptr == nullptr) {
878             throw BoxIsEmpty{};
879         }
880         return *m_ptr;
881     }
882
883     const T& operator*() const {
884         if (m_ptr == nullptr) {
885             throw BoxIsEmpty{};
886         }
887         return *m_ptr;
888     }
889
890     T& operator*() {
891         if (m_ptr == nullptr) {
892             throw BoxIsEmpty{};
893         }
894         return *m_ptr;
895     }
896
897     const T& operator*() const {
898         if (m_ptr == nullptr) {
899             throw BoxIsEmpty{};
900         }
901         return *m_ptr;
902     }
903
904     T& operator*() {
905         if (m_ptr == nullptr) {
906             throw BoxIsEmpty{};
907         }
908         return *m_ptr;
909     }
910
911     const T& operator*() const {
912         if (m_ptr == nullptr) {
913             throw BoxIsEmpty{};
914         }
915         return *m_ptr;
916     }
917
918     T& operator*() {
919         if (m_ptr == nullptr) {
920             throw BoxIsEmpty{};
921         }
922         return *m_ptr;
923     }
924
925     const T& operator*() const {
926         if (m_ptr == nullptr) {
927             throw BoxIsEmpty{};
928         }
929         return *m_ptr;
930     }
931
932     T& operator*() {
933         if (m_ptr == nullptr) {
934             throw BoxIsEmpty{};
935         }
936         return *m_ptr;
937     }
938
939     const T& operator*() const {
940         if (m_ptr == nullptr) {
941             throw BoxIsEmpty{};
942         }
943         return *m_ptr;
944     }
945
946     T& operator*() {
947         if (m_ptr == nullptr) {
948             throw BoxIsEmpty{};
949         }
950         return *m_ptr;
951     }
952
953     const T& operator*() const {
954         if (m_ptr == nullptr) {
955             throw BoxIsEmpty{};
956         }
957         return *m_ptr;
958     }
959
960     T& operator*() {
961         if (m_ptr == nullptr) {
962             throw BoxIsEmpty{};
963         }
964         return *m_ptr;
965     }
966
967     const T& operator*() const {
968         if (m_ptr == nullptr) {
969             throw BoxIsEmpty{};
970         }
971         return *m_ptr;
972     }
973
974     T& operator*() {
975         if (m_ptr == nullptr) {
976             throw BoxIsEmpty{};
977         }
978         return *m_ptr;
979     }
980
981     const T& operator*() const {
982         if (m_ptr == nullptr) {
983             throw BoxIsEmpty{};
984         }
985         return *m_ptr;
986     }
987
988     T& operator*() {
989         if (m_ptr == nullptr) {
990             throw BoxIsEmpty{};
991         }
992         return *m_ptr;
993     }
994
995     const T& operator*() const {
996         if (m_ptr == nullptr) {
997             throw BoxIsEmpty{};
998         }
999         return *m_ptr;
1000

```

```

84         }
85         return *m_ptr;
86     }
87
88 private:
89     std::unique_ptr<T> m_ptr;
90 };
91
92 template <typename T>
93 Box(T&& value) -> Box<T>;
94
95 template <typename T>
96 Box(std::unique_ptr<T> ptr) -> Box<T>;
97
98 template <typename T, typename U>
99 bool operator==(const Box<T>& lhs, const Box<U>& rhs) {
100     return *lhs == *rhs;
101 }
102
103 template <typename T, typename U>
104 bool operator!=(const Box<T>& lhs, const Box<U>& rhs) {
105     return !(lhs == rhs);
106 }
107
108 // namespace wr22::utils

```

7.5 include/wr22/utils/nonconcurrent_semaphore.hpp File Reference

```

#include <cstdint>
#include <functional>
#include <optional>

```

Classes

- class [wr22::utils::NonconcurrentSemaphore](#)
- class [wr22::utils::NonconcurrentSemaphoreGuard](#)

Namespaces

- namespace [wr22](#)
- namespace [wr22::utils](#)

7.6 nonconcurrent_semaphore.hpp

[Go to the documentation of this file.](#)

```

1 #pragma once
2
3 // stl
4 #include <cstdint>
5 #include <functional>
6 #include <optional>
7
8 namespace wr22::utils {
9
10 class NonconcurrentSemaphoreGuard;
11
12 class NonconcurrentSemaphore {
13 public:
14     explicit NonconcurrentSemaphore(size_t max_capacity);
15
16     std::optional<NonconcurrentSemaphoreGuard> enter();
17
18 private:

```

```

19     bool raw_enter();
20     void raw_exit();
21
22     size_t m_occupied;
23     size_t m_max_capacity;
24
25     friend class NonconcurrentSemaphoreGuard;
26 };
27
28 class NonconcurrentSemaphoreGuard {
29 public:
30     NonconcurrentSemaphoreGuard(const NonconcurrentSemaphoreGuard&) = delete;
31     // TODO: undelete the move constructor if necessary.
32     NonconcurrentSemaphoreGuard(NonconcurrentSemaphoreGuard&& other);
33
34     NonconcurrentSemaphoreGuard& operator=(const NonconcurrentSemaphoreGuard&) = delete;
35     NonconcurrentSemaphoreGuard& operator=(NonconcurrentSemaphoreGuard&& other);
36
37     ~NonconcurrentSemaphoreGuard();
38
39 private:
40     explicit NonconcurrentSemaphoreGuard(NonconcurrentSemaphore& semaphore_ref);
41
42     void destroy();
43
44     std::reference_wrapper<NonconcurrentSemaphore> m_semaphore_ref;
45     bool m_is_active;
46
47     friend class NonconcurrentSemaphore;
48 };
49
50 } // namespace wr22::utils

```

7.7 src/box.cpp File Reference

```
#include <wr22/utils/box.hpp>
```

Namespaces

- namespace [wr22](#)
- namespace [wr22::utils](#)

7.8 src/nonconcurrent_semaphore.cpp File Reference

```
#include <wr22/utils/nonconcurrent_semaphore.hpp>
#include <stdexcept>
```

Namespaces

- namespace [wr22](#)
- namespace [wr22::utils](#)

Index

- [~NonconcurrentSemaphoreGuard](#)
 - [wr22::utils::NonconcurrentSemaphoreGuard, 21](#)
- [Adt](#)
 - [wr22::utils::Adt< Variants >, 14](#)
- [as_variant](#)
 - [wr22::utils::Adt< Variants >, 14, 15](#)
- [Box](#)
 - [wr22::utils, 10](#)
 - [wr22::utils::Box< T >, 17](#)
- [construct_in_place](#)
 - [wr22::utils::Box< T >, 18](#)
- [enter](#)
 - [wr22::utils::NonconcurrentSemaphore, 20](#)
- [include/wr22/utils/adt.hpp, 23, 24](#)
- [include/wr22/utils/box.hpp, 24, 25](#)
- [include/wr22/utils/nonconcurrent_semaphore.hpp, 26](#)
- [m_variant](#)
 - [wr22::utils::Adt< Variants >, 16](#)
- [MultiCallable](#)
 - [wr22::utils::detail::adt::MultiCallable< Fs >, 19](#)
- [NonconcurrentSemaphore](#)
 - [wr22::utils::NonconcurrentSemaphore, 20](#)
 - [wr22::utils::NonconcurrentSemaphoreGuard, 22](#)
- [NonconcurrentSemaphoreGuard](#)
 - [wr22::utils::NonconcurrentSemaphore, 20](#)
 - [wr22::utils::NonconcurrentSemaphoreGuard, 21](#)
- [operator!=](#)
 - [wr22::utils, 10](#)
- [operator*](#)
 - [wr22::utils::Box< T >, 18](#)
- [operator=](#)
 - [wr22::utils::NonconcurrentSemaphoreGuard, 22](#)
- [operator==](#)
 - [wr22::utils, 10](#)
- [src/box.cpp, 27](#)
- [src/nonconcurrent_semaphore.cpp, 27](#)
- [VariantType](#)
 - [wr22::utils::Adt< Variants >, 14](#)
- [visit](#)
 - [wr22::utils::Adt< Variants >, 15](#)
- [what](#)
 - [wr22::utils::BoxIsEmpty, 19](#)
 - [wr22, 9](#)
 - [wr22::utils, 9](#)
 - [Box, 10](#)
 - [operator!=, 10](#)
 - [operator==, 10](#)
 - [wr22::utils::Adt< Variants >, 13](#)
 - [Adt, 14](#)
 - [as_variant, 14, 15](#)
 - [m_variant, 16](#)
 - [VariantType, 14](#)
 - [visit, 15](#)
 - [wr22::utils::Box< T >, 16](#)
 - [Box, 17](#)
 - [construct_in_place, 18](#)
 - [operator*, 18](#)
 - [wr22::utils::BoxIsEmpty, 18](#)
 - [what, 19](#)
 - [wr22::utils::detail, 11](#)
 - [wr22::utils::detail::adt, 11](#)
 - [wr22::utils::detail::adt::MultiCallable< Fs >, 19](#)
 - [MultiCallable, 19](#)
 - [wr22::utils::NonconcurrentSemaphore, 20](#)
 - [enter, 20](#)
 - [NonconcurrentSemaphore, 20](#)
 - [NonconcurrentSemaphoreGuard, 20](#)
 - [wr22::utils::NonconcurrentSemaphoreGuard, 21](#)
 - [~NonconcurrentSemaphoreGuard, 21](#)
 - [NonconcurrentSemaphore, 22](#)
 - [NonconcurrentSemaphoreGuard, 21](#)
 - [operator=, 22](#)