Result-CPP

0.1.0

Generated by Doxygen 1.10.0

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 4

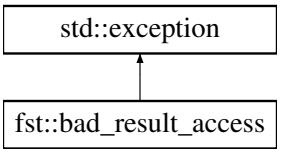
Class Documentation

4.1 fst::bad_result_access Class Reference

Exception class for indicating invalid access to a result object.

`#include <result.hpp>`

Inheritance diagram for fst::bad_result_access:



Public Member Functions

- **bad_result_access** (const char *reason) noexcept
- **bad_result_access** (const std::string &reason) noexcept
- const char * **what** () const noexcept override

4.1.1 Detailed Description

Exception class for indicating invalid access to a result object.

This exception is thrown when attempting to access the value or error of a result object in an invalid state, such as when trying to access the success value of an error result or vice versa.

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

- G:/DevelopmentEnvironment/C Projects/result-cpp/include/fst/result.hpp

4.2 fst::result< T, E > Class Template Reference

Generic class that implements the monadic pattern for error handling. It can store either a successful value of type T or an error value of type E. The class includes methods for extracting the success or error values safely and handling different outcomes through chaining operations.

`#include <result.hpp>`

Public Types

- using **value_type** = T
- using **error_type** = E

Public Member Functions

- template<typename U = T, std::enable_if<!std::is_same_v< U, E > > * = nullptr>
  constexpr result (const T &value)

  Constructor for a successful result with a value.
- template<typename U = E, std::enable_if<!std::is_same_v< T, U > > * = nullptr>
  constexpr result (const E &error)

  Constructor for a failed result with an error value.
- constexpr result (success_tag tag, const T &value)

  Constructor for a successful result with a value, using a success tag.
- constexpr result (error_tag tag, const E &error)

  Constructor for a failed result with an error value, using an error tag.
- **result** (result< T, E > &res)
- **result** (result< T, E > &&res)
- constexpr const std::optional< T > success () const

  Retrieves the success value if the result is in a success state.
- constexpr const std::optional< E > error () const

  Retrieves the error value if the result is in an error state.
- constexpr const T & value () const

  Retrieves the success value of the result.
- constexpr const T value_or (const T &default_value=T{}) const

  Retrieves the success value of the result; otherwise, returns a default value.
- constexpr const T & expect (const std::string &message) const

  Retrieves the success value if the result is in a success state.
- constexpr const result_state & state () const

  Retrieves the state of the result (success or error).
- constexpr bool has_value () const

  Checks if the result contains a success value.
- constexpr bool has_error () const

  Checks if the result contains an error value.
- constexpr bool is_empty () const

  Checks if the result is empty.
- template<typename F >
  constexpr auto or_else (F &&f) const -> result< T, typename decltype(f(*error()))::error_type >

  Applies the provided function to the error value if the result is in an error state, returning a new result.
- template<typename F >
  constexpr auto and_then (F &&f) const -> result< typename decltype(f(*success()))::value_type, E >

Applies the provided callable function to the success value if the result is in a success state, returning a new result.

- template<typename F >
  constexpr auto map (F &&f)

  Maps the success value using the provided function.
- template<typename F >
  constexpr auto map_error (F &&f)

  Maps the error value using the provided function.
- template<typename F >
  constexpr auto transform (F &&f) const -> result< typename decltype(f(*this))::value_type, typename decltype(f(*this))::error_type >

  Transforms the result using the provided callable function.
- template<typename F >
  constexpr result< T, E > inspect (F &&f) const

  Invokes a callable function on the current result without modifying it.
- operator bool () const

  Explicit conversion to bool.
- const T & operator* () const

  Dereference operator.
- template<typename U >
  constexpr operator result< U, E > () const

  Converts the result to a different result type, mapping the success value with the provided conversion function.
- template<typename U >
  constexpr operator result< T, U > () const

  Converts the result to a different result type, mapping the error value with the provided conversion function.

Friends

- std::ostream & **operator**<< (std::ostream &os, const result< T, E > &res)

4.2.1 Detailed Description

template<typename T, typename E>
class fst::result< T, E >

Generic class that implements the monadic pattern for error handling. It can store either a successful value of type T or an error value of type E. The class includes methods for extracting the success or error values safely and handling different outcomes through chaining operations.

Template Parameters

T	Type of the success value.
E	Type of the error value.

4.2.2 Constructor & Destructor Documentation

4.2.2.1 result() [1/4]

```
template<typename T , typename E >
template<typename U = T, std::enable_if<!std::is_same_v< U, E > > * = nullptr>
```

```
constexpr fst::result< T, E >::result (
            const T & value )  [inline], [constexpr]
```

Constructor for a successful result with a value.

Template Parameters

T	The type of the success value.

Parameters

value	The success value to be stored.

4.2.2.2 result() [2/4]

```
template<typename T , typename E >
template<typename U = E, std::enable_if<!std::is_same_v< T, U > > * = nullptr>
constexpr fst::result< T, E >::result (
            const E & error )  [inline], [constexpr]
```

Constructor for a failed result with an error value.

Template Parameters

E	The type of the error value.

Parameters

error	The error value to be stored.

4.2.2.3 result() [3/4]

```
template<typename T , typename E >
constexpr fst::result< T, E >::result (
            success_tag tag,
            const T & value )  [inline], [constexpr]
```

Constructor for a successful result with a value, using a success tag.

This constructor is particularly useful in cases where the type of the success value (T) is the same as the type of the error value (E) or when there might be ambiguity in type conversion.

Template Parameters

T	The type of the success value.

Parameters

tag	The success tag, indicating a successful result.

Parameters

value	The success value to be stored.

4.2.2.4 result() [4/4]

```
template<typename T , typename E >
constexpr fst::result< T, E >::result (
            error_tag tag,
            const E & error )  [inline], [constexpr]
```

Constructor for a failed result with an error value, using an error tag.

This constructor is particularly useful in cases where the type of the success value (T) is the same as the type of the error value (E) or when there might be ambiguity in type conversion.

Template Parameters

E	The type of the error value.

Parameters

tag	The error tag, indicating an error result.
error	The error value to be stored.

4.2.3 Member Function Documentation

4.2.3.1 and_then()

```
template<typename T , typename E >
template<typename F >
constexpr auto fst::result< T, E >::and_then (
            F && f ) const -> result<typename decltype(f(*success()))::value_type, E>   [inline],
[constexpr]
```

Applies the provided callable function to the success value if the result is in a success state, returning a new result.

If the result is in an error state, returns the original error result. If the result is in a success state, applies the provided callable function to the success value and returns the result.

Template Parameters

F	Type of the callable function.

Parameters

f	Callable function to be applied to the success value if the result is in a success state.

Returns

The original error result or the result of applying the callable function to the success value.

Note

The provided callable function must have the following signatures: `auto func(const T& value) -> result<U, E>`, `auto func(const auto& value) -> result<U, E>`, `auto func(T value) -> result<U, E>`, or `auto func(auto value) -> result<U, E>` where U is the desired success value type and E is the original error value type.

4.2.3.2 error()

```
template<typename T , typename E >
constexpr const std::optional< E > fst::result< T, E >::error ( ) const  [inline], [constexpr]
```

Retrieves the error value if the result is in an error state.

Template Parameters

T	Type of the success value.
E	Type of the error value.

Returns

An optional containing the error value if available, otherwise std::nullopt.

4.2.3.3 expect()

```
template<typename T , typename E >
constexpr const T & fst::result< T, E >::expect (
            const std::string & message ) const  [inline], [constexpr]
```

Retrieves the success value if the result is in a success state.

Template Parameters

T	Type of the success value.
E	Type of the error value.

Parameters

message	A message to include in the exception if the result is in an error state.

Returns

A const reference to the success value.

Exceptions

std::runtime_error	If the result is not in a success state, including the specified message.

4.2.3.4 has_error()

```
template<typename T , typename E >
constexpr bool fst::result< T, E >::has_error ( ) const  [inline], [constexpr]
```

Checks if the result contains an error value.

Template Parameters

T	Type of the success value.
E	Type of the error value.

Returns

True if the result is in an error state; otherwise, false.

4.2.3.5 has_value()

```
template<typename T , typename E >
constexpr bool fst::result< T, E >::has_value ( ) const  [inline], [constexpr]
```

Checks if the result contains a success value.

Template Parameters

T	Type of the success value.
E	Type of the error value.

Returns

True if the result is in a success state; otherwise, false.

4.2.3.6 inspect()

```
template<typename T , typename E >
template<typename F >
constexpr result< T, E > fst::result< T, E >::inspect (
            F && f ) const  [inline], [constexpr]
```

Invokes a callable function on the current result without modifying it.

This function calls the provided function `f` with a constant reference to the current result without modifying it. It is designed for cases where side effects are intended without altering the result.

Parameters

f	A function that takes a constant reference to the result. The function should have the signatures: `void f(const result<T,E>& res)` or `void f(const auto& res)`.

Returns

The unmodified current result after invoking the function.

4.2.3.7 is_empty()

```
template<typename T , typename E >
constexpr bool fst::result< T, E >::is_empty ( ) const  [inline], [constexpr]
```

Checks if the result is empty.

Template Parameters

T	Type of the success value.
E	Type of the error value.

Returns

True if the result is empty; otherwise, false.

4.2.3.8 map()

```
template<typename T , typename E >
template<typename F >
constexpr auto fst::result< T, E >::map (
            F && f )  [inline], [constexpr]
```

Maps the success value using the provided function.

If the result is in a success state, applies the provided function `f` to the success value, modifying it in place.

Template Parameters

T	Type of the success value.
E	Type of the error value.

Parameters

f	Callable function to be applied to the success value if the result is in a success state.

Returns

Reference to the modified current result.

Note

The provided callable function must have the following signatures: `auto f(const T& value) -> T` or `auto f(const auto& value) -> T` where T is the type of the success value.

4.2.3.9 map_error()

```
template<typename T , typename E >
template<typename F >
constexpr auto fst::result< T, E >::map_error (
            F && f )  [inline], [constexpr]
```

Maps the error value using the provided function.

If the result is in an error state, applies the provided function `f` to the error value, modifying it in place.

Template Parameters

T	Type of the success value.
E	Type of the error value.

Parameters

f	Callable function to be applied to the error value if the result is in an error state.

Returns

Reference to the modified current result.

Note

The provided callable function must have the following signatures: `f(const E& error) -> E` or `f(const auto& error) -> E` where E is the type of the error value.

4.2.3.10 operator bool()

```
template<typename T , typename E >
fst::result< T, E >::operator bool ( ) const  [inline], [explicit]
```

Explicit conversion to bool.

Returns

True if the result is in a success state; otherwise, false.

4.2.3.11 operator result< T, U >()

```
template<typename T , typename E >
template<typename U >
constexpr fst::result< T, E >::operator result< T, U > ( ) const  [inline], [constexpr]
```

Converts the result to a different result type, mapping the error value with the provided conversion function.

Template Parameters

T	Type of the success value in the original result.
U	Type of the error value in the new result.

Returns

A new result with the error value converted to type E.

4.2.3.12 operator result< U, E >()

```
template<typename T , typename E >
template<typename U >
constexpr fst::result< T, E >::operator result< U, E > ( ) const  [inline], [constexpr]
```

Converts the result to a different result type, mapping the success value with the provided conversion function.

Template Parameters

U	Type of the success value in the new result.
E	Type of the error value in the original result.

Returns

A new result with the success value converted to type U.

4.2.3.13 operator*()

```
template<typename T , typename E >
const T & fst::result< T, E >::operator* ( ) const  [inline]
```

Dereference operator.

Returns

Const reference to the success value.

Exceptions

bad_result_access	If the result is not in a success state.

4.2.3.14 or_else()

```
template<typename T , typename E >
template<typename F >
constexpr auto fst::result< T, E >::or_else (
            F && f ) const -> result<T, typename decltype(f(*error()))::error_type>   [inline],
[constexpr]
```

Applies the provided function to the error value if the result is in an error state, returning a new result.

If the result is in a success state, returns the original success result. If the result is in an error state, applies the provided callable function to the error value and returns the result.

Template Parameters

F	Type of the callable function.

Parameters

f	Callable function to be applied to the error value if the result is in an error state.

Returns

The original success result or the result of applying the callable function to the error value.

Note

The provided callable function must have the signature: `auto lambda(const E& error) -> result<T, U>`, `auto lambda(const auto& error) -> result<T, U>`, `auto lambda(E error) -> result<T, U>` or `auto lambda(auto error) -> result<T, U>` where T is the original success value type and U is the desired error value type.

4.2.3.15 state()

```
template<typename T , typename E >
constexpr const result_state & fst::result< T, E >::state ( ) const  [inline], [constexpr]
```

Retrieves the state of the result (success or error).

Template Parameters

T	Type of the success value.
E	Type of the error value.

Returns

The state of the result.

4.2.3.16 success()

```
template<typename T , typename E >
constexpr const std::optional< T > fst::result< T, E >::success ( ) const  [inline], [constexpr]
```

Retrieves the success value if the result is in a success state.

Template Parameters

T	Type of the success value.
E	Type of the error value.

Returns

An optional containing the success value if available, otherwise std::nullopt.

4.2.3.17 transform()

```
template<typename T , typename E >
template<typename F >
constexpr auto fst::result< T, E >::transform (
            F && f ) const -> result<typename decltype(f(*this))::value_type, typename decltype(f(*this))↩
::error_type>   [inline], [constexpr]
```

Transforms the result using the provided callable function.

Applies the callable function to the current result, producing a new result based on the transformation. The transformation function is expected to take the current result as an argument and return a new result.

Template Parameters

F	Type of the callable function.

Parameters

f	Callable function to transform the current result.

Returns

The result of applying the transformation function to the current result.

Note

The provided callable function must have the signatures: `func(const result<T, E>& res) -> result<U, V>` or `func(const auto& res) -> result<U, V>`, where U is the desired success value type and V is the desired error value type for the transformed result.

4.2.3.18 value()

```
template<typename T , typename E >
constexpr const T & fst::result< T, E >::value ( ) const  [inline], [constexpr]
```

Retrieves the success value of the result.

Template Parameters

T	Type of the success value.
E	Type of the error value.

Returns

The const reference to success value.

Exceptions

std::bad_result_access	if result does not contain a success value.

4.2.3.19 value_or()

```
template<typename T , typename E >
constexpr const T fst::result< T, E >::value_or (
            const T & default_value = T{} ) const  [inline], [constexpr]
```

Retrieves the success value of the result; otherwise, returns a default value.

Template Parameters

T	Type of the success value.
E	Type of the error value.

Parameters

default_value	The value to return if the result is in an error state.

Returns

The success value if available; otherwise, the specified default value.

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

- G:/DevelopmentEnvironment/C Projects/result-cpp/include/fst/result.hpp

Chapter 5

File Documentation

5.1 result.hpp

```
00001 // Result.h
00002 #ifndef FST_RESULT_HPP
00003 #define FST_RESULT_HPP
00004
00005 #include <exception>
00006 #include <iostream>
00007 #include <optional>
00008 #include <stdexcept>
00009 #include <string>
00010 #include <type_traits>
00011
00012 namespace fst {
00013
00019 enum class result_state : unsigned char { empty, success, error };
00020
00021 // Enum to represent the success state tag.
00022 enum class success_tag : unsigned char { success };
00023
00024 // Enum to represent the error state tag.
00025 enum class error_tag : unsigned char { error };
00026
00027 // Alias for the success tag value.
00028 constexpr success_tag success_t = success_tag::success;
00029
00030 // Alias for the error tag value.
00031 constexpr error_tag error_t = error_tag::error;
00032
00039 std::string to_string(const result_state& state) {
00040   switch (state) {
00041     case result_state::empty:
00042       return "empty";
00043     case result_state::success:
00044       return "success";
00045     case result_state::error:
00046       return "error";
00047     default:
00048       return "unknown";
00049   }
00050 }
00051
00060 std::ostream& operator<<(std::ostream& os, const result_state& state) {
00061   return os << to_string(state);
00062 }
00063
00071 class bad_result_access : public std::exception {
00072  public:
00073   bad_result_access() noexcept {}
00074   bad_result_access(const char* reason) noexcept : m_reason(reason) {}
00075   bad_result_access(const std::string& reason) noexcept
00076       : m_reason(reason.c_str()) {}
00077   const char* what() const noexcept override { return m_reason; }
00078
00079  private:
00080   const char* m_reason = "bad result access";
00081 };
00082
00092 template <typename T, typename E>
00093 class result final {
```

```
00094  public:
00095   using value_type = T;
00096   using error_type = E;
00097
00098   // Default constructor, creates an empty result
00099   constexpr result() : m_state(result_state::empty), m_self(success_t, T{}) {}
00100
00107   template <typename U = T, std::enable_if<!std::is_same_v<U, E>>* = nullptr>
00108   constexpr result(const T& value)
00109       : m_state(result_state::success), m_self(success_t, value) {}
00110
00117   template <typename U = E, std::enable_if<!std::is_same_v<T, U>>* = nullptr>
00118   constexpr result(const E& error)
00119       : m_state(result_state::error), m_self(error_t, error) {}
00120
00133   constexpr result(success_tag tag, const T& value)
00134       : m_state(result_state::success), m_self(success_t, value) {}
00135
00148   constexpr result(error_tag tag, const E& error)
00149       : m_state(result_state::error), m_self(error_t, error) {}
00150
00151   result(result<T, E>& res) : m_state(res.state()) {
00152     switch (res.state()) {
00153       case result_state::success:
00154         m_self.m_value = res.m_self.m_value;
00155         break;
00156       case result_state::error:
00157         m_self.m_error = res.m_self.m_error;
00158         break;
00159       case result_state::empty:
00160         break;
00161       default:
00162         throw bad_result_access("Invalid state in copy constructor");
00163     }
00164   }
00165
00166   result(result<T, E>&& res) : m_state(res.state()) {
00167     switch (res.m_state) {
00168       case result_state::success:
00169         m_self.m_value = std::move(res.m_self.m_value);
00170         break;
00171       case result_state::error:
00172         m_self.m_error = std::move(res.m_self.m_error);
00173         break;
00174       case result_state::empty:
00175         break;
00176       default:
00177         throw bad_result_access("Invalid state in move constructor");
00178     }
00179     res.m_state = result_state::empty;
00180   }
00181
00182   ~result() {
00183     std::cout « "«DESTROYED»: " « *this « " @ STATE: " « m_state « '\n';
00184     switch (m_state) {
00185       case result_state::success:
00186         m_self.m_value.~T();
00187         break;
00188       case result_state::error:
00189         m_self.m_error.~E();
00190         break;
00191       default:
00192         break;
00193     }
00194   }
00195
00203   [[nodiscard]] constexpr const std::optional<T> success() const {
00204     return m_state == result_state::success ? std::optional<T>(m_self.m_value)
00205                                             : std::nullopt;
00206   }
00207
00215   [[nodiscard]] constexpr const std::optional<E> error() const {
00216     return m_state == result_state::error ? std::optional<E>(m_self.m_error)
00217                                           : std::nullopt;
00218   }
00219
00227   [[nodiscard]] constexpr const T& value() const {
00228     return m_state == result_state::success
00229                ? m_self.m_value
00230                : throw bad_result_access(
00231                      "Invalid state for value access, result's state was: " +
00232                      to_string(m_state));
00233   }
00234
00245   [[nodiscard]] constexpr const T value_or(const T& default_value = T{}) const {
00246     return m_state == result_state::success ? m_self.m_value : default_value;
00247   }
```

```
00248
00259   [[nodiscard]] constexpr const T& expect(const std::string& message) const {
00260     return m_state == result_state::success ? m_self.m_value
00261                                             : throw std::runtime_error(message);
00262   }
00263
00270   [[nodiscard]] constexpr const result_state& state() const { return m_state; }
00271
00278   [[nodiscard]] constexpr bool has_value() const {
00279     return m_state == result_state::success;
00280   }
00281
00288   [[nodiscard]] constexpr bool has_error() const {
00289     return m_state == result_state::error;
00290   }
00291
00298   [[nodiscard]] constexpr bool is_empty() const {
00299     return m_state == result_state::empty;
00300   }
00301
00324   template <typename F>
00325   constexpr auto or_else(F&& f) const
00326       -> result<T, typename decltype(f(*error()))::error_type> {
00327     return m_state == result_state::error
00328                ? f(m_self.m_error)
00329                : decltype(f(*error()))(success_t, value_or());
00330   }
00331
00354   template <typename F>
00355   constexpr auto and_then(F&& f) const
00356       -> result<typename decltype(f(*success()))::value_type, E> {
00357     return m_state == result_state::success
00358                ? f(m_self.m_value)
00359                : decltype(f(*success()))(error_t, m_self.m_error);
00360   }
00361
00379   template <typename F>
00380   constexpr auto map(F&& f) {
00381     return m_state == result_state::success
00382                ? result<T, E>(success_t, f(m_self.m_value))
00383            : m_state == result_state::error
00384                ? result<T, E>(error_t, m_self.m_error)
00385                : result<T, E>();
00386   }
00387
00405   template <typename F>
00406   constexpr auto map_error(F&& f) {
00407     return m_state == result_state::error
00408                ? result<T, E>(error_t, f(m_self.m_error))
00409            : m_state == result_state::success
00410                ? result<T, E>(success_t, m_self.m_value)
00411                : result<T, E>();
00412   }
00413
00432   template <typename F>
00433   constexpr auto transform(F&& f) const
00434       -> result<typename decltype(f(*this))::value_type,
00435                 typename decltype(f(*this))::error_type> {
00436     return f(*this);
00437     ;
00438   }
00439
00456   template <typename F>
00457   constexpr result<T, E> inspect(F&& f) const {
00458     f(*this);
00459     return m_state == result_state::success
00460                ? result<T, E>(success_t, m_self.m_value)
00461            : m_state == result_state::error
00462                ? result<T, E>(error_t, m_self.m_error)
00463                : result<T, E>();
00464   }
00465
00470   explicit operator bool() const { return m_state == result_state::success; }
00471
00477   const T& operator*() const { return *success(); }
00478
00486   template <typename U>
00487   constexpr operator result<U, E>() const {
00488     return result<U, E>(U(value()));
00489   }
00490
00498   template <typename U>
00499   constexpr operator result<T, U>() const {
00500     return result<T, U>(U(*error()));
00501   }
00502
00503   friend std::ostream& operator<<(std::ostream& os, const result<T, E>& res) {
```

```
00504     switch (res.state()) {
00505       case result_state::success:
00506         return os « *res.success();
00507
00508       case result_state::error:
00509         return os « *res.error();
00510
00511       case result_state::empty:
00512         return os;
00513
00514       default:
00515         throw bad_result_access();
00516     }
00517   }
00518
00519  private:
00520   result_state m_state = result_state::empty;
00521   union members {
00522     T m_value;
00523     E m_error;
00524     members() {}
00525     members(success_tag tag, T val) : m_value(val) {}
00526     members(error_tag tag, E err) : m_error(err) {}
00527     ~members() {}
00528   } m_self;
00529 };
00530
00531 }  // namespace fst
00532
00533 #endif  // FST_RESULT_HPP
```

Index