Writing Regexps 2021-22 / Utils

Generated by Doxygen 1.9.3

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

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

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

Here is a list of all files with brief descriptions:

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/utils/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
  Derefencing operator: obtain a const reference to the stored value.
- T & operator* ()
  Derefencing 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 bahavior.

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

`other`	the Box from which to copy.

Exceptions

BoxIsEmpty	if `other` 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]
```

Derefencing operator: obtain a reference to the stored value.

Exceptions

BoxIsEmpty	if this `Box` 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]
```

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

Exceptions

BoxIsEmpty	if this `Box` does not contain a value at the moment.

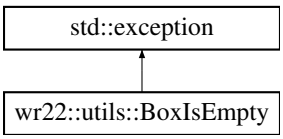
The documentation for this class was generated from the following file:

- 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]
```

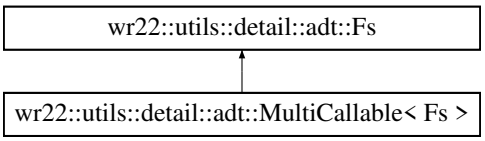
The documentation for this struct was generated from the following files:

- 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/utils/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
28 template <typename... Variants>
29 class Adt {
30 public:
32     using VariantType = std::variant<Variants...>;
33
43     template <typename V>
44     Adt(V variant) : m_variant(std::move(variant)) {}
45
74     template <typename... Fs>
75     decltype(auto) visit(Fs&&... visitors) const {
76         return std::visit(
77             detail::adt::MultiCallable<Fs...>(std::forward<Fs>(visitors)...),
78             m_variant);
79     }
80
87     template <typename... Fs>
88     decltype(auto) visit(Fs&&... visitors) {
89         return std::visit(
90             detail::adt::MultiCallable<Fs...>(std::forward<Fs>(visitors)...),
91             m_variant);
92     }
93
95     const VariantType& as_variant() const {
96         return m_variant;
97     }
98
100     VariantType& as_variant() {
101         return m_variant;
102     }
103
104 protected:
105     VariantType m_variant;
106 };
107
109 template <typename... Variants>
110 bool operator==(const Adt<Variants...>& lhs, const Adt<Variants...>& rhs) {
111     return lhs.as_variant() == rhs.as_variant();
112 }
113
115 template <typename... Variants>
116 bool operator!=(const Adt<Variants...>& lhs, const Adt<Variants...>& rhs) {
117     return !(lhs == rhs);
118 }
119
120 }  // 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
25 template <typename T>
26 class Box {
27 public:
32     explicit Box(T&& value) : m_ptr(std::make_unique<T>(std::forward<T>(value))) {}
33
49     explicit Box(std::unique_ptr<T> ptr) : m_ptr(std::move(ptr)) {}
50
57     template <typename Dummy = T>
58     Box(const Box& other) : m_ptr(std::make_unique<T>(*other)) {}
59
63     template <typename... Args>
64     static Box<T> construct_in_place(Args&&... args) {
65         return Box(std::make_unique<T>(std::forward<Args>(args)...));
66     }
67
71     const T& operator*() const {
72         if (m_ptr == nullptr) {
73             throw BoxIsEmpty{};
74         }
75         return *m_ptr;
76     }
77
81     T& operator*() {
82         if (m_ptr == nullptr) {
83             throw BoxIsEmpty{};
```

```
84          }
85          return *m_ptr;
86      }
87
88  private:
89      std::unique_ptr<T> m_ptr;
90  };
91
93  template <typename T>
94  Box(T&& value) -> Box<T>;
95
97  template <typename T>
98  Box(std::unique_ptr<T> ptr) -> Box<T>;
99
100 template <typename T, typename U>
101 bool operator==(const Box<T>& lhs, const Box<U>& rhs) {
102     return *lhs == *rhs;
103 }
104
105 template <typename T, typename U>
106 bool operator!=(const Box<T>& lhs, const Box<U>& rhs) {
107     return !(lhs == rhs);
108 }
109
110 }  // namespace wr22::utils
```

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

```
#include <cstddef>
#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 <cstddef>
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