External Sort

1.0

Generated by Doxygen 1.9.8

Chapter 1

Namespace Index

1.1 Namespace List

Here is a list of all namespaces with brief descriptions:

external_sort . . . . . ??
external_sort::detail . . . . . ??
io . . . . . ??
io::detail . . . . . ??
logging . . . . . ??
logging::detail . . . . . ??
serialization . . . . . ??
serialization::detail . . . . . ??
serialization::detail_adl
Helpers for ADL - defined in a nested namespace . . . . . ??

Chapter 2

Concept Index

2.1 Concepts

Here is a list of all concepts with brief descriptions:

serialization::CustomSerializable
: Concept for types with custom serialization through free functions . . . . . **??**

serialization::FileSerializable
: Concept for types that can be serialized into files . . . . . **??**

serialization::MethodSerializable
: Concept for types with serialization methods . . . . . **??**

serialization::PodSerializable
: Concept for POD types that can be serialized through fwrite/fread . . . . . **??**

serialization::SpecializedSerializable
: Concept for types with Serializer specialization . . . . . **??**

Chapter 3

Hierarchical Index

3.1 Class Hierarchy

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

serialization::detail::ByteCountingFile . . . . . ??
ConsoleLogger . . . . . ??
io::ElementBuffer< T > . . . . . ??
FileLogger . . . . . ??
io::IInputStream< T > . . . . . ??
    io::FileInputStream< T > . . . . . ??
    io::InMemoryInputStream< T > . . . . . ??
logging::ILogger . . . . . ??
    logging::LoggerAdapter< UserLogger > . . . . . ??
    logging::NullLogger . . . . . ??
    logging::SpdlogWrapper . . . . . ??
logging::SpdlogWrapper::Impl . . . . . ??
io::IOutputStream< T > . . . . . ??
    io::FileOutputStream< T > . . . . . ??
    io::InMemoryOutputStream< T > . . . . . ??
io::IStreamFactory< T > . . . . . ??
    io::FileStreamFactory< T > . . . . . ??
    io::InMemoryStreamFactory< T > . . . . . ??
external_sort::KWayMergeSorter< T > . . . . . ??
external_sort::MergeSource< T > . . . . . ??
external_sort::KWayMergeSorter< T >::MergeSourceComparator . . . . . ??
external_sort::MergeSourceComparatorStatic< T > . . . . . ??
serialization::Serializer< T > . . . . . ??
    serialization::CustomFunctionSerializer< T > . . . . . ??
    serialization::MethodSerializer< T > . . . . . ??
    serialization::PodSerializer< T > . . . . . ??
serialization::Serializer< std::string > . . . . . ??
serialization::Serializer< std::vector< T > > . . . . . ??
io::TempFileManager . . . . . ??

Chapter 4

Class Index

4.1 Class List

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

serialization::detail::ByteCountingFile
: Helper class to calculate serialized size by writing to a null device . . . ??

ConsoleLogger
: Custom console logger with colored output and message prefixes . . . ??

serialization::CustomFunctionSerializer< T >
: Serializer for types with free function serialization . . . ??

io::ElementBuffer< T >
: A template class for a buffer of elements of type T . . . ??

io::FileInputStream< T >
: File implementation of an input stream . . . ??

FileLogger
: Non-copyable file logger that writes to a log file . . . ??

io::FileOutputStream< T >
: File implementation of an output stream . . . ??

io::FileStreamFactory< T >
: Factory for file streams . . . ??

io::IInputStream< T >
: Interface for input stream of elements of type T . . . ??

logging::ILogger
: Abstract base class defining the logger interface . . . ??

logging::SpdlogWrapper::Impl
: Implementation class hiding spdlog dependencies . . . ??

io::InMemoryInputStream< T >
: In-memory implementation of input stream . . . ??

io::InMemoryOutputStream< T >
: In-memory implementation of output stream . . . ??

io::InMemoryStreamFactory< T >
: Factory for in-memory streams . . . ??

io::IOutputStream< T >
: Interface for output stream of elements of type T . . . ??

io::IStreamFactory< T >
: Interface for a stream factory . . . ??

external_sort::KWayMergeSorter< T >
: Template class for performing k-way external merge sort . . . ??

logging::LoggerAdapter< UserLogger >
: Adapter class that wraps a user-defined logger to conform to the ILogger interface . . . ??

external_sort::MergeSource< T >
Structure representing a data source for merging . . . . . ??
external_sort::KWayMergeSorter< T >::MergeSourceComparator
Comparator for merge sources . . . . . ??
external_sort::MergeSourceComparatorStatic< T >
Comparator for merge sources that compares the current value of the streams . . . . . ??
serialization::MethodSerializer< T >
Serializer for types with method serialization . . . . . ??
logging::NullLogger
A logger implementation that discards all log messages . . . . . ??
serialization::PodSerializer< T >
Serializer for POD types . . . . . ??
serialization::Serializer< T >
Base interface for serializable types . . . . . ??
serialization::Serializer< std::string > . . . . . ??
serialization::Serializer< std::vector< T > > . . . . . ??
logging::SpdlogWrapper
Concrete logger implementation wrapping the spdlog library . . . . . ??
io::TempFileManager
A manager for temporary files . . . . . ??

Chapter 5

File Index

5.1 File List

Here is a list of all files with brief descriptions:

external_sort/examples/example.cpp . . . ??
external_sort/include/external_sort_logging.hpp . . . ??
external_sort/include/k_way_merge_sorter.hpp
K-way external merge sort . . . ??
io/include/element_buffer.hpp
Element buffer template class . . . ??
io/include/file_stream.hpp
File implementations of input/output streams . . . ??
io/include/interfaces.hpp
Base interfaces for input and output strems and for stream factories . . . ??
io/include/io_logging.hpp
Logging integration for io library . . . ??
io/include/memory_stream.hpp
In-memory implementations of input/output streams . . . ??
io/include/storage_types.hpp
Definitions of different identificators for files . . . ??
io/include/temp_file_manager.hpp
Temporary file manager . . . ??
logging/include/ILogger.hpp
Interface definition for all logger implementations . . . ??
logging/include/LoggerAdapter.hpp
Wrapper for any user-defined logger to be used with the logging library . . . ??
logging/include/NullLogger.hpp
Empty no-op logger implementation for when logging is disabled . . . ??
logging/include/Registry.hpp
Global logger registry for setting and managing the active logger instance . . . ??
logging/include/SpdlogWrapper.hpp
Wrapper for spdlog library using PIMPL pattern . . . ??
logging/src/DefaultLogger.cpp
Implementation of default logger factory function . . . ??
logging/src/SpdlogWrapper.cpp
Implementation of the spdlog wrapper . . . ??
serialization/examples/example.cpp . . . ??
serialization/include/serialization_logging.hpp
Logging integration for serialization library . . . ??

serialization/include/serializers.hpp
Serializers implementations for some types . . . . . ??
serialization/include/type_concepts.hpp
Concepts for checking types in serialization library . . . . . ??
serialization/src/serialization_logging.cpp
Implementation of logging functions for serialization library . . . . . ??

Chapter 6

Namespace Documentation

6.1 external_sort Namespace Reference

Namespaces

- namespace detail

Classes

- class KWayMergeSorter

  Template class for performing k-way external merge sort.
- struct MergeSource

  Structure representing a data source for merging.
- struct MergeSourceComparatorStatic

  Comparator for merge sources that compares the current value of the streams.

6.2 external_sort::detail Namespace Reference

Functions

- void LogError (const std::string &message)

  Log an error message from External Sort operations.
- void LogWarning (const std::string &message)

  Log a warning message from io operations.
- void LogInfo (const std::string &message)

  Log an informational message from External Sort operations.

6.2.1 Function Documentation

6.2.1.1 LogError()

```
void external_sort::detail::LogError (
            const std::string & message )  [inline]
```

Log an error message from External Sort operations.

Parameters

message	The error message External Sort log

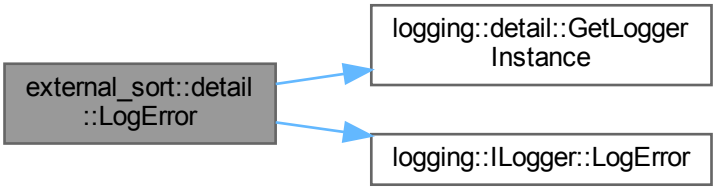
Definition at line 18 of file external_sort_logging.hpp.

```
00018                                                {
00019     auto& logger = logging::detail::GetLoggerInstance();
00020     if (logger) {
00021         logger->LogError("[External Sort] " + message);
00022     }
00023 }
```

References logging::detail::GetLoggerInstance(), and logging::ILogger::LogError().

Referenced by external_sort::KWayMergeSorter< T >::Sort().

Here is the call graph for this function:



Here is the caller graph for this function:

6.2.1.2 LogInfo()

```
void external_sort::detail::LogInfo (
            const std::string & message )  [inline]
```

Log an informational message from External Sort operations.

Parameters

message	The info message to log

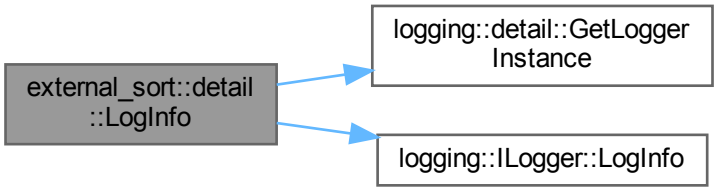
Definition at line 40 of file external_sort_logging.hpp.

```
00040                                                {
00041     auto& logger = logging::detail::GetLoggerInstance();
00042     if (logger) {
00043         logger->LogInfo("[External Sort] " + message);
00044     }
00045 }
```

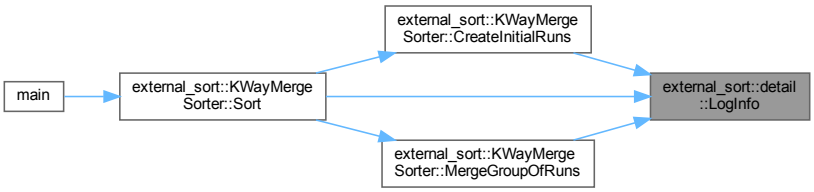
References logging::detail::GetLoggerInstance(), and logging::ILogger::LogInfo().

Referenced by external_sort::KWayMergeSorter< T >::CreateInitialRuns(), external_sort::KWayMergeSorter< T >::MergeGroupOfR and external_sort::KWayMergeSorter< T >::Sort().

Here is the call graph for this function:



Here is the caller graph for this function:



6.2.1.3 LogWarning()

```
void external_sort::detail::LogWarning (
            const std::string & message )  [inline]
```

Log a warning message from io operations.

Parameters

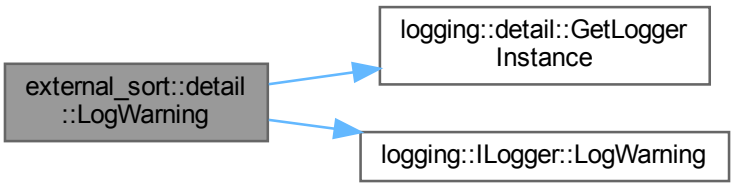
message	The warning message External Sort log

Definition at line 29 of file external_sort_logging.hpp.

```
00029                                                 {
00030     auto& logger = logging::detail::GetLoggerInstance();
00031     if (logger) {
00032         logger->LogWarning("[External Sort] " + message);
00033     }
00034 }
```

References logging::detail::GetLoggerInstance(), and logging::ILogger::LogWarning().

Here is the call graph for this function:



6.3 io Namespace Reference

Namespaces

- namespace detail

Classes

- class ElementBuffer

  A template class for a buffer of elements of type T.
- class FileInputStream

  File implementation of an input stream.
- class FileOutputStream

  File implementation of an output stream.
- class FileStreamFactory

  Factory for file streams.
- class IInputStream

  Interface for input stream of elements of type T.
- class InMemoryInputStream

  In-memory implementation of input stream.
- class InMemoryOutputStream

  In-memory implementation of output stream.
- class InMemoryStreamFactory

  Factory for in-memory streams.
- class IOutputStream

  Interface for output stream of elements of type T.
- class IStreamFactory

  Interface for a stream factory.
- class TempFileManager

  A manager for temporary files.

Typedefs

- using StorageId = std::string

  Alias for std::string to identify files by their name.

6.3.1 Typedef Documentation

6.3.1.1 StorageId

```
using io::StorageId = typedef std::string
```

Alias for std::string to identify files by their name.

Definition at line 15 of file storage_types.hpp.

6.4 io::detail Namespace Reference

Functions

- void LogError (const std::string &message)

  Log an error message from io operations.
- void LogWarning (const std::string &message)

  Log a warning message from io operations.
- void LogInfo (const std::string &message)

  Log an informational message from io operations.

6.4.1 Function Documentation

6.4.1.1 LogError()

```
void io::detail::LogError (
            const std::string & message )  [inline]
```

Log an error message from io operations.

Parameters

message	The error message to log

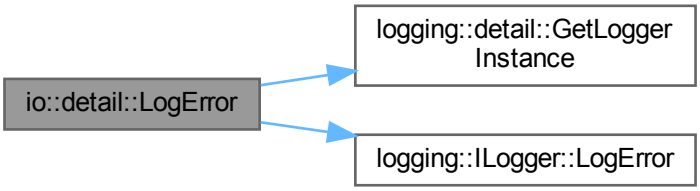
Definition at line 18 of file io_logging.hpp.

```
00018                                                {
00019     auto& logger = logging::detail::GetLoggerInstance();
00020     if (logger) {
00021         logger->LogError("[IO] " + message);
00022     }
00023 }
```

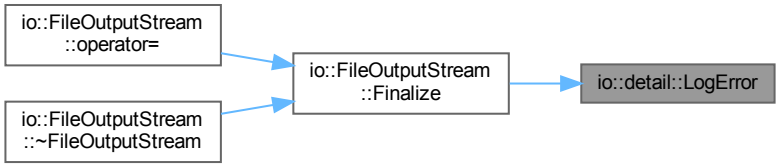
References logging::detail::GetLoggerInstance(), and logging::ILogger::LogError().

Referenced by io::FileOutputStream< T >::Finalize().

Here is the call graph for this function:



Here is the caller graph for this function:



6.4.1.2 LogInfo()

```
void io::detail::LogInfo (
            const std::string & message )  [inline]
```

Log an informational message from io operations.

Parameters

message	The info message to log

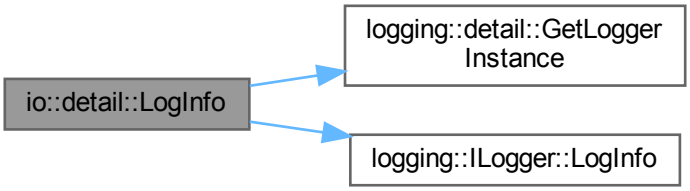
Definition at line 40 of file io_logging.hpp.

```
00040                                                {
00041     auto& logger = logging::detail::GetLoggerInstance();
00042     if (logger) {
00043         logger->LogInfo("[IO] " + message);
00044     }
00045 }
```

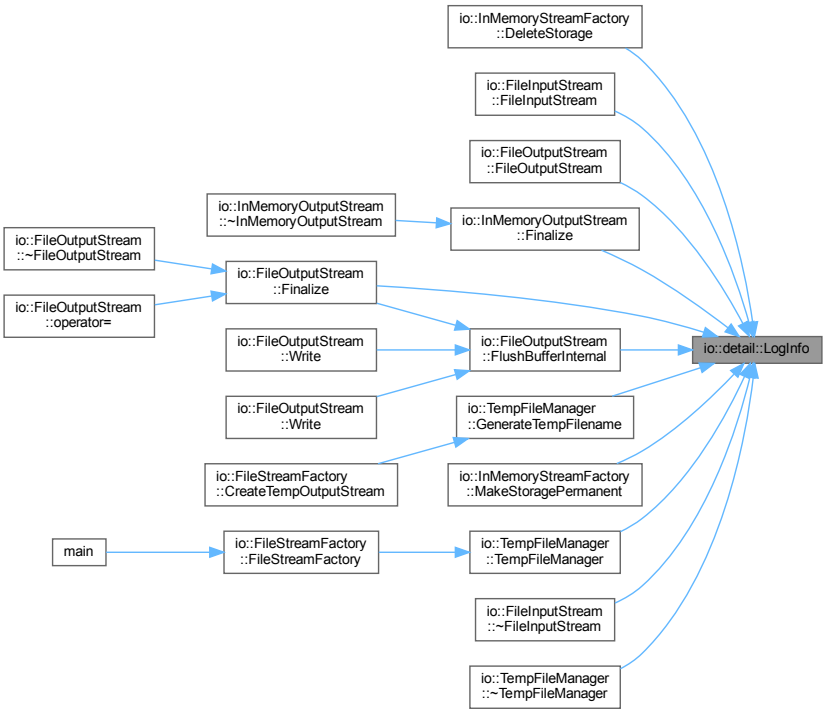
References logging::detail::GetLoggerInstance(), and logging::ILogger::LogInfo().

Referenced by io::InMemoryStreamFactory< T >::DeleteStorage(), io::FileInputStream< T >::FileInputStream(), io::FileOutputStream< T >::FileOutputStream(), io::FileOutputStream< T >::Finalize(), io::InMemoryOutputStream< T >::Finalize() io::FileOutputStream< T >::FlushBufferInternal(), io::TempFileManager::GenerateTempFilename(), io::InMemoryStreamFactory< T io::TempFileManager::TempFileManager(), io::FileInputStream< T >::~FileInputStream(), and io::TempFileManager::~TempFileMan

Here is the call graph for this function:



Here is the caller graph for this function:



6.4.1.3 LogWarning()

```
void io::detail::LogWarning (
            const std::string & message )  [inline]
```

Log a warning message from io operations.

Parameters

message	The warning message to log

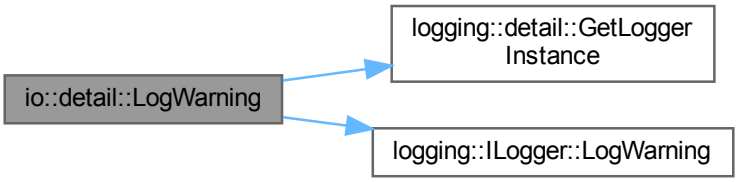
Definition at line 29 of file io_logging.hpp.

```
00029                                                {
00030     auto& logger = logging::detail::GetLoggerInstance();
00031     if (logger) {
00032         logger->LogWarning("[IO] " + message);
00033     }
00034 }
```

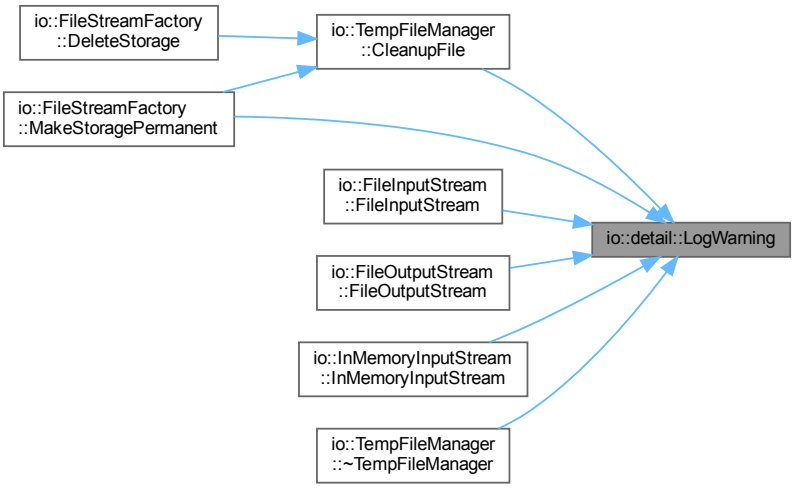
References logging::detail::GetLoggerInstance(), and logging::ILogger::LogWarning().

Referenced by io::TempFileManager::CleanupFile(), io::FileInputStream< T >::FileInputStream(), io::FileOutputStream< T >::FileOu
io::InMemoryInputStream< T >::InMemoryInputStream(), io::FileStreamFactory< T >::MakeStoragePermanent(),
and io::TempFileManager::~TempFileManager().

Here is the call graph for this function:



Here is the caller graph for this function:



6.5 logging Namespace Reference

Namespaces

- namespace detail

Classes

- class ILogger
  Abstract base class defining the logger interface.
- class LoggerAdapter
  Adapter class that wraps a user-defined logger to conform to the ILogger interface.
- class NullLogger
  A logger implementation that discards all log messages.
- class SpdlogWrapper
  Concrete logger implementation wrapping the spdlog library.

Enumerations

- enum class SpdlogSinkType { Console , File , Both }
  Types of logging sinks available for SpdlogWrapper.

Functions

- std::shared_ptr< ILogger > CreateDefaultLogger ()
  Factory function to create a default logger instance.
- template<typename UserLogger >
  void SetLogger (UserLogger &&user_logger)
  Set a custom user-defined logger as the global logger.
- template<typename UserLogger >
  requires (!std::derived_from<UserLogger, ILogger>)
  void SetLogger (std::shared_ptr< UserLogger > user_logger)
  Set a custom user-defined logger passed in std::shared_ptr as the global logger.
- template<typename DerivedLogger >
  requires std::derived_from<DerivedLogger, ILogger>
  void SetLogger (std::shared_ptr< DerivedLogger > logger)
  Set a logger instance derived from ILogger as the global logger.
- void SetLogger (std::shared_ptr< ILogger > logger)
  Set a logger instance that already conforms to the ILogger interface.
- void SetDefaultLogger ()
  Reset the logger to the default NullLogger.

6.5.1 Enumeration Type Documentation

6.5.1.1 SpdlogSinkType

```
enum class logging::SpdlogSinkType  [strong]
```

Types of logging sinks available for SpdlogWrapper.

Enumerator

Console	Log to console (stdout) with colors.
File	Log to file only.
Both	Log to both console and file.

Definition at line 17 of file SpdlogWrapper.hpp.

```
00017                           {
00018     Console,
00019     File,
00020     Both
00021 };
```

6.5.2 Function Documentation

6.5.2.1 CreateDefaultLogger()

```
std::shared_ptr< ILogger > logging::CreateDefaultLogger ( )
```

Factory function to create a default logger instance.

Returns

Shared pointer to a NullLogger (default no-op logger)

Definition at line 12 of file DefaultLogger.cpp.

```
00012                                           {
00013     return std::make_shared<NullLogger>();
00014 }
```

6.5.2.2 SetDefaultLogger()

```
void logging::SetDefaultLogger ( )  [inline]
```

Reset the logger to the default NullLogger.

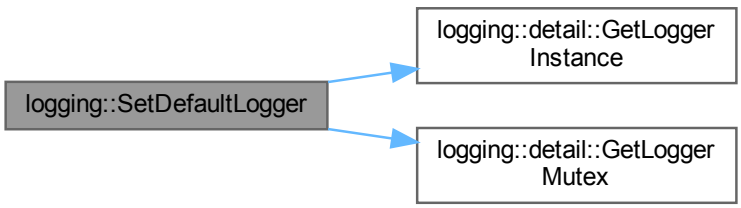
Convenience function that sets a NullLogger instance. Thread-safe: uses a mutex to protect the global logger instance.

Definition at line 131 of file Registry.hpp.

```
00131                                {
00132     std::lock_guard<std::mutex> lock(detail::GetLoggerMutex());
00133     detail::GetLoggerInstance() = std::make_shared<NullLogger>();
00134 }
```

References logging::detail::GetLoggerInstance(), and logging::detail::GetLoggerMutex().

Here is the call graph for this function:



6.5.2.3 SetLogger() [1/4]

```
template<typename DerivedLogger >
requires std::derived_from<DerivedLogger, ILogger>
void logging::SetLogger (
            std::shared_ptr< DerivedLogger > logger )
```

Set a logger instance derived from ILogger as the global logger.

Thread-safe: uses a mutex to protect the global logger instance. This overload is used when passing a shared pointer to a type that derives from ILogger.

Template Parameters

DerivedLogger	A type that derives from ILogger

Parameters

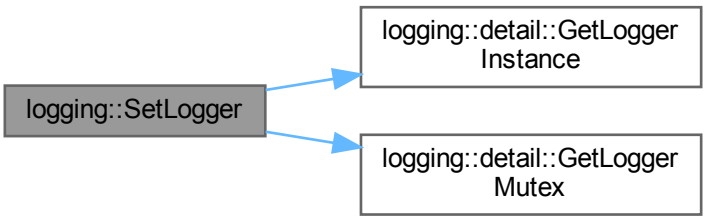
logger	Shared pointer to the derived logger instance to set

Definition at line 107 of file Registry.hpp.

```
00107                                                   {
00108     std::lock_guard<std::mutex> lock(detail::GetLoggerMutex());
00109     detail::GetLoggerInstance() = std::move(logger);
00110 }
```

References logging::detail::GetLoggerInstance(), and logging::detail::GetLoggerMutex().

Here is the call graph for this function:



6.5.2.4 SetLogger() [2/4]

```
void logging::SetLogger (
            std::shared_ptr< ILogger > logger )  [inline]
```

Set a logger instance that already conforms to the ILogger interface.

Thread-safe: uses a mutex to protect the global logger instance. If nullptr is passed, sets a NullLogger instead.

Parameters

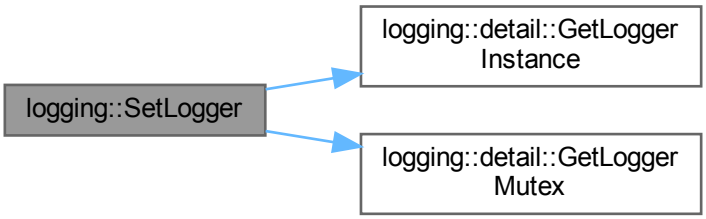
logger	Shared pointer to the logger to set, or nullptr for NullLogger

Definition at line 120 of file Registry.hpp.

```
00120                                                 {
00121     std::lock_guard<std::mutex> lock(detail::GetLoggerMutex());
00122     detail::GetLoggerInstance() = logger ? std::move(logger) : std::make_shared<NullLogger>();
00123 }
```

References logging::detail::GetLoggerInstance(), and logging::detail::GetLoggerMutex().

Here is the call graph for this function:



6.5.2.5 SetLogger() [3/4]

```
template<typename UserLogger >
requires (!std::derived_from<UserLogger, ILogger>)
void logging::SetLogger (
            std::shared_ptr< UserLogger > user_logger )
```

Set a custom user-defined logger passed in std::shared_ptr as the global logger.

Wraps the user logger (held by shared_ptr) in a LoggerAdapter and sets it as the active logger. This overload can be used for non-copyable user loggers. Thread-safe: uses a mutex to protect the global logger instance.

Template Parameters

UserLogger	The type of the user-defined logger (not derived from ILogger)

Parameters

user_logger	Shared pointer to the user logger instance

Example:

```
struct HeavyLogger {
    HeavyLogger() = default;
    HeavyLogger(const HeavyLogger&) = delete;
```

```
    void info(const std::string& msg) { }
    void warn(const std::string& msg) { }
    void error(const std::string& msg) { }
};
auto logger = std::make_shared<HeavyLogger>();
SetLogger(logger);
```

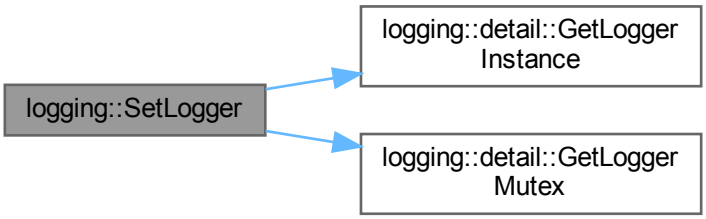
Definition at line 90 of file Registry.hpp.

```
00090                                                 {
00091     auto adapter = std::make_shared<LoggerAdapter<UserLogger»(std::move(user_logger));
00092     std::lock_guard<std::mutex> lock(detail::GetLoggerMutex());
00093     detail::GetLoggerInstance() = std::move(adapter);
00094 }
```

References logging::detail::GetLoggerInstance(), and logging::detail::GetLoggerMutex().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:

6.5.2.6 SetLogger() [4/4]

```
template<typename UserLogger >
void logging::SetLogger (
            UserLogger && user_logger )
```

Set a custom user-defined logger as the global logger.

Wraps the user logger in a LoggerAdapter and sets it as the active logger. The user logger must provide info(), warn(), and error() methods. Thread-safe: uses a mutex to protect the global logger instance.

Template Parameters

UserLogger	The type of the user-defined logger

Parameters

user_logger	The logger instance to set (will be moved or copied)

Example:

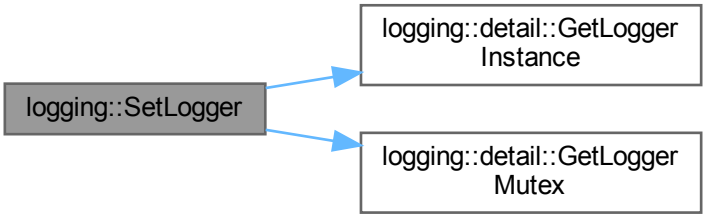
```
struct MyLogger {
    void info(const std::string& msg) { }
    void warn(const std::string& msg) { }
    void error(const std::string& msg) { }
};
SetLogger(MyLogger{});
```

Definition at line 58 of file Registry.hpp.

```
00058                                       {
00059     auto adapter = std::make_shared<LoggerAdapter<std::decay_t<UserLogger>>(
00060         std::forward<UserLogger>(user_logger));
00061     std::lock_guard<std::mutex> lock(detail::GetLoggerMutex());
00062     detail::GetLoggerInstance() = std::move(adapter);
00063 }
```

References logging::detail::GetLoggerInstance(), and logging::detail::GetLoggerMutex().

Here is the call graph for this function:



6.6 logging::detail Namespace Reference

Functions

- std::shared_ptr< ILogger > & GetLoggerInstance ()

  Get the global logger instance.
- std::mutex & GetLoggerMutex ()

  Get the mutex protecting the logger instance.

6.6.1 Function Documentation

6.6.1.1 GetLoggerInstance()

```
std::shared_ptr< ILogger > & logging::detail::GetLoggerInstance ( )  [inline]
```

Get the global logger instance.

Returns

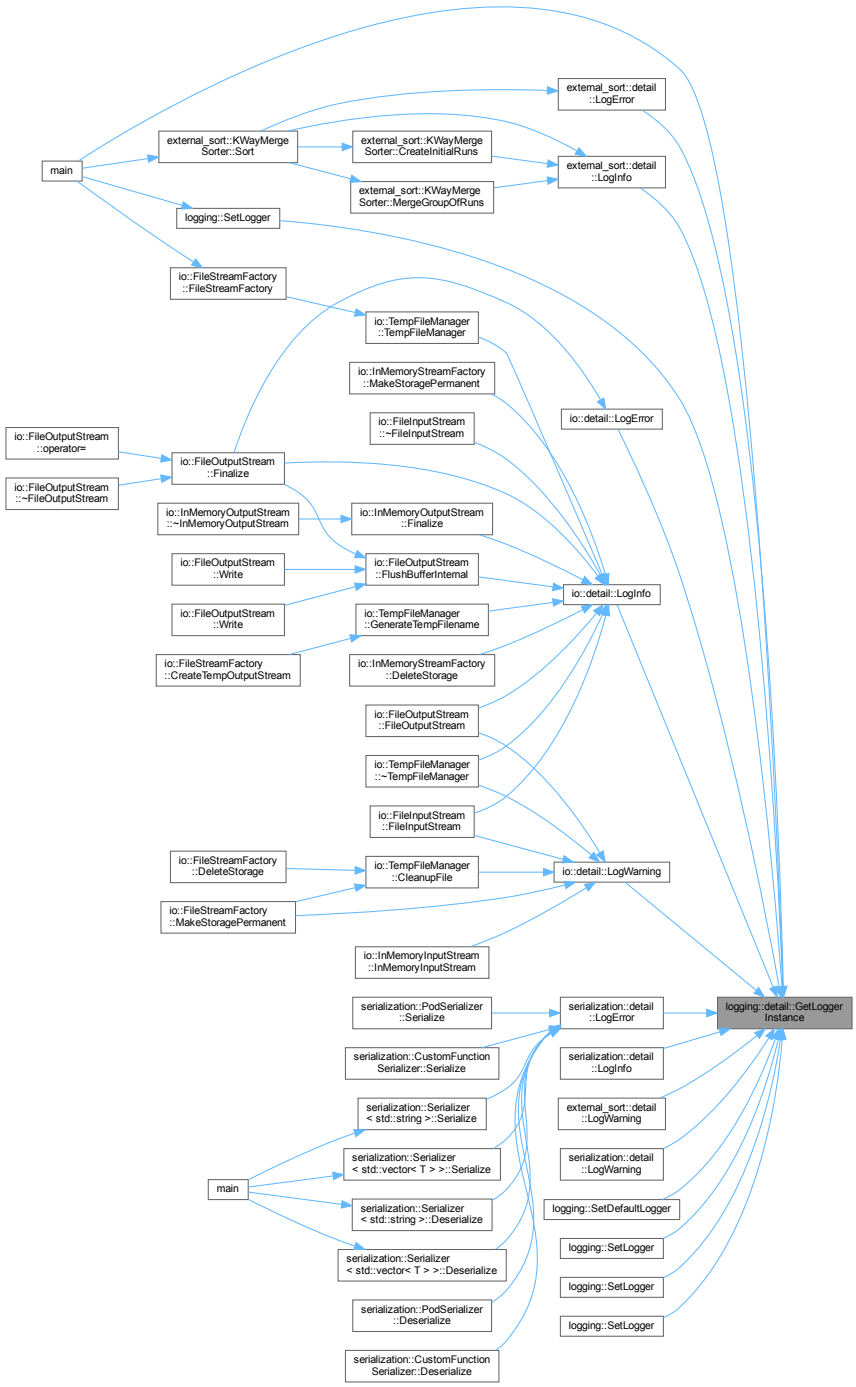
Reference to the shared pointer holding the current logger

Definition at line 22 of file Registry.hpp.

```
00022                                                {
00023     static std::shared_ptr<ILogger> logger = std::make_shared<NullLogger>();
00024     return logger;
00025 }
```

Referenced by external_sort::detail::LogError(), io::detail::LogError(), serialization::detail::LogError(), external_sort::detail::LogInfo(), io::detail::LogInfo(), serialization::detail::LogInfo(), external_sort::detail::LogWarning(), io::detail::LogWarning(), serialization::detail::LogWarning(), main(), logging::SetDefaultLogger(), logging::SetLogger(), logging::SetLogger(), logging::SetLogger(), and logging::SetLogger().

Here is the caller graph for this function:



6.6.1.2 GetLoggerMutex()

```
std::mutex & logging::detail::GetLoggerMutex ( )  [inline]
```

Get the mutex protecting the logger instance.

Returns

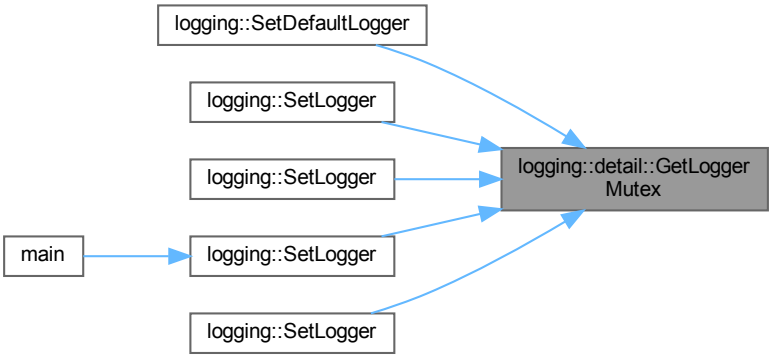
Reference to the global logger mutex

Definition at line 31 of file Registry.hpp.

```
00031                                {
00032     static std::mutex m;
00033     return m;
00034 }
```

Referenced by logging::SetDefaultLogger(), logging::SetLogger(), logging::SetLogger(), logging::SetLogger(), and logging::SetLogger().

Here is the caller graph for this function:



6.7 serialization Namespace Reference

Namespaces

- namespace detail
- namespace detail_adl

  Helpers for ADL - defined in a nested namespace.

Classes

- class CustomFunctionSerializer

  Serializer for types with free function serialization.
- class MethodSerializer

  Serializer for types with method serialization.
- class PodSerializer

  Serializer for POD types.
- class Serializer

  Base interface for serializable types.
- class Serializer< std::string >
- class Serializer< std::vector< T > >

Concepts

- concept PodSerializable

  Concept for POD types that can be serialized through fwrite/fread.
- concept CustomSerializable

  Concept for types with custom serialization through free functions.
- concept MethodSerializable

  Concept for types with serialization methods.
- concept SpecializedSerializable

  Concept for types with Serializer specialization.
- concept FileSerializable

  Concept for types that can be serialized into files.

Functions

- template<typename T >
  std::unique_ptr< Serializer< T > > CreateSerializer ()

  Serializer factory.

6.7.1 Function Documentation

6.7.1.1 CreateSerializer()

```
template<typename T >
std::unique_ptr< Serializer< T > > serialization::CreateSerializer ( )
```

Serializer factory.

Creates appropriate serializer based on type traits

Definition at line 437 of file serializers.hpp.

```
00437                                                {
00438     static_assert(FileSerializable<T>, "Type must be serializable to be used with this library");
00439     if constexpr (PodSerializable<T>) {
00440         return std::make_unique<PodSerializer<T»();
00441     } else if constexpr (CustomSerializable<T>) {
00442         return std::make_unique<CustomFunctionSerializer<T»();
00443     } else if constexpr (MethodSerializable<T>) {
00444         return std::make_unique<MethodSerializer<T»();
00445     } else if constexpr (SpecializedSerializable<T>) {
00446         return std::make_unique<Serializer<T»();
00447     } else {
00448         throw std::logic_error("No serializer found for type " + std::string(typeid(T).name()));
00449     }
00450 }
```

Referenced by main().

Here is the caller graph for this function:

6.8 serialization::detail Namespace Reference

Classes

- class ByteCountingFile

  Helper class to calculate serialized size by writing to a null device.

Functions

- template<typename T , typename SerializeFunc >
  uint64_t CalculateSizeBySerializing (const T &obj, SerializeFunc serialize_func)

  Calculate serialized size by actually serializing to /dev/null.
- void LogError (const std::string &message)

  Log an error message from serialization operations.
- void LogWarning (const std::string &message)

  Log a warning message from serialization operations.
- void LogInfo (const std::string &message)

  Log an informational message from serialization operations.

6.8.1 Function Documentation

6.8.1.1 CalculateSizeBySerializing()

```
template<typename T , typename SerializeFunc >
uint64_t serialization::detail::CalculateSizeBySerializing (
            const T & obj,
            SerializeFunc serialize_func )
```

Calculate serialized size by actually serializing to /dev/null.

This is a fallback mechanism that works for any serializable type. It temporarily serializes the object to count the bytes.

Performance Warning: This function involves actual I/O operations to /dev/null. While /dev/null is optimized by the OS, this is still slower than calculating size arithmetically. Use this only when no faster alternative is available.

Template Parameters

T	Type to serialize
SerializeFunc	Function type that performs serialization

Parameters

obj	Object to calculate size for
serialize_func	Function that performs serialization (signature: bool(const T&, FILE∗))

Returns

Number of bytes that would be written

Exceptions

std::runtime_error	if /dev/null cannot be opened or serialization fails

Definition at line 168 of file serializers.hpp.

```
00168                                                                                {
00169     // Save current position if we're using a real file
00170     FILE* null_file = nullptr;
00171 #ifdef _WIN32
00172     null_file = fopen("NUL", "wb");
00173 #else
00174     null_file = fopen("/dev/null", "wb");
00175 #endif
00176
00177     if (!null_file) {
00178         throw std::runtime_error("Failed to open null device for size calculation");
00179     }
00180
00181     // Get initial position
00182     int64_t start_pos = ftell(null_file);
00183
00184     // Serialize the object
00185     bool result = serialize_func(obj, null_file);
00186
00187     // Get final position
00188     int64_t end_pos = ftell(null_file);
00189     fclose(null_file);
00190
00191     if (!result) {
00192         throw std::runtime_error("Failed to serialize object for size calculation");
00193     }
00194
00195     // Return the number of bytes written
00196     return static_cast<uint64_t>(end_pos - start_pos);
00197 }
```

6.8.1.2 LogError()

```
void serialization::detail::LogError (
            const std::string & message )  [inline]
```

Log an error message from serialization operations.

Parameters

message	The error message to log

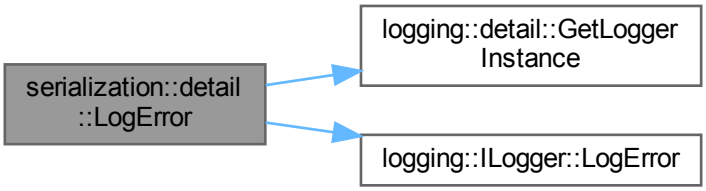
Definition at line 18 of file serialization_logging.hpp.

```
00018                                                  {
00019     auto& logger = logging::detail::GetLoggerInstance();
00020     if (logger) {
00021         logger->LogError("[Serialization] " + message);
00022     }
00023 }
```

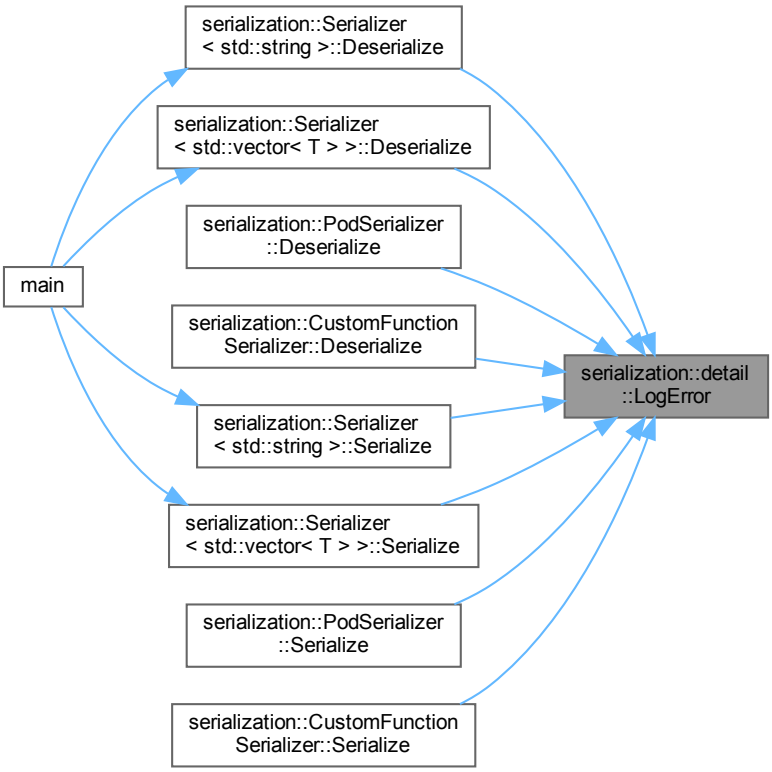
References logging::detail::GetLoggerInstance(), and logging::ILogger::LogError().

Referenced by serialization::Serializer< std::string >::Deserialize(), serialization::Serializer< std::vector< T > >::Deserialize(), serialization::PodSerializer< T >::Deserialize(), serialization::CustomFunctionSerializer< T >::Deserialize(), serialization::Serializer< std::string >::Serialize(), serialization::Serializer< std::vector< T > >::Serialize(), serialization::PodSerializer< T >::Serialize(), and serialization::CustomFunctionSerializer< T >::Serialize().

Here is the call graph for this function:



Here is the caller graph for this function:



6.8.1.3 LogInfo()

```
void serialization::detail::LogInfo (
            const std::string & message )  [inline]
```

Log an informational message from serialization operations.

Parameters

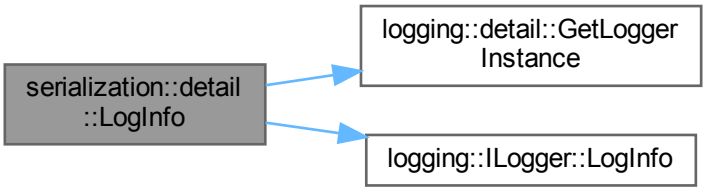
message	The info message to log

Definition at line 40 of file serialization_logging.hpp.

```
00040                                                 {
00041     auto& logger = logging::detail::GetLoggerInstance();
00042     if (logger) {
00043         logger->LogInfo("[Serialization] " + message);
00044     }
00045 }
```

References logging::detail::GetLoggerInstance(), and logging::ILogger::LogInfo().

Here is the call graph for this function:



6.8.1.4 LogWarning()

```
void serialization::detail::LogWarning (
            const std::string & message )  [inline]
```

Log a warning message from serialization operations.

Parameters

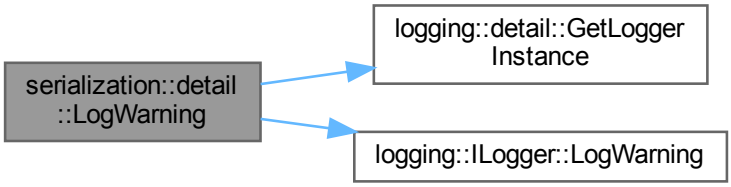
message	The warning message to log

Definition at line 29 of file serialization_logging.hpp.

```
00029                                                    {
00030     auto& logger = logging::detail::GetLoggerInstance();
00031     if (logger) {
00032         logger->LogWarning("[Serialization] " + message);
00033     }
00034 }
```

References logging::detail::GetLoggerInstance(), and logging::ILogger::LogWarning().

Here is the call graph for this function:



6.9 serialization::detail_adl Namespace Reference

Helpers for ADL - defined in a nested namespace.

Functions

- template<typename T >
  auto AdlSerialize (const T &obj, FILE *file) -> decltype(Serialize(obj, file))
- template<typename T >
  auto AdlDeserialize (T &obj, FILE *file) -> decltype(Deserialize(obj, file))

6.9.1 Detailed Description

Helpers for ADL - defined in a nested namespace.

These functions use unqualified lookup which enables ADL. They will find Serialize/Deserialize either:

- In the namespace of the type T (via ADL)
- In the serialization namespace (via normal lookup)

6.9.2 Function Documentation

6.9.2.1 AdlDeserialize()

```
template<typename T >
auto serialization::detail_adl::AdlDeserialize (
            T & obj,
            FILE * file ) -> decltype(Deserialize(obj, file))
```

Definition at line 48 of file type_concepts.hpp.

```
00048                                                                    {
00049     return Deserialize(obj, file);
00050 }
```

6.9.2.2 AdlSerialize()

```
template<typename T >
auto serialization::detail_adl::AdlSerialize (
             const T & obj,
             FILE * file ) -> decltype(Serialize(obj, file))
```

Definition at line 43 of file type_concepts.hpp.

```
00043                                                                    {
00044     return Serialize(obj, file);
00045 }
```

Chapter 7

Concept Documentation

7.1 serialization::CustomSerializable Concept Reference

Concept for types with custom serialization through free functions.

`#include <type_concepts.hpp>`

7.1.1 Concept definition

```
template<typename T>
concept serialization::CustomSerializable =  requires(const T& obj, T& obj_mut, FILE* file) {
    { detail_adl::AdlSerialize(obj, file) } -> std::same_as<bool>;
    { detail_adl::AdlDeserialize(obj_mut, file) } -> std::same_as<bool>;
}
```

7.1.2 Detailed Description

Concept for types with custom serialization through free functions.

Checks that type has free functions Serialize and Deserialize. These functions can be defined either:

1. In the serialization namespace (for std types like std::string, std::vector)
2. In the same namespace as the type for ADL (Argument-Dependent Lookup)

Functions must have the following signatures:

```
bool Serialize(const T& obj, FILE* file);
bool Deserialize(T& obj, FILE* file);
```

- Serialize writes obj into file and returns true on success
- Deserialize reads one object from file into obj and returns true on success

See also

serialization::FileSerializable
serialization::PodSerializable
serialization::MethodSerializable

Definition at line 75 of file type_concepts.hpp.

7.2 serialization::FileSerializable Concept Reference

Concept for types that can be serialized into files.

`#include <type_concepts.hpp>`

7.2.1 Concept definition

```
template<typename T>
concept serialization::FileSerializable =  PodSerializable<T> || CustomSerializable<T> ||
      MethodSerializable<T> || SpecializedSerializable<T>
```

7.2.2 Detailed Description

Concept for types that can be serialized into files.

Types must satisfy one of the following concepts:

- serialization::PodSerializable - POD types with fwrite/fread
- serialization::CustomSerializable - types with free Serialize/Deserialize functions
- serialization::MethodSerializable - types with Serialize/Deserialize methods

See also

serialization::PodSerializable
serialization::CustomSerializable
serialization::MethodSerializable

Definition at line 133 of file type_concepts.hpp.

7.3 serialization::MethodSerializable Concept Reference

Concept for types with serialization methods.

`#include <type_concepts.hpp>`

7.3.1 Concept definition

```
template<typename T>
concept serialization::MethodSerializable =  requires(const T& obj, T& obj_mut, FILE* file) {
    { obj.Serialize(file) } -> std::same_as<bool>;
    { obj_mut.Deserialize(file) } -> std::same_as<bool>;
}
```

7.3.2 Detailed Description

Concept for types with serialization methods.

Checks that type has member methods Serialize and Deserialize.

Methods must have the following signatures:
```
bool Serialize(FILE* file) const;
bool Deserialize(FILE* file);
```

- Serialize writes the object into file and returns true on success
- Deserialize reads one object from file and returns true on success

See also

serialization::FileSerializable

serialization::PodSerializable

serialization::CustomSerializable

Definition at line 99 of file type_concepts.hpp.

7.4 serialization::PodSerializable Concept Reference

Concept for POD types that can be serialized through fwrite/fread.

```
#include <type_concepts.hpp>
```

7.4.1 Concept definition

```
template<typename T>
concept serialization::PodSerializable =  std::is_trivially_copyable_v<T> && std::is_standard_layout_v<T>
```

7.4.2 Detailed Description

Concept for POD types that can be serialized through fwrite/fread.

Checks that type is trivially copyable and has standard layout. Such types can be directly written to/read from files using binary I/O.

Examples: int, double, struct with POD members.

See also

serialization::FileSerializable

serialization::CustomSerializable

serialization::MethodSerializable

Definition at line 31 of file type_concepts.hpp.

7.5 serialization::SpecializedSerializable Concept Reference

Concept for types with Serializer specialization.

`#include <type_concepts.hpp>`

7.5.1 Concept definition

```
template<typename T>
concept serialization::SpecializedSerializable =  Serializer<T>::Specialized::value
```

7.5.2 Detailed Description

Concept for types with Serializer specialization.

Checks that type has a specialization of class Serializer

Specialization must define
`using Specialized = std::true_type;`

See also

serialization::Serializer

serialization::Serializer<std::string>

Definition at line 118 of file type_concepts.hpp.

Chapter 8

Class Documentation

8.1 serialization::detail::ByteCountingFile Class Reference

Helper class to calculate serialized size by writing to a null device.

`#include <serializers.hpp>`

Public Member Functions

- ByteCountingFile ()
- ~ByteCountingFile ()
- FILE * GetFile ()
- void TrackWrite (size_t bytes)
- uint64_t GetBytesWritten () const
- ByteCountingFile (const ByteCountingFile &)=delete
- ByteCountingFile & operator= (const ByteCountingFile &)=delete

Private Attributes

- FILE * file_
- uint64_t bytes_written_ = 0

8.1.1 Detailed Description

Helper class to calculate serialized size by writing to a null device.

This class provides a fallback mechanism to calculate serialized size when the type doesn't provide GetSerialized↩ Size() method. It works by actually serializing the object and counting bytes written.

Definition at line 115 of file serializers.hpp.

8.1.2 Constructor & Destructor Documentation

8.1.2.1 ByteCountingFile() [1/2]

```
serialization::detail::ByteCountingFile::ByteCountingFile ( )  [inline]
```

Definition at line 121 of file serializers.hpp.

```
00121                           {
00122 #ifdef _WIN32
00123         file_ = fopen("NUL", "wb");
00124 #else
00125         file_ = fopen("/dev/null", "wb");
00126 #endif
00127         if (!file_) {
00128             throw std::runtime_error("Failed to open null device for byte counting");
00129         }
00130     }
```

References file_.

8.1.2.2 ~ByteCountingFile()

```
serialization::detail::ByteCountingFile::~ByteCountingFile ( )  [inline]
```

Definition at line 132 of file serializers.hpp.

```
00132                            {
00133         if (file_) {
00134             fclose(file_);
00135         }
00136     }
```

References file_.

8.1.2.3 ByteCountingFile() [2/2]

```
serialization::detail::ByteCountingFile::ByteCountingFile (
            const ByteCountingFile &  )  [delete]
```

8.1.3 Member Function Documentation

8.1.3.1 GetBytesWritten()

```
uint64_t serialization::detail::ByteCountingFile::GetBytesWritten ( ) const  [inline]
```

Definition at line 144 of file serializers.hpp.

```
00144 { return bytes_written_; }
```

References bytes_written_.

8.1.3.2 GetFile()

```
FILE * serialization::detail::ByteCountingFile::GetFile ( )  [inline]
```

Definition at line 138 of file serializers.hpp.

```
00138 { return file_; }
```

References file_.

8.1.3.3 operator=()

```
ByteCountingFile & serialization::detail::ByteCountingFile::operator= (
            const ByteCountingFile &  )  [delete]
```

8.1.3.4 TrackWrite()

```
void serialization::detail::ByteCountingFile::TrackWrite (
            size_t bytes )  [inline]
```

Definition at line 140 of file serializers.hpp.

```
00140                                       {
00141         bytes_written_ += bytes;
00142     }
```

References bytes_written_.

8.1.4 Member Data Documentation

8.1.4.1 bytes_written_

```
uint64_t serialization::detail::ByteCountingFile::bytes_written_ = 0  [private]
```

Definition at line 118 of file serializers.hpp.

Referenced by GetBytesWritten(), and TrackWrite().

8.1.4.2 file_

```
FILE* serialization::detail::ByteCountingFile::file_  [private]
```

Definition at line 117 of file serializers.hpp.

Referenced by ByteCountingFile(), GetFile(), and ~ByteCountingFile().

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

- serialization/include/serializers.hpp

8.2 ConsoleLogger Class Reference

Custom console logger with colored output and message prefixes.

Public Member Functions

- void info (const std::string &msg)
- void warn (const std::string &msg)
- void error (const std::string &msg)

8.2.1 Detailed Description

Custom console logger with colored output and message prefixes.

Definition at line 30 of file example.cpp.

8.2.2 Member Function Documentation

8.2.2.1 error()

```
void ConsoleLogger::error (
            const std::string & msg )  [inline]
```

Definition at line 40 of file example.cpp.

```
00040                                     {  // NOLINT(readability-identifier-naming)
00041         std::cerr « "\033[31m[ERROR]\033[0m " « msg « std::endl;
00042     }
```

8.2.2.2 info()

```
void ConsoleLogger::info (
            const std::string & msg )  [inline]
```

Definition at line 32 of file example.cpp.

```
00032                                    {  // NOLINT(readability-identifier-naming)
00033         std::cout « "\033[32m[INFO]\033[0m " « msg « std::endl;
00034     }
```

8.2.2.3 warn()

```
void ConsoleLogger::warn (
            const std::string & msg )  [inline]
```

Definition at line 36 of file example.cpp.

```
00036                                    {  // NOLINT(readability-identifier-naming)
00037         std::cout « "\033[33m[WARN]\033[0m " « msg « std::endl;
00038     }
```

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

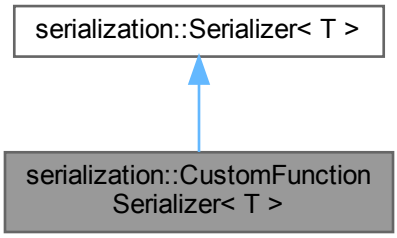
- serialization/examples/example.cpp

8.3 serialization::CustomFunctionSerializer< T > Class Template Reference

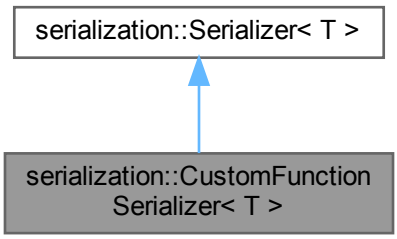
Serializer for types with free function serialization.

`#include <serializers.hpp>`

Inheritance diagram for serialization::CustomFunctionSerializer< T >:



Collaboration diagram for serialization::CustomFunctionSerializer< T >:



Public Member Functions

- bool Serialize (const T &obj, FILE *file) override

  Serialize object into file.
- bool Deserialize (T &obj, FILE *file) override

  Deserialize object from file.
- uint64_t GetSerializedSize (const T &obj) override

  Calculate the serialized size of an object in bytes.

Public Member Functions inherited from serialization::Serializer< T >

- virtual ~Serializer ()=default
- Serializer ()=default
- Serializer (const Serializer &)=default
- Serializer & operator= (const Serializer &)=default
- Serializer (Serializer &&)=default
- Serializer & operator= (Serializer &&)=default

Additional Inherited Members

Public Types inherited from serialization::Serializer< T >

- using Specialized = std::false_type

8.3.1 Detailed Description

template<CustomSerializable T>
class serialization::CustomFunctionSerializer< T >

Serializer for types with free function serialization.

This serializer is used for types that provide Serialize() and Deserialize() as free functions found via ADL (Argument Dependent Lookup).

Size Calculation: GetSerializedSize() performs actual serialization to /dev/null to count bytes. This works reliably but involves I/O overhead.

Performance Note:

> If you need optimal performance for size calculation, consider using MethodSerializable instead (with member functions) and adding a GetSerializedSize() method, or create a specialized Serializer<YourType> with optimized GetSerializedSize().

Template Parameters

T	Type with free Serialize() and Deserialize() functions

See also

> serialization::CustomSerializable

Definition at line 346 of file serializers.hpp.

8.3.2 Member Function Documentation

8.3.2.1 Deserialize()

```
template<CustomSerializable T>
bool serialization::CustomFunctionSerializer< T >::Deserialize (
```

```
            T & obj,
            FILE * file )  [inline], [override], [virtual]
```

Deserialize object from file.

Parameters

obj	Object to deserialize to
file	Pointer to file to read from

Returns

true, if deserialization is successful

Implements serialization::Serializer< T >.

Definition at line 357 of file serializers.hpp.

```
00357                                                {
00358         bool result = detail_adl::AdlDeserialize(obj, file);
00359         result = result && (ferror(file) == 0) && (feof(file) == 0);
00360         if (!result) {
00361             detail::LogError("Custom deserialization failed for type");
00362         }
00363         return result;
00364     }
```

References serialization::detail::LogError().

Here is the call graph for this function:

8.3.2.2 GetSerializedSize()

```
template<CustomSerializable T>
uint64_t serialization::CustomFunctionSerializer< T >::GetSerializedSize (
            const T & obj )  [inline], [override], [virtual]
```

Calculate the serialized size of an object in bytes.

This method computes the exact number of bytes that would be written to a file when serializing the given object. This is useful for:

- Estimating storage requirements before serialization
- Tracking actual data size for complex types (e.g., types with dynamic memory)
- Generating correct amount of data to match target file sizes in benchmarks

Parameters

obj	Object to calculate serialized size for

Returns

Number of bytes that would be written when serializing this object

Note

For POD types, this returns sizeof(T). For complex types (std::string, std::vector, custom types), this computes the actual serialized size including all dynamic data.

Performance Optimization for Custom Types:

For types using MethodSerializer (types with Serialize/Deserialize methods), there are two implementation paths:

1. **Optimized path (RECOMMENDED)**: If your type provides a `GetSerializedSize()` const method, it will be called directly. This is **much faster** as it only calculates the size without actual serialization.
2. **Fallback path**: If `GetSerializedSize()` is not provided, the serializer will perform actual serialization to /dev/null and count bytes. This works but is **significantly slower** (involves I/O operations).

Example (Basic usage):

```
auto serializer = serialization::CreateSerializer<Person>();
Person p{"Alice", 30, 170.5, 65, "123 Main St"};
uint64_t size = serializer->GetSerializedSize(p);
// size includes: sizeof(name length) + name data + age + height + weight +
//                sizeof(address length) + address data
```

Example (Optimized custom type):

```
struct Person {
    std::string name;
    int32_t age;
    std::string address;

    bool Serialize(FILE* file) const { ... }
    bool Deserialize(FILE* file) { ... }

    // Add this method for optimal performance!
    uint64_t GetSerializedSize() const {
        return sizeof(size_t) + name.size()
             + sizeof(int32_t)
             + sizeof(size_t) + address.size();
    }
};
```

Warning

If your type is used in performance-critical code (e.g., benchmarks, large datasets), strongly consider implementing GetSerializedSize() method to avoid I/O overhead.

Implements serialization::Serializer< T >.

Definition at line 366 of file serializers.hpp.

```
00366                                                                {
00367         // For custom serialization, try to calculate size by actually serializing
00368         // This works as a fallback when specialized GetSerializedSize is not available
00369         return detail::CalculateSizeBySerializing(obj, [this](const T& o, FILE* f) {
00370             return detail_adl::AdlSerialize(o, f);
00371         });
00372     }
```

8.3.2.3 Serialize()

```
template<CustomSerializable T>
bool serialization::CustomFunctionSerializer< T >::Serialize (
            const T & obj,
            FILE * file )  [inline], [override], [virtual]
```

Serialize object into file.

Parameters

obj	Object ro serialize
file	Pointer to file to write to

Returns

true, if serialization is successful

Implements serialization::Serializer< T >.

Definition at line 348 of file serializers.hpp.

```
00348                                                   {
00349         bool result = detail_adl::AdlSerialize(obj, file);
00350         result = result && (ferror(file) == 0);
00351         if (!result) {
00352             detail::LogError("Custom serialization failed for type");
00353         }
00354         return result;
00355     }
```

References serialization::detail::LogError().

Here is the call graph for this function:

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

- serialization/include/serializers.hpp

8.4 io::ElementBuffer< T > Class Template Reference

A template class for a buffer of elements of type T.

```
#include <element_buffer.hpp>
```

Public Member Functions

- ElementBuffer (uint64_t capacity)

  Constructs a buffer with a given capacity.
- bool PushBack (const T &element)

  Adds an element to the end of the buffer.
- bool PushBack (T &&element)

  Adds an element to the end of the buffer (move overload)
- const T * Data () const

  Returns a pointer to the buffer's data (read-only)
- uint64_t Size () const

  Returns the number of valid elements in the buffer.
- T * RawDataPtr ()

  Returns a pointer to the raw buffer data (for writing)
- void SetValidElementsCount (uint64_t count)

  Sets the number of valid elements in the buffer.
- T ReadNext ()

  Reads the next element from the buffer and advances the cursor.
- bool HasMoreToRead () const

  Checks if there are more elements to read.
- uint64_t Capacity () const

  Returns the maximum capacity of the buffer.
- bool IsEmpty () const

  Checks if the buffer is empty.
- bool IsFull () const

  Checks if the buffer is full.
- void Clear ()

  Clears the buffer, resetting the count of valid elements and the read cursor.
- ElementBuffer (ElementBuffer &&)=default
- ElementBuffer & operator= (ElementBuffer &&)=default
- ElementBuffer (const ElementBuffer &)=delete
- ElementBuffer & operator= (const ElementBuffer &)=delete
- ~ElementBuffer ()=default

Private Attributes

- std::vector< T > storage_ {}

  Internal storage for elements.
- uint64_t capacity_elements_ {}

  Maximum buffer capacity in elements.
- uint64_t num_valid_elements_ {}

  Current number of valid (filled) elements in the buffer.
- uint64_t read_cursor_ {}

  Cursor position for the next read from the buffer.

8.4.1 Detailed Description

template<typename T>
class io::ElementBuffer< T >

A template class for a buffer of elements of type T.

ElementBuffer provides in-memory buffering for reading or writing a sequence of elements. It manages the capacity, the current number of valid elements, and the read cursor.

Template Parameters

T	The type of elements stored in the buffer

Definition at line 25 of file element_buffer.hpp.

8.4.2 Constructor & Destructor Documentation

8.4.2.1 ElementBuffer() [1/3]

```
template<typename T >
io::ElementBuffer< T >::ElementBuffer (
            uint64_t capacity )  [explicit]
```

Constructs a buffer with a given capacity.

Parameters

capacity	The buffer capacity in elements (minimum 1)

Definition at line 129 of file element_buffer.hpp.

```
00130     : capacity_elements_(std::max(static_cast<uint64_t>(1), capacity)),
00131       num_valid_elements_(0),
00132       read_cursor_(0) {
00133     storage_.resize(capacity_elements_);
00134 }
```

References io::ElementBuffer< T >::capacity_elements_, io::ElementBuffer< T >::num_valid_elements_, io::ElementBuffer< T >::read_cursor_, and io::ElementBuffer< T >::storage_.

8.4.2.2 ElementBuffer() [2/3]

```
template<typename T >
io::ElementBuffer< T >::ElementBuffer (
            ElementBuffer< T > &&  )  [default]
```

8.4.2.3 ElementBuffer() [3/3]

```
template<typename T >
io::ElementBuffer< T >::ElementBuffer (
            const ElementBuffer< T > &  )  [delete]
```

8.4.2.4 ~ElementBuffer()

```
template<typename T >
io::ElementBuffer< T >::~ElementBuffer ( )  [default]
```

8.4.3 Member Function Documentation

8.4.3.1 Capacity()

```
template<typename T >
uint64_t io::ElementBuffer< T >::Capacity ( ) const
```

Returns the maximum capacity of the buffer.

Returns

The buffer capacity in elements

Definition at line 194 of file element_buffer.hpp.

```
00194                                                {
00195     return capacity_elements_;
00196 }
```

References io::ElementBuffer< T >::capacity_elements_.

8.4.3.2 Clear()

```
template<typename T >
void io::ElementBuffer< T >::Clear ( )
```

Clears the buffer, resetting the count of valid elements and the read cursor.

Definition at line 209 of file element_buffer.hpp.

```
00209                                {
00210     num_valid_elements_ = 0;
00211     read_cursor_ = 0;
00212 }
```

References io::ElementBuffer< T >::num_valid_elements_, and io::ElementBuffer< T >::read_cursor_.

8.4.3.3 Data()

```
template<typename T >
const T * io::ElementBuffer< T >::Data ( ) const
```

Returns a pointer to the buffer's data (read-only)

Returns

A const pointer to the beginning of the data

Definition at line 157 of file element_buffer.hpp.

```
00157                                          {
00158     return storage_.data();
00159 }
```

References io::ElementBuffer< T >::storage_.

8.4.3.4 HasMoreToRead()

```
template<typename T >
bool io::ElementBuffer< T >::HasMoreToRead ( ) const
```

Checks if there are more elements to read.

Returns

true if there are more elements to read

Definition at line 189 of file element_buffer.hpp.

```
00189                                           {
00190     return read_cursor_ < num_valid_elements_;
00191 }
```

References io::ElementBuffer< T >::num_valid_elements_, and io::ElementBuffer< T >::read_cursor_.

8.4.3.5 IsEmpty()

```
template<typename T >
bool io::ElementBuffer< T >::IsEmpty ( ) const
```

Checks if the buffer is empty.

Returns

true if there are no valid elements in the buffer

Definition at line 199 of file element_buffer.hpp.

```
00199                                       {
00200     return num_valid_elements_ == 0;
00201 }
```

References io::ElementBuffer< T >::num_valid_elements_.

8.4.3.6 IsFull()

```
template<typename T >
bool io::ElementBuffer< T >::IsFull ( ) const
```

Checks if the buffer is full.

Returns

true if the buffer is filled to its maximum capacity

Definition at line 204 of file element_buffer.hpp.

```
00204                                       {
00205     return num_valid_elements_ == capacity_elements_;
00206 }
```

References io::ElementBuffer< T >::capacity_elements_, and io::ElementBuffer< T >::num_valid_elements_.

8.4.3.7 operator=() [1/2]

```
template<typename T >
ElementBuffer & io::ElementBuffer< T >::operator= (
            const ElementBuffer< T > &  )  [delete]
```

8.4.3.8 operator=() [2/2]

```
template<typename T >
ElementBuffer & io::ElementBuffer< T >::operator= (
            ElementBuffer< T > &&  )  [default]
```

8.4.3.9 PushBack() [1/2]

```
template<typename T >
bool io::ElementBuffer< T >::PushBack (
            const T & element )
```

Adds an element to the end of the buffer.

Parameters

element	The element to add

Returns

true if the buffer becomes full after adding the element

Definition at line 137 of file element_buffer.hpp.

```
00137                                                {
00138     if (num_valid_elements_ < capacity_elements_) {
00139         storage_[num_valid_elements_] = element;
00140         num_valid_elements_++;
00141         return num_valid_elements_ == capacity_elements_;
00142     }
00143     return true;
00144 }
```

References io::ElementBuffer< T >::capacity_elements_, io::ElementBuffer< T >::num_valid_elements_, and io::ElementBuffer< T >::storage_.

8.4.3.10 PushBack() [2/2]

```
template<typename T >
bool io::ElementBuffer< T >::PushBack (
            T && element )
```

Adds an element to the end of the buffer (move overload)

Parameters

element	The element to add (moved)

Returns

true if the buffer becomes full after adding the element

Definition at line 147 of file element_buffer.hpp.

```
00147                                                {
00148     if (num_valid_elements_ < capacity_elements_) {
00149         storage_[num_valid_elements_] = std::move(element);
00150         num_valid_elements_++;
00151         return num_valid_elements_ == capacity_elements_;
00152     }
00153     return true;
00154 }
```

References io::ElementBuffer< T >::capacity_elements_, io::ElementBuffer< T >::num_valid_elements_, and io::ElementBuffer< T >::storage_.

8.4.3.11 RawDataPtr()

```
template<typename T >
T * io::ElementBuffer< T >::RawDataPtr ( )
```

Returns a pointer to the raw buffer data (for writing)

Returns

A pointer to the beginning of the data

Definition at line 167 of file element_buffer.hpp.

```
00167                                  {
00168     return storage_.data();
00169 }
```

References io::ElementBuffer< T >::storage_.

8.4.3.12 ReadNext()

```
template<typename T >
T io::ElementBuffer< T >::ReadNext ( )
```

Reads the next element from the buffer and advances the cursor.

For non-trivial types this function returns the element by value and may use move semantics to avoid copies. Callers that only need the element once should accept the returned value by value to enable moves.

Returns

The next element, or T{} if there are no more elements

Definition at line 181 of file element_buffer.hpp.

```
00181                                  {
00182     if (read_cursor_ < num_valid_elements_) {
00183         return std::move(storage_[read_cursor_++]);
00184     }
00185     return T{};
00186 }
```

References io::ElementBuffer< T >::num_valid_elements_, io::ElementBuffer< T >::read_cursor_, and io::ElementBuffer< T >::sto

8.4.3.13 SetValidElementsCount()

```
template<typename T >
void io::ElementBuffer< T >::SetValidElementsCount (
            uint64_t count )
```

Sets the number of valid elements in the buffer.

Parameters

count	The new number of valid elements

Exceptions

std::length_error	if count exceeds the buffer's capacity

Definition at line 172 of file element_buffer.hpp.

```
00172                                                       {
00173     if (count > capacity_elements_) {
00174         throw std::length_error("ElementBuffer: Count exceeds capacity.");
00175     }
00176     num_valid_elements_ = count;
00177     read_cursor_ = 0;
00178 }
```

References io::ElementBuffer< T >::capacity_elements_, io::ElementBuffer< T >::num_valid_elements_, and io::ElementBuffer< T >::read_cursor_.

8.4.3.14 Size()

```
template<typename T >
uint64_t io::ElementBuffer< T >::Size ( ) const
```

Returns the number of valid elements in the buffer.

Returns

The number of valid elements

Definition at line 162 of file element_buffer.hpp.

```
00162                                   {
00163     return num_valid_elements_;
00164 }
```

References io::ElementBuffer< T >::num_valid_elements_.

8.4.4 Member Data Documentation

8.4.4.1 capacity_elements_

```
template<typename T >
uint64_t io::ElementBuffer< T >::capacity_elements_ {}  [private]
```

Maximum buffer capacity in elements.

Definition at line 28 of file element_buffer.hpp.

```
00028 {};
```

Referenced by io::ElementBuffer< T >::Capacity(), io::ElementBuffer< T >::ElementBuffer(), io::ElementBuffer< T >::IsFull(), io::ElementBuffer< T >::PushBack(), io::ElementBuffer< T >::PushBack(), and io::ElementBuffer< T >::SetValidElementsCount().

8.4.4.2 num_valid_elements_

```
template<typename T >
uint64_t io::ElementBuffer< T >::num_valid_elements_ {}  [private]
```

Current number of valid (filled) elements in the buffer.

Definition at line 29 of file element_buffer.hpp.
```
00029 {};
```

Referenced by io::ElementBuffer< T >::Clear(), io::ElementBuffer< T >::ElementBuffer(), io::ElementBuffer< T >::HasMoreToRead(), io::ElementBuffer< T >::IsEmpty(), io::ElementBuffer< T >::IsFull(), io::ElementBuffer< T >::PushBack(), io::ElementBuffer< T >::PushBack(), io::ElementBuffer< T >::ReadNext(), io::ElementBuffer< T >::SetValidElementsCount(), and io::ElementBuffer< T >::Size().

8.4.4.3 read_cursor_

```
template<typename T >
uint64_t io::ElementBuffer< T >::read_cursor_ {}  [private]
```

Cursor position for the next read from the buffer.

Definition at line 30 of file element_buffer.hpp.
```
00030 {};
```

Referenced by io::ElementBuffer< T >::Clear(), io::ElementBuffer< T >::ElementBuffer(), io::ElementBuffer< T >::HasMoreToRead(), io::ElementBuffer< T >::ReadNext(), and io::ElementBuffer< T >::SetValidElementsCount().

8.4.4.4 storage_

```
template<typename T >
std::vector<T> io::ElementBuffer< T >::storage_ {}  [private]
```

Internal storage for elements.

Definition at line 27 of file element_buffer.hpp.
```
00027 {};
```

Referenced by io::ElementBuffer< T >::Data(), io::ElementBuffer< T >::ElementBuffer(), io::ElementBuffer< T >::PushBack(), io::ElementBuffer< T >::PushBack(), io::ElementBuffer< T >::RawDataPtr(), and io::ElementBuffer< T >::ReadNext().

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

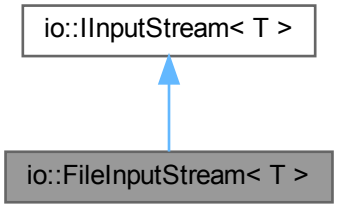
- io/include/element_buffer.hpp

8.5 io::FileInputStream< T > Class Template Reference

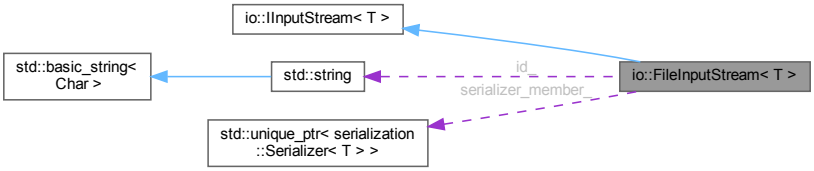
File implementation of an input stream.

`#include <file_stream.hpp>`

Inheritance diagram for io::FileInputStream< T >:



Collaboration diagram for io::FileInputStream< T >:



Public Member Functions

- FileInputStream (const StorageId &filename, uint64_t buffer_capacity_elements)

  Constructor for the file input stream.
- ~FileInputStream () override

  Destructor, closes the file.
- FileInputStream (const FileInputStream &)=delete
- FileInputStream & operator= (const FileInputStream &)=delete
- FileInputStream (FileInputStream &&other) noexcept
- FileInputStream & operator= (FileInputStream &&other) noexcept
- void Advance () override

  Advances the stream to the next element.
- const T & Value () const override

  Returns the current element of the stream.
- bool IsExhausted () const override

  Checks if the stream is exhausted.
- bool IsEmptyOriginalStorage () const override

  Checks if the original storage was empty.
- T TakeValue ()

  Transfers the ownership of the current value and resets the state.

Public Member Functions inherited from io::IInputStream< T >

- virtual ~IInputStream ()=default

Private Member Functions

- void FillBuffer ()

  Fills the internal buffer with data from the file.

Private Attributes

- StorageId id_

  Storage identifier (file name)
- FILE * file_ptr_ = nullptr

  Pointer to the file.
- ElementBuffer< T > buffer_

  Buffer for reading.
- uint64_t total_elements_in_file_ = 0

  Total number of elements in the file.
- uint64_t total_elements_read_ = 0

  Number of elements read so far.
- bool is_exhausted_ = false

  Flag indicating if the stream is exhausted.
- T current_value_ {}

  The current element.
- bool has_valid_value_ = false

  Flag indicating if the current element is valid.
- std::unique_ptr< serialization::Serializer< T > > serializer_member_ {}

  Serializer for the elements.

8.5.1 Detailed Description

template<typename T>
class io::FileInputStream< T >

File implementation of an input stream.

Implements the IInputStream interface for reading data from a file. Uses buffering for efficient reading.

Note: This implementation provides efficient access for non-trivial types by supporting TakeValue() (move out) and minimizing copies when possible.

Template Parameters

T	The type of elements in the stream

Definition at line 39 of file file_stream.hpp.

8.5.2 Constructor & Destructor Documentation

8.5.2.1 FileInputStream() [1/3]

```
template<typename T >
io::FileInputStream< T >::FileInputStream (
            const StorageId & filename,
            uint64_t buffer_capacity_elements )
```

Constructor for the file input stream.

Parameters

filename	The name of the file to read from
buffer_capacity_elements	Buffer capacity in elements

Exceptions

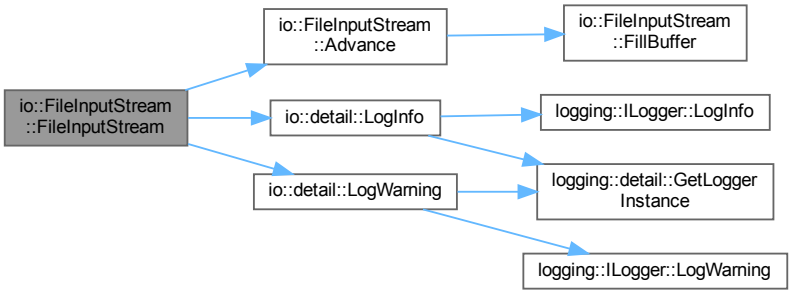
std::runtime_error	if the file cannot be opened

Definition at line 319 of file file_stream.hpp.

```
00320     : id_(filename), buffer_(buffer_capacity_elements) {
00321     if constexpr (!serialization::PodSerializable<T>) {
00322         serializer_member_ = serialization::CreateSerializer<T>();
00323     }
00324     file_ptr_ = fopen(id_.c_str(), "rb");
00325     if (!file_ptr_) {
00326         throw std::runtime_error("FileInputStream: Cannot open input file: " + id_);
00327     }
00328     if (setvbuf(file_ptr_, nullptr, _IONBF, 0) != 0) {
00329         detail::LogWarning("FileInputStream could not disable stdio buffering for " + id_);
00330     }
00331
00332     if (fread(&total_elements_in_file_, sizeof(uint64_t), 1, file_ptr_) != 1) {
00333         if (feof(file_ptr_) && std::filesystem::file_size(id_) < sizeof(uint64_t)) {
00334             total_elements_in_file_ = 0;
00335         } else {
00336             fclose(file_ptr_);
00337             throw std::runtime_error("FileInputStream: Cannot read size header from file: " + id_);
00338         }
00339     }
00340     detail::LogInfo("FileInputStream: " + id_ +
00341                     " opened. Header elements: " + std::to_string(total_elements_in_file_));
00342
00343     if (total_elements_in_file_ == 0) {
00344         is_exhausted_ = true;
00345         has_valid_value_ = false;
00346     } else {
00347         Advance();
00348     }
00349 }
```

References io::FileInputStream< T >::Advance(), io::FileInputStream< T >::buffer_, io::FileInputStream< T >::file_ptr_, io::FileInputStream< T >::has_valid_value_, io::FileInputStream< T >::id_, io::FileInputStream< T >::is_exhausted_, io::detail::LogInfo(), io::detail::LogWarning(), io::FileInputStream< T >::serializer_member_, and io::FileInputStream< T >::total_eler

Here is the call graph for this function:



8.5.2.2 ~FileInputStream()

```
template<typename T >
io::FileInputStream< T >::~FileInputStream ( )  [override]
```

Destructor, closes the file.

Definition at line 352 of file file_stream.hpp.

```
00352                                       {
00353     if (file_ptr_) {
00354         fclose(file_ptr_);
00355         detail::LogInfo("FileInputStream: Closed " + id_);
00356     }
00357 }
```

References io::FileInputStream< T >::file_ptr_, io::FileInputStream< T >::id_, and io::detail::LogInfo().

Here is the call graph for this function:

8.5.2.3 FileInputStream() [2/3]

```
template<typename T >
io::FileInputStream< T >::FileInputStream (
            const FileInputStream< T > &  )  [delete]
```

8.5.2.4 FileInputStream() [3/3]

```
template<typename T >
io::FileInputStream< T >::FileInputStream (
            FileInputStream< T > && other )  [noexcept]
```

Definition at line 360 of file file_stream.hpp.

```
00361     : id_(std::move(other.id_)),
00362       file_ptr_(other.file_ptr_),
00363       buffer_(std::move(other.buffer_)),
00364       total_elements_in_file_(other.total_elements_in_file_),
00365       total_elements_read_(other.total_elements_read_),
00366       is_exhausted_(other.is_exhausted_),
00367       current_value_(std::move(other.current_value_)),
00368       has_valid_value_(other.has_valid_value_),
00369       serializer_member_(std::move(other.serializer_member_)) {
00370     other.file_ptr_ = nullptr;
00371 }
```

References io::FileInputStream< T >::buffer_, io::FileInputStream< T >::current_value_, io::FileInputStream< T >::file_ptr_, io::FileInputStream< T >::has_valid_value_, io::FileInputStream< T >::id_, io::FileInputStream< T >::is_exhausted_, io::FileInputStream< T >::serializer_member_, io::FileInputStream< T >::total_elements_in_file_, and io::FileInputStream< T >::tot

8.5.3 Member Function Documentation

8.5.3.1 Advance()

```
template<typename T >
void io::FileInputStream< T >::Advance ( )  [override], [virtual]
```

Advances the stream to the next element.

Implements io::IInputStream< T >.

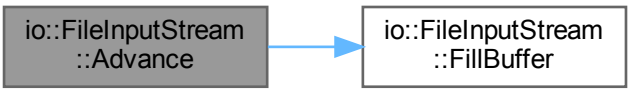
Definition at line 394 of file file_stream.hpp.

```
00394                                {
00395     if (is_exhausted_ ||
00396         (total_elements_in_file_ > 0 && total_elements_read_ >= total_elements_in_file_)) {
00397         has_valid_value_ = false;
00398         is_exhausted_ = true;
00399         return;
00400     }
00401     if (!buffer_.HasMoreToRead()) {
00402         FillBuffer();
00403         if (!buffer_.HasMoreToRead()) {
00404             has_valid_value_ = false;
00405             is_exhausted_ = true;
00406             return;
00407         }
00408     }
00409     current_value_ = buffer_.ReadNext();
00410     total_elements_read_++;
00411     has_valid_value_ = true;
00412     if (total_elements_read_ >= total_elements_in_file_) {
00413         is_exhausted_ = true;
00414     }
00415 }
```

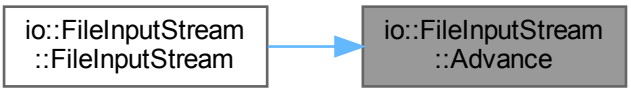
References io::FileInputStream< T >::buffer_, io::FileInputStream< T >::current_value_, io::FileInputStream< T >::FillBuffer(), io::FileInputStream< T >::has_valid_value_, io::FileInputStream< T >::is_exhausted_, io::FileInputStream< T >::total_elements_in_ and io::FileInputStream< T >::total_elements_read_.

Referenced by io::FileInputStream< T >::FileInputStream().

Here is the call graph for this function:



Here is the caller graph for this function:



8.5.3.2 FillBuffer()

```
template<typename T >
void io::FileInputStream< T >::FillBuffer ( )  [private]
```

Fills the internal buffer with data from the file.

Definition at line 267 of file file_stream.hpp.

```
00267                                    {
00268     if (is_exhausted_ ||
00269         (total_elements_in_file_ > 0 && total_elements_read_ >= total_elements_in_file_)) {
00270         buffer_.SetValidElementsCount(0);
00271         is_exhausted_ = true;
00272         has_valid_value_ = false;
00273         return;
00274     }
00275     uint64_t elements_to_read_this_pass =
00276         std::min(buffer_.Capacity(), total_elements_in_file_ - total_elements_read_);
00277
00278     if (elements_to_read_this_pass == 0) {
00279         buffer_.SetValidElementsCount(0);
00280         is_exhausted_ = true;
00281         has_valid_value_ = false;
00282         return;
00283     }
00284
00285     buffer_.Clear();
00286
00287     uint64_t elements_actually_read = 0;
00288     if constexpr (serialization::PodSerializable<T>) {
00289         elements_actually_read =
00290             fread(buffer_.RawDataPtr(), sizeof(T), elements_to_read_this_pass, file_ptr_);
00291         buffer_.SetValidElementsCount(elements_actually_read);
00292     } else {
00293         for (uint64_t i = 0; i < elements_to_read_this_pass; ++i) {
00294             T element;
00295             if (!serializer_member_->Deserialize(element, file_ptr_)) {
00296                 break;
00297             }
```

```
00298             buffer_.PushBack(element);
00299             elements_actually_read++;
00300         }
00301     }
00302
00303     if (elements_actually_read < elements_to_read_this_pass && !feof(file_ptr_)) {
00304         if (ferror(file_ptr_)) {
00305             throw std::runtime_error("FileInputStream: Error reading from file: " + id_ +
00306                                      " (ferror set)");
00307         }
00308         throw std::runtime_error("FileInputStream: Failed to read expected elements from file: " +
00309                                  id_);
00310     }
00311     if (elements_actually_read == 0 && total_elements_read_ < total_elements_in_file_ &&
00312         feof(file_ptr_)) {
00313         is_exhausted_ = true;
00314         has_valid_value_ = false;
00315     }
00316 }
```

References io::FileInputStream< T >::buffer_, io::FileInputStream< T >::file_ptr_, io::FileInputStream< T >::has_valid_value_, io::FileInputStream< T >::id_, io::FileInputStream< T >::is_exhausted_, io::FileInputStream< T >::serializer_member_, io::FileInputStream< T >::total_elements_in_file_, and io::FileInputStream< T >::total_elements_read_.

Referenced by io::FileInputStream< T >::Advance().

Here is the caller graph for this function:

8.5.3.3 IsEmptyOriginalStorage()

```
template<typename T >
bool io::FileInputStream< T >::IsEmptyOriginalStorage ( ) const  [override], [virtual]
```

Checks if the original storage was empty.

Returns

true if the file contained no elements

Implements io::IInputStream< T >.

Definition at line 441 of file file_stream.hpp.

```
00441                                                 {
00442     return total_elements_in_file_ == 0;
00443 }
```

References io::FileInputStream< T >::total_elements_in_file_.

8.5.3.4 IsExhausted()

```
template<typename T >
bool io::FileInputStream< T >::IsExhausted ( ) const  [override], [virtual]
```

Checks if the stream is exhausted.

Returns

true if there are no more elements to read

Implements io::IInputStream< T >.

Definition at line 436 of file file_stream.hpp.

```
00436                                          {
00437     return is_exhausted_ && !has_valid_value_;
00438 }
```

References io::FileInputStream< T >::has_valid_value_, and io::FileInputStream< T >::is_exhausted_.

8.5.3.5 operator=() [1/2]

```
template<typename T >
FileInputStream & io::FileInputStream< T >::operator= (
            const FileInputStream< T > &  )  [delete]
```

8.5.3.6 operator=() [2/2]

```
template<typename T >
FileInputStream< T > & io::FileInputStream< T >::operator= (
            FileInputStream< T > && other )  [noexcept]
```

Definition at line 374 of file file_stream.hpp.

```
00374                                                                                {
00375     if (this != &other) {
00376         if (file_ptr_) {
00377             fclose(file_ptr_);
00378         }
00379         id_ = std::move(other.id_);
00380         file_ptr_ = other.file_ptr_;
00381         buffer_ = std::move(other.buffer_);
00382         total_elements_in_file_ = other.total_elements_in_file_;
00383         total_elements_read_ = other.total_elements_read_;
00384         is_exhausted_ = other.is_exhausted_;
00385         current_value_ = std::move(other.current_value_);
00386         has_valid_value_ = other.has_valid_value_;
00387         serializer_member_ = std::move(other.serializer_member_);
00388         other.file_ptr_ = nullptr;
00389     }
00390     return *this;
00391 }
```

References io::FileInputStream< T >::buffer_, io::FileInputStream< T >::current_value_, io::FileInputStream< T >::file_ptr_, io::FileInputStream< T >::has_valid_value_, io::FileInputStream< T >::id_, io::FileInputStream< T >::is_exhausted_, io::FileInputStream< T >::serializer_member_, io::FileInputStream< T >::total_elements_in_file_, and io::FileInputStream< T >::tot

8.5.3.7 TakeValue()

```
template<typename T >
T io::FileInputStream< T >::TakeValue ( )  [virtual]
```

Transfers the ownership of the current value and resets the state.

Returns

The current value

Exceptions

std::logic_error	if the stream is exhausted

Implements io::IInputStream< T >.

Definition at line 426 of file file_stream.hpp.

```
00426                                {
00427     if (!has_valid_value_) {
00428         throw std::logic_error("TakeValue from exhausted FileInputStream: " + id_);
00429     }
00430     T tmp = std::move(current_value_);
00431     has_valid_value_ = false;
00432     return tmp;
00433 }
```

References io::FileInputStream< T >::current_value_, io::FileInputStream< T >::has_valid_value_, and io::FileInputStream< T >::id

8.5.3.8 Value()

```
template<typename T >
const T & io::FileInputStream< T >::Value ( ) const  [override], [virtual]
```

Returns the current element of the stream.

Returns

A constant reference to the current element

Exceptions

std::logic_error	if the stream is exhausted

Implements io::IInputStream< T >.

Definition at line 418 of file file_stream.hpp.

```
00418                                      {
00419     if (!has_valid_value_) {
00420         throw std::logic_error("Value from exhausted FileInputStream: " + id_);
00421     }
00422     return current_value_;
00423 }
```

References io::FileInputStream< T >::current_value_, io::FileInputStream< T >::has_valid_value_, and io::FileInputStream< T >::id

8.5.4 Member Data Documentation

8.5.4.1 buffer_

```
template<typename T >
ElementBuffer<T> io::FileInputStream< T >::buffer_  [private]
```

Buffer for reading.

Definition at line 43 of file file_stream.hpp.

Referenced by io::FileInputStream< T >::Advance(), io::FileInputStream< T >::FileInputStream(), io::FileInp
io::FileInputStream< T >::FillBuffer(), and io::FileInputStream< T >::operator=().

8.5.4.2 current_value_

```
template<typename T >
T io::FileInputStream< T >::current_value_ {}  [private]
```

The current element.

Definition at line 47 of file file_stream.hpp.
```
00047 {};
```

Referenced by io::FileInputStream< T >::Advance(), io::FileInputStream< T >::FileInputStream(), io::FileInputStream< T >::operato
io::FileInputStream< T >::TakeValue(), and io::FileInputStream< T >::Value().

8.5.4.3 file_ptr_

```
template<typename T >
FILE* io::FileInputStream< T >::file_ptr_ = nullptr  [private]
```

Pointer to the file.

Definition at line 42 of file file_stream.hpp.

Referenced by io::FileInputStream< T >::FileInputStream(), io::FileInputStream< T >::FileInputStream(),
io::FileInputStream< T >::FillBuffer(), io::FileInputStream< T >::operator=(), and io::FileInputStream< T >::~FileInputStream().

8.5.4.4 has_valid_value_

```
template<typename T >
bool io::FileInputStream< T >::has_valid_value_ = false  [private]
```

Flag indicating if the current element is valid.

Definition at line 48 of file file_stream.hpp.

Referenced by io::FileInputStream< T >::Advance(), io::FileInputStream< T >::FileInputStream(), io::FileInputStream< T >::FileInp
io::FileInputStream< T >::FillBuffer(), io::FileInputStream< T >::IsExhausted(), io::FileInputStream< T >::operator=(),
io::FileInputStream< T >::TakeValue(), and io::FileInputStream< T >::Value().

8.5.4.5 id_

```
template<typename T >
StorageId io::FileInputStream< T >::id_  [private]
```

Storage identifier (file name)

Definition at line 41 of file file_stream.hpp.

Referenced by io::FileInputStream< T >::FileInputStream(), io::FileInputStream< T >::FileInputStream(),
io::FileInputStream< T >::FillBuffer(), io::FileInputStream< T >::operator=(), io::FileInputStream< T >::TakeValue(),
io::FileInputStream< T >::Value(), and io::FileInputStream< T >::~FileInputStream().

8.5.4.6 is_exhausted_

```
template<typename T >
bool io::FileInputStream< T >::is_exhausted_ = false  [private]
```

Flag indicating if the stream is exhausted.

Definition at line 46 of file file_stream.hpp.

Referenced by io::FileInputStream< T >::Advance(), io::FileInputStream< T >::FileInputStream(), io::FileInputStream< T >::FileInp
io::FileInputStream< T >::FillBuffer(), io::FileInputStream< T >::IsExhausted(), and io::FileInputStream< T >::operator=().

8.5.4.7 serializer_member_

```
template<typename T >
std::unique_ptr<serialization::Serializer<T> > io::FileInputStream< T >::serializer_member↩
_ {}  [private]
```

Serializer for the elements.

Definition at line 50 of file file_stream.hpp.
00050 {};

Referenced by io::FileInputStream< T >::FileInputStream(), io::FileInputStream< T >::FileInputStream(),
io::FileInputStream< T >::FillBuffer(), and io::FileInputStream< T >::operator=().

8.5.4.8 total_elements_in_file_

```
template<typename T >
uint64_t io::FileInputStream< T >::total_elements_in_file_ = 0  [private]
```

Total number of elements in the file.

Definition at line 44 of file file_stream.hpp.

Referenced by io::FileInputStream< T >::Advance(), io::FileInputStream< T >::FileInputStream(), io::FileInp
io::FileInputStream< T >::FillBuffer(), io::FileInputStream< T >::IsEmptyOriginalStorage(), and io::FileInputStream< T >::operator=

8.5.4.9 total_elements_read_

```
template<typename T >
uint64_t io::FileInputStream< T >::total_elements_read_ = 0  [private]
```

Number of elements read so far.

Definition at line 45 of file file_stream.hpp.

Referenced by io::FileInputStream< T >::Advance(), io::FileInputStream< T >::FileInputStream(), io::FillBuff
and io::FileInputStream< T >::operator=().

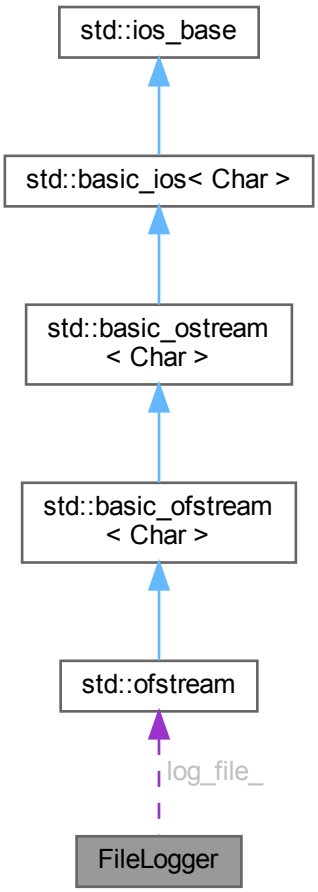
The documentation for this class was generated from the following file:

- io/include/file_stream.hpp

8.6 FileLogger Class Reference

Non-copyable file logger that writes to a log file.

Collaboration diagram for FileLogger:



Public Member Functions

- FileLogger (const std::string &filename)
- ~FileLogger ()
- FileLogger (const FileLogger &)=delete
- FileLogger & operator= (const FileLogger &)=delete
- FileLogger (FileLogger &&)=default
- FileLogger & operator= (FileLogger &&)=default
- void info (const std::string &msg)
- void warn (const std::string &msg)
- void error (const std::string &msg)

Private Attributes

- std::ofstream log_file_

8.6.1 Detailed Description

Non-copyable file logger that writes to a log file.

This logger uses RAII to manage a file handle and is explicitly non-copyable to demonstrate that LoggerAdapter can handle such types via shared_ptr.

Definition at line 51 of file example.cpp.

8.6.2 Constructor & Destructor Documentation

8.6.2.1 FileLogger() [1/3]

```
FileLogger::FileLogger (
            const std::string & filename )  [inline], [explicit]
```

Definition at line 53 of file example.cpp.

```
00053                                                   : log_file_(filename, std::ios::app) {
00054         if (log_file_.is_open()) {
00055             log_file_ « "\n=== New logging session started ===\n";
00056         }
00057     }
```

References log_file_.

8.6.2.2 ~FileLogger()

```
FileLogger::~FileLogger ( )  [inline]
```

Definition at line 59 of file example.cpp.

```
00059                    {
00060         if (log_file_.is_open()) {
00061             log_file_ « "=== Logging session ended ===\n\n";
00062         }
00063     }
```

References log_file_.

8.6.2.3 FileLogger() [2/3]

```
FileLogger::FileLogger (
            const FileLogger &  )  [delete]
```

8.6.2.4 FileLogger() [3/3]

```
FileLogger::FileLogger (
            FileLogger &&  )  [default]
```

8.6.3 Member Function Documentation

8.6.3.1 error()

```
void FileLogger::error (
            const std::string & msg )  [inline]
```

Definition at line 82 of file example.cpp.

```
00082                                     {  // NOLINT(readability-identifier-naming)
00083         if (log_file_.is_open()) {
00084             log_file_ « "[ERROR] " « msg « std::endl;
00085         }
00086     }
```

References log_file_.

8.6.3.2 info()

```
void FileLogger::info (
            const std::string & msg )  [inline]
```

Definition at line 70 of file example.cpp.

```
00070                                    {  // NOLINT(readability-identifier-naming)
00071         if (log_file_.is_open()) {
00072             log_file_ « "[INFO] " « msg « std::endl;
00073         }
00074     }
```

References log_file_.

8.6.3.3 operator=() [1/2]

```
FileLogger & FileLogger::operator= (
            const FileLogger &  )  [delete]
```

8.6.3.4 operator=() [2/2]

```
FileLogger & FileLogger::operator= (
            FileLogger &&  )  [default]
```

8.6.3.5 warn()

```
void FileLogger::warn (
            const std::string & msg )  [inline]
```

Definition at line 76 of file example.cpp.

```
00076                                     {  // NOLINT(readability-identifier-naming)
00077         if (log_file_.is_open()) {
00078             log_file_ « "[WARN] " « msg « std::endl;
00079         }
00080     }
```

References log_file_.

8.6.4 Member Data Documentation

8.6.4.1 log_file_

```
std::ofstream FileLogger::log_file_  [private]
```

Definition at line 89 of file example.cpp.

Referenced by error(), FileLogger(), info(), warn(), and ~FileLogger().

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

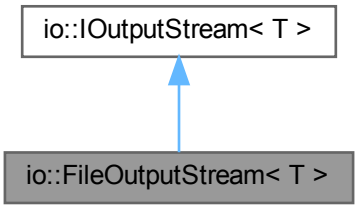
- serialization/examples/example.cpp

8.7 io::FileOutputStream< T > Class Template Reference

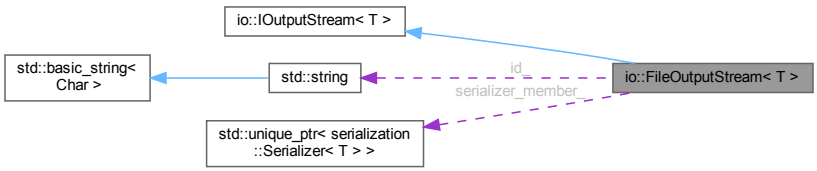
File implementation of an output stream.

```
#include <file_stream.hpp>
```

Inheritance diagram for io::FileOutputStream< T >:



Collaboration diagram for io::FileOutputStream< T >:



Public Member Functions

- FileOutputStream (const StorageId &filename, uint64_t buffer_capacity_elements)

  Constructor for the file output stream.
- ~FileOutputStream () override

  Destructor, finalizes the stream.
- FileOutputStream (const FileOutputStream &)=delete
- FileOutputStream & operator= (const FileOutputStream &)=delete
- FileOutputStream (FileOutputStream &&other) noexcept
- FileOutputStream & operator= (FileOutputStream &&other) noexcept
- void Write (const T &value) override

  Writes an element to the stream.
- void Write (T &&value) override

  Writes an rvalue element to the stream.
- void Finalize () override

  Finalizes the stream, writing all buffered data to the file.
- uint64_t GetTotalElementsWritten () const override

  Returns the total number of elements written.
- uint64_t GetTotalBytesWritten () const override

  Returns the total number of bytes written to the file.
- StorageId GetId () const override

  Returns the file identifier.

Public Member Functions inherited from io::IOutputStream< T >

- virtual ~IOutputStream ()=default

Private Member Functions

- void FlushBufferInternal ()

  Flushes the buffer to the file.

Private Attributes

- StorageId id_

  Storage identifier (file name)
- FILE * file_ptr_ = nullptr

  Pointer to the file.
- ElementBuffer< T > buffer_

  Buffer for writing.
- uint64_t total_elements_written_ = 0

  Number of elements written.
- uint64_t total_bytes_written_ = 0

  Total bytes written (data + header)
- bool finalized_ = false

  Flag indicating if the stream has been finalized.
- std::unique_ptr< serialization::Serializer< T > > serializer_member_ {}

  Serializer for the elements.

8.7.1 Detailed Description

template<typename T>
class io::FileOutputStream< T >

File implementation of an output stream.

Implements the IOutputStream interface for writing data to a file. Uses buffering for efficient writing.

Template Parameters

T	The type of elements in the stream

Definition at line 118 of file file_stream.hpp.

8.7.2 Constructor & Destructor Documentation

8.7.2.1 FileOutputStream() [1/3]

```
template<typename T >
io::FileOutputStream< T >::FileOutputStream (
            const StorageId & filename,
            uint64_t buffer_capacity_elements )
```

Constructor for the file output stream.

Parameters

filename	The name of the file to write to
buffer_capacity_elements	Buffer capacity in elements

Exceptions

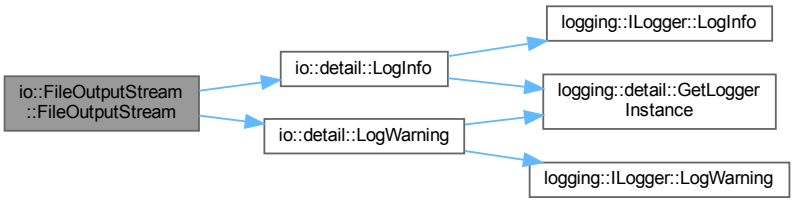
std::runtime_error	if the file cannot be created

Definition at line 478 of file file_stream.hpp.

```
00479     : id_(filename), buffer_(buffer_capacity_elements) {
00480     if constexpr (!serialization::PodSerializable<T>) {
00481         serializer_member_ = serialization::CreateSerializer<T>();
00482     }
00483     file_ptr_ = fopen(id_.c_str(), "wb");
00484     if (!file_ptr_) {
00485         throw std::runtime_error("FileOutputStream: Cannot open output file: " + id_);
00486     }
00487     if (setvbuf(file_ptr_, nullptr, _IONBF, 0) != 0) {
00488         detail::LogWarning("FileOutputStream could not disable stdio buffering for " + id_);
00489     }
00490     uint64_t placeholder_size = 0;
00491     if (fwrite(&placeholder_size, sizeof(uint64_t), 1, file_ptr_) != 1) {
00492         fclose(file_ptr_);
00493         throw std::runtime_error("FileOutputStream: Failed to write placeholder size to " + id_);
00494     }
00495     // Account for the header size in total bytes written
00496     total_bytes_written_ = sizeof(uint64_t);
00497     detail::LogInfo("FileOutputStream: " + id_ + " opened for writing.");
00498 }
```

References io::FileOutputStream< T >::buffer_, io::FileOutputStream< T >::file_ptr_, io::FileOutputStream< T >::id_, io::detail::LogInfo(), io::detail::LogWarning(), io::FileOutputStream< T >::serializer_member_, and io::FileOutputStream< T >::total_

Here is the call graph for this function:



8.7.2.2 ~FileOutputStream()

```
template<typename T >
io::FileOutputStream< T >::~FileOutputStream ( )  [override]
```

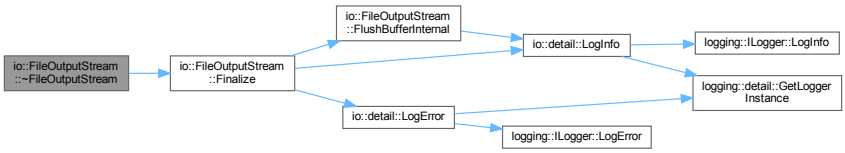
Destructor, finalizes the stream.

Definition at line 501 of file file_stream.hpp.

```
00501                                       {
00502     Finalize();
00503 }
```

References io::FileOutputStream< T >::Finalize().

Here is the call graph for this function:



8.7.2.3 FileOutputStream() [2/3]

```
template<typename T >
io::FileOutputStream< T >::FileOutputStream (
            const FileOutputStream< T > &  )  [delete]
```

8.7.2.4 FileOutputStream() [3/3]

```
template<typename T >
io::FileOutputStream< T >::FileOutputStream (
            FileOutputStream< T > && other )  [noexcept]
```

Definition at line 506 of file file_stream.hpp.

```
00507     : id_(std::move(other.id_)),
00508       file_ptr_(other.file_ptr_),
00509       buffer_(std::move(other.buffer_)),
00510       total_elements_written_(other.total_elements_written_),
00511       total_bytes_written_(other.total_bytes_written_),
00512       finalized_(other.finalized_),
00513       serializer_member_(std::move(other.serializer_member_)) {
00514     other.file_ptr_ = nullptr;
00515     other.finalized_ = true;
00516 }
```

References io::FileOutputStream< T >::buffer_, io::FileOutputStream< T >::file_ptr_, io::FileOutputStream< T >::finalized_, io::FileOutputStream< T >::id_, io::FileOutputStream< T >::serializer_member_, io::FileOutputStream< T >::total_bytes_written_, and io::FileOutputStream< T >::total_elements_written_.

8.7.3 Member Function Documentation

8.7.3.1 Finalize()

```
template<typename T >
void io::FileOutputStream< T >::Finalize ( )  [override], [virtual]
```

Finalizes the stream, writing all buffered data to the file.

Implements io::IOutputStream< T >.

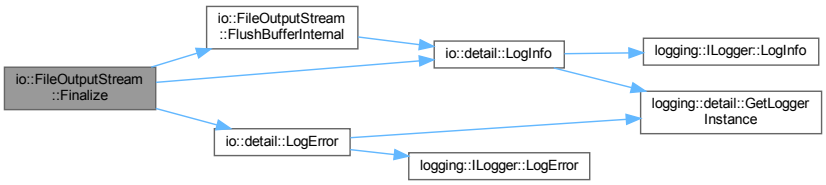
Definition at line 558 of file file_stream.hpp.

```
00558                                   {
00559     if (finalized_ || !file_ptr_) {
00560         return;
00561     }
00562     FlushBufferInternal();
00563     if (fseek(file_ptr_, 0, SEEK_SET) != 0) {
00564         detail::LogError("Error: FileOutputStream fseek failed for " + id_);
00565     } else {
00566         if (fwrite(&total_elements_written_, sizeof(uint64_t), 1, file_ptr_) != 1) {
00567             detail::LogError("Error: FileOutputStream fwrite header failed for " + id_);
00568         } else {
00569             detail::LogInfo("FileOutputStream: Finalized " + id_ +
00570                             ". Header elements: " + std::to_string(total_elements_written_));
00571         }
00572     }
00573     fflush(file_ptr_);
00574     fclose(file_ptr_);
00575     file_ptr_ = nullptr;
00576     finalized_ = true;
00577 }
```

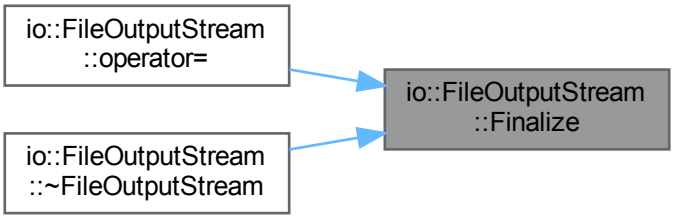
References io::FileOutputStream< T >::file_ptr_, io::FileOutputStream< T >::finalized_, io::FileOutputStream< T >::FlushBufferInte io::FileOutputStream< T >::id_, io::detail::LogError(), io::detail::LogInfo(), and io::FileOutputStream< T >::total_elements_written_.

Referenced by io::FileOutputStream< T >::operator=(), and io::FileOutputStream< T >::~FileOutputStream().

Here is the call graph for this function:



Here is the caller graph for this function:



8.7.3.2 FlushBufferInternal()

```
template<typename T >
void io::FileOutputStream< T >::FlushBufferInternal ( )  [private]
```

Flushes the buffer to the file.

Definition at line 446 of file file_stream.hpp.

```
00446                                                {
00447     if (buffer_.IsEmpty() || !file_ptr_ || finalized_) {
00448         return;
00449     }
00450
00451     uint64_t successful_writes = 0;
00452     if constexpr (serialization::PodSerializable<T>) {
00453         successful_writes = fwrite(buffer_.Data(), sizeof(T), buffer_.Size(), file_ptr_);
00454         total_bytes_written_ += successful_writes * sizeof(T);
00455     } else {
00456         const T* data = buffer_.Data();
00457         for (uint64_t i = 0; i < buffer_.Size(); ++i) {
00458             uint64_t element_size = serializer_member_->GetSerializedSize(data[i]);
00459             if (!serializer_member_->Serialize(data[i], file_ptr_)) {
00460                 throw std::runtime_error("FileOutputStream: Failed to Serialize element to file: " +
00461                                          id_);
00462             }
00463             total_bytes_written_ += element_size;
00464             successful_writes++;
00465         }
00466     }
00467
00468     if (successful_writes != buffer_.Size()) {
00469         throw std::runtime_error("FileOutputStream: Failed to write full buffer to file: " + id_);
00470     }
```

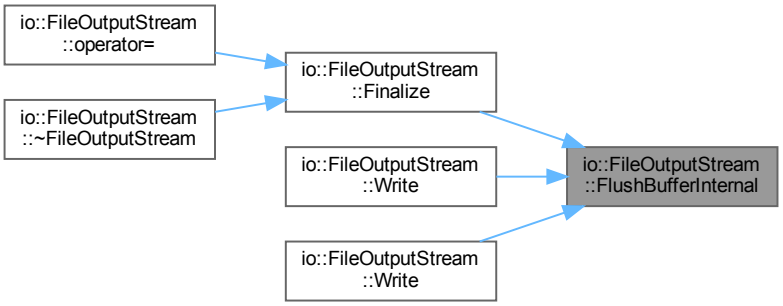
```
00471
00472     detail::LogInfo("FileOutputStream: Flushed " + std::to_string(buffer_.Size()) +
00473                     " elements to " + id_);
00474     buffer_.Clear();
00475 }
```

References io::FileOutputStream< T >::buffer_, io::FileOutputStream< T >::file_ptr_, io::FileOutputStream< T >::finalized_, io::FileOutputStream< T >::id_, io::detail::LogInfo(), io::FileOutputStream< T >::serializer_member_, and io::FileOutputStream< T >::total_bytes_written_.

Referenced by io::FileOutputStream< T >::Finalize(), io::FileOutputStream< T >::Write(), and io::FileOutputStream< T >::Write().

Here is the call graph for this function:

Here is the caller graph for this function:



8.7.3.3 GetId()

```
template<typename T >
StorageId io::FileOutputStream< T >::GetId ( ) const  [override], [virtual]
```

Returns the file identifier.

Returns

The StorageId of the file

Implements io::IOutputStream< T >.

Definition at line 590 of file file_stream.hpp.

```
00590                                   {
00591     return id_;
00592 }
```

References io::FileOutputStream< T >::id_.

8.7.3.4 GetTotalBytesWritten()

```
template<typename T >
uint64_t io::FileOutputStream< T >::GetTotalBytesWritten ( ) const  [override], [virtual]
```

Returns the total number of bytes written to the file.

Returns

The total number of bytes written (including header)

Implements io::IOutputStream< T >.

Definition at line 585 of file file_stream.hpp.

```
00585                                                    {
00586     return total_bytes_written_;
00587 }
```

References io::FileOutputStream< T >::total_bytes_written_.

8.7.3.5 GetTotalElementsWritten()

```
template<typename T >
uint64_t io::FileOutputStream< T >::GetTotalElementsWritten ( ) const  [override], [virtual]
```

Returns the total number of elements written.

Returns

The number of elements written to the stream

Implements io::IOutputStream< T >.

Definition at line 580 of file file_stream.hpp.

```
00580                                                       {
00581     return total_elements_written_;
00582 }
```

References io::FileOutputStream< T >::total_elements_written_.

8.7.3.6 operator=() [1/2]

```
template<typename T >
FileOutputStream & io::FileOutputStream< T >::operator= (
            const FileOutputStream< T > &  )  [delete]
```

8.7.3.7 operator=() [2/2]

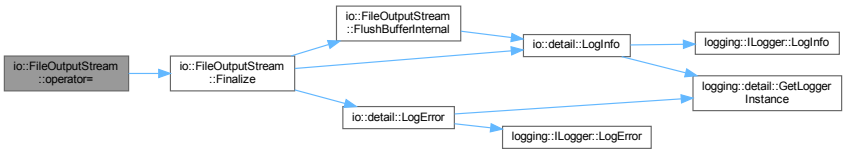
```
template<typename T >
FileOutputStream< T > & io::FileOutputStream< T >::operator= (
            FileOutputStream< T > && other )  [noexcept]
```

Definition at line 519 of file file_stream.hpp.

```
00519                                                                                {
00520     if (this != &other) {
00521         Finalize();
00522         id_ = std::move(other.id_);
00523         file_ptr_ = other.file_ptr_;
00524         buffer_ = std::move(other.buffer_);
00525         total_elements_written_ = other.total_elements_written_;
00526         total_bytes_written_ = other.total_bytes_written_;
00527         finalized_ = other.finalized_;
00528         serializer_member_ = std::move(other.serializer_member_);
00529         other.file_ptr_ = nullptr;
00530         other.finalized_ = true;
00531     }
00532     return *this;
00533 }
```

References io::FileOutputStream< T >::buffer_, io::FileOutputStream< T >::file_ptr_, io::FileOutputStream< T >::Finalize(), io::FileOutputStream< T >::finalized_, io::FileOutputStream< T >::id_, io::FileOutputStream< T >::serializer_member_, io::FileOutputStream< T >::total_bytes_written_, and io::FileOutputStream< T >::total_elements_written_.

Here is the call graph for this function:



8.7.3.8 Write() [1/2]

```
template<typename T >
void io::FileOutputStream< T >::Write (
            const T & value )  [override], [virtual]
```

Writes an element to the stream.

Parameters

value	The element to write

Exceptions

std::logic_error	if the stream has been finalized

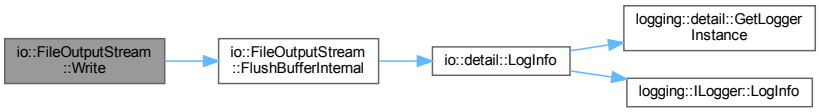
Implements io::IOutputStream< T >.

Definition at line 536 of file file_stream.hpp.

```
00536                                                {
00537     if (finalized_) {
00538         throw std::logic_error("Write to finalized FileOutputStream: " + id_);
00539     }
00540     if (buffer_.PushBack(value)) {
00541         FlushBufferInternal();
00542     }
00543     total_elements_written_++;
00544 }
```

References io::FileOutputStream< T >::buffer_, io::FileOutputStream< T >::finalized_, io::FileOutputStream< T >::FlushBufferInter
io::FileOutputStream< T >::id_, and io::FileOutputStream< T >::total_elements_written_.

Here is the call graph for this function:



8.7.3.9 Write() [2/2]

```
template<typename T >
void io::FileOutputStream< T >::Write (
            T && value )  [override], [virtual]
```

Writes an rvalue element to the stream.

Parameters

value	The rvalue element to write

Exceptions

std::logic_error	if the stream has been finalized

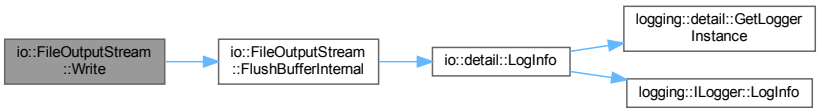
Implements io::IOutputStream< T >.

Definition at line 547 of file file_stream.hpp.

```
00547                                                {
00548     if (finalized_) {
00549         throw std::logic_error("Write to finalized FileOutputStream: " + id_);
00550     }
00551     if (buffer_.PushBack(std::move(value))) {
00552         FlushBufferInternal();
00553     }
00554     total_elements_written_++;
00555 }
```

References io::FileOutputStream< T >::buffer_, io::FileOutputStream< T >::finalized_, io::FileOutputStream< T >::FlushBufferInter
io::FileOutputStream< T >::id_, and io::FileOutputStream< T >::total_elements_written_.

Here is the call graph for this function:



8.7.4 Member Data Documentation

8.7.4.1 buffer_

```
template<typename T >
ElementBuffer<T> io::FileOutputStream< T >::buffer_  [private]
```

Buffer for writing.

Definition at line 122 of file file_stream.hpp.

Referenced by io::FileOutputStream< T >::FileOutputStream(), io::FileOutputStream< T >::FileOutputStream(), io::FileOutputStream< T >::FlushBufferInternal(), io::FileOutputStream< T >::operator=(), io::FileOutputStream< T >::Write(), and io::FileOutputStream< T >::Write().

8.7.4.2 file_ptr_

```
template<typename T >
FILE* io::FileOutputStream< T >::file_ptr_ = nullptr  [private]
```

Pointer to the file.

Definition at line 121 of file file_stream.hpp.

Referenced by io::FileOutputStream< T >::FileOutputStream(), io::FileOutputStream< T >::FileOutputStream(), io::FileOutputStream< T >::Finalize(), io::FileOutputStream< T >::FlushBufferInternal(), and io::FileOutputStream< T >::operator=(

8.7.4.3 finalized_

```
template<typename T >
bool io::FileOutputStream< T >::finalized_ = false  [private]
```

Flag indicating if the stream has been finalized.

Definition at line 125 of file file_stream.hpp.

Referenced by io::FileOutputStream< T >::FileOutputStream(), io::FileOutputStream< T >::Finalize(), io::FileOutputStream< T >::F io::FileOutputStream< T >::operator=(), io::FileOutputStream< T >::Write(), and io::FileOutputStream< T >::Write().

8.7.4.4 id_

```
template<typename T >
StorageId io::FileOutputStream< T >::id_  [private]
```

Storage identifier (file name)

Definition at line 120 of file file_stream.hpp.

Referenced by io::FileOutputStream< T >::FileOutputStream(), io::FileOutputStream< T >::FileOutputStream(), io::FileOutputStream< T >::Finalize(), io::FileOutputStream< T >::FlushBufferInternal(), io::FileOutputStream< T >::GetId(), io::FileOutputStream< T >::operator=(), io::FileOutputStream< T >::Write(), and io::FileOutputStream< T >::Write().

8.7.4.5 serializer_member_

```
template<typename T >
std::unique_ptr<serialization::Serializer<T> > io::FileOutputStream< T >::serializer_↩
member_ {}  [private]
```

Serializer for the elements.

Definition at line 127 of file file_stream.hpp.

```
00127 {};
```

Referenced by io::FileOutputStream< T >::FileOutputStream(), io::FileOutputStream< T >::FileOutputStream(), io::FileOutputStream< T >::FlushBufferInternal(), and io::FileOutputStream< T >::operator=().

8.7.4.6 total_bytes_written_

```
template<typename T >
uint64_t io::FileOutputStream< T >::total_bytes_written_ = 0  [private]
```

Total bytes written (data + header)

Definition at line 124 of file file_stream.hpp.

Referenced by io::FileOutputStream< T >::FileOutputStream(), io::FileOutputStream< T >::FileOutputStream(), io::FileOutputStream< T >::FlushBufferInternal(), io::FileOutputStream< T >::GetTotalBytesWritten(), and io::FileOutputStream< T >::operator=().

8.7.4.7 total_elements_written_

```
template<typename T >
uint64_t io::FileOutputStream< T >::total_elements_written_ = 0  [private]
```

Number of elements written.

Definition at line 123 of file file_stream.hpp.

Referenced by io::FileOutputStream< T >::FileOutputStream(), io::FileOutputStream< T >::Finalize(), io::FileOutputStream< T >::G
io::FileOutputStream< T >::operator=(), io::FileOutputStream< T >::Write(), and io::FileOutputStream< T >::Write().

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

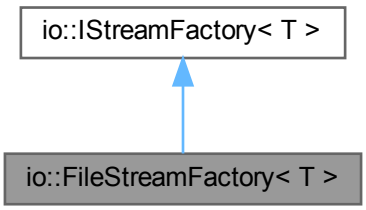
- io/include/file_stream.hpp

8.8 io::FileStreamFactory< T > Class Template Reference

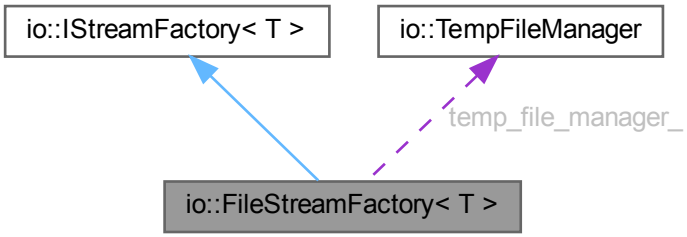
Factory for file streams.

`#include <file_stream.hpp>`

Inheritance diagram for io::FileStreamFactory< T >:



Collaboration diagram for io::FileStreamFactory< T >:



Public Member Functions

- FileStreamFactory (const std::string &base_temp_dir_name="temp_files")

  Constructor for the file stream factory.
- std::unique_ptr< IInputStream< T > > CreateInputStream (const StorageId &id, uint64_t buffer_capacity_elements) override

  Creates an input stream for reading from a file.
- std::unique_ptr< IOutputStream< T > > CreateOutputStream (const StorageId &id, uint64_t buffer_capacity_elements) override

  Creates an output stream for writing to a file.
- std::unique_ptr< IOutputStream< T > > CreateTempOutputStream (StorageId &out_temp_id, uint64_t buffer_capacity_elements) override

  Creates an output stream for writing to a temporary file.

- void DeleteStorage (const StorageId &id) override

  Deletes a file.
- void MakeStoragePermanent (const StorageId &temp_id, const StorageId &final_id) override

  Makes a temporary file permanent.
- bool StorageExists (const StorageId &id) const override

  Checks if a file exists.
- StorageId GetTempStorageContextId () const override

  Returns the path to the temporary files directory.

Public Member Functions inherited from io::IStreamFactory< T >

- virtual ~IStreamFactory ()=default

Private Attributes

- TempFileManager temp_file_manager_

  Temporary file manager.

8.8.1 Detailed Description

template<typename T>
class io::FileStreamFactory< T >

Factory for file streams.

Implements the IStreamFactory interface for creating file-based I/O streams and managing temporary files.

Template Parameters

T	The type of elements in the streams

Definition at line 201 of file file_stream.hpp.

8.8.2 Constructor & Destructor Documentation

8.8.2.1 FileStreamFactory()

```
template<typename T >
io::FileStreamFactory< T >::FileStreamFactory (
            const std::string & base_temp_dir_name = "temp_files" )  [explicit]
```

Constructor for the file stream factory.

Parameters

base_temp_dir_name	The name of the base directory for temporary files

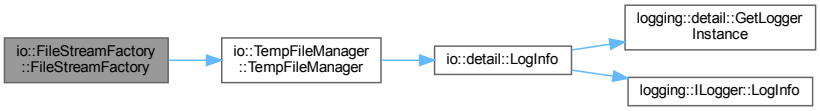
Definition at line 595 of file file_stream.hpp.

```
00596     : temp_file_manager_(base_temp_dir_name) {
00597 }
```

References io::FileStreamFactory< T >::temp_file_manager_, and io::TempFileManager::TempFileManager().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:

8.8.3 Member Function Documentation

8.8.3.1 CreateInputStream()

```
template<typename T >
std::unique_ptr< IInputStream< T > > io::FileStreamFactory< T >::CreateInputStream (
            const StorageId & id,
            uint64_t buffer_capacity_elements )  [override], [virtual]
```

Creates an input stream for reading from a file.

Parameters

id	The identifier of the file
buffer_capacity_elements	The buffer capacity

Returns

A unique pointer to an IInputStream

Implements io::IStreamFactory< T >.

Definition at line 600 of file file_stream.hpp.

```
00601                                                                {
00602     return std::make_unique<FileInputStream<T»(id, buffer_capacity_elements);
00603 }
```

Referenced by main().

Here is the caller graph for this function:

8.8.3.2 CreateOutputStream()

```
template<typename T >
std::unique_ptr< IOutputStream< T > > io::FileStreamFactory< T >::CreateOutputStream (
            const StorageId & id,
            uint64_t buffer_capacity_elements )  [override], [virtual]
```

Creates an output stream for writing to a file.

Parameters

id	The identifier of the file
buffer_capacity_elements	The buffer capacity

Returns

A unique pointer to an IOutputStream

Implements io::IStreamFactory< T >.

Definition at line 607 of file file_stream.hpp.

```
00608                                                                {
00609     return std::make_unique<FileOutputStream<T»(id, buffer_capacity_elements);
00610 }
```

Referenced by main().

Here is the caller graph for this function:

8.8.3.3 CreateTempOutputStream()

```
template<typename T >
std::unique_ptr< IOutputStream< T > > io::FileStreamFactory< T >::CreateTempOutputStream (
            StorageId & out_temp_id,
            uint64_t buffer_capacity_elements )  [override], [virtual]
```

Creates an output stream for writing to a temporary file.

Parameters

out_temp_id	Output parameter for the temporary file's ID
buffer_capacity_elements	The buffer capacity

Returns

A unique pointer to an IOutputStream

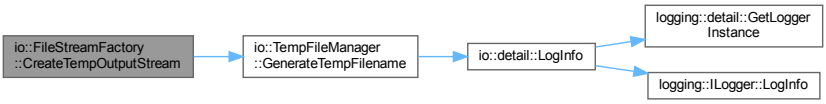
Implements io::IStreamFactory< T >.

Definition at line 614 of file file_stream.hpp.

```
00615                                                                {
00616     out_temp_id = temp_file_manager_.GenerateTempFilename("r", ".b");
00617     return std::make_unique<FileOutputStream<T»(out_temp_id, buffer_capacity_elements);
00618 }
```

References io::TempFileManager::GenerateTempFilename(), and io::FileStreamFactory< T >::temp_file_manager_.

Here is the call graph for this function:



8.8.3.4 DeleteStorage()

```
template<typename T >
void io::FileStreamFactory< T >::DeleteStorage (
            const StorageId & id )  [override], [virtual]
```

Deletes a file.

Parameters

id	The identifier of the file to delete

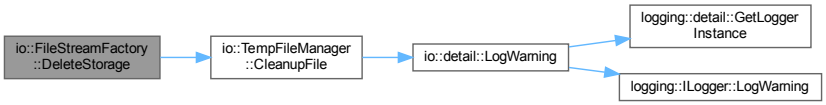
Implements io::IStreamFactory< T >.

Definition at line 622 of file file_stream.hpp.

```
00622                                                    {
00623     temp_file_manager_.CleanupFile(id);
00624 }
```

References io::TempFileManager::CleanupFile(), and io::FileStreamFactory< T >::temp_file_manager_.

Here is the call graph for this function:



8.8.3.5 GetTempStorageContextId()

```
template<typename T >
StorageId io::FileStreamFactory< T >::GetTempStorageContextId ( ) const  [override], [virtual]
```

Returns the path to the temporary files directory.

Returns

The path to the temporary files directory

Implements io::IStreamFactory< T >.

Definition at line 661 of file file_stream.hpp.

```
00661                                                    {
00662     return temp_file_manager_.GetBaseDirPath().string();
00663 }
```

References io::TempFileManager::GetBaseDirPath(), and io::FileStreamFactory< T >::temp_file_manager_.

Here is the call graph for this function:

8.8.3.6 MakeStoragePermanent()

```
template<typename T >
void io::FileStreamFactory< T >::MakeStoragePermanent (
            const StorageId & temp_id,
            const StorageId & final_id )  [override], [virtual]
```

Makes a temporary file permanent.

Parameters

temp↩ _id	The identifier of the temporary file
final_id	The identifier of the final file

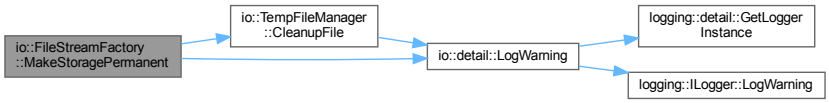
Implements io::IStreamFactory< T >.

Definition at line 628 of file file_stream.hpp.

```
00629                                                                    {
00630     if (temp_id == final_id) {
00631         return;
00632     }
00633     try {
00634         if (std::filesystem::exists(final_id)) {
00635             std::filesystem::remove(final_id);
00636         }
00637         std::filesystem::rename(temp_id, final_id);
00638     } catch (const std::filesystem::filesystem_error& e) {
00639         detail::LogWarning("MakeStoragePermanent: Rename failed (" + std::string(e.what()) +
00640                            "), attempting copy for " + temp_id + " to " + final_id);
00641         {
00642             FileInputStream<T> src(temp_id, 1024);
00643             FileOutputStream<T> dst(final_id, 1024);
00644             while (!src.IsExhausted()) {
00645                 dst.Write(src.TakeValue());
00646                 src.Advance();
00647             }
00648         }
00649         temp_file_manager_.CleanupFile(temp_id);
00650     }
00651 }
```

References io::TempFileManager::CleanupFile(), io::detail::LogWarning(), and io::FileStreamFactory< T >::temp_file_manager_.

Here is the call graph for this function:



8.8.3.7 StorageExists()

```
template<typename T >
bool io::FileStreamFactory< T >::StorageExists (
            const StorageId & id ) const  [override], [virtual]
```

Checks if a file exists.

Parameters

id	The identifier of the file

Returns

true if the file exists

Implements io::IStreamFactory< T >.

Definition at line 655 of file file_stream.hpp.

```
00655                                                    {
00656     return std::filesystem::exists(id);
00657 }
```

8.8.4 Member Data Documentation

8.8.4.1 temp_file_manager_

```
template<typename T >
TempFileManager io::FileStreamFactory< T >::temp_file_manager_  [private]
```

Temporary file manager.

Definition at line 203 of file file_stream.hpp.

Referenced by io::FileStreamFactory< T >::CreateTempOutputStream(), io::FileStreamFactory< T >::DeleteStorage(), io::FileStreamFactory< T >::FileStreamFactory(), io::FileStreamFactory< T >::GetTempStorageContextId(), and io::FileStreamFactory< T >::MakeStoragePermanent().

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

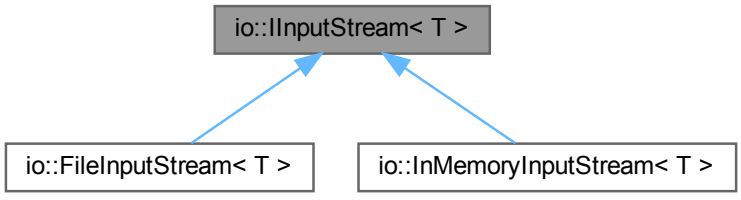
- io/include/file_stream.hpp

8.9 io::IInputStream< T > Class Template Reference

Interface for input stream of elements of type T.

```
#include <interfaces.hpp>
```

Inheritance diagram for io::IInputStream< T >:



Public Member Functions

- virtual ~IInputStream ()=default
- virtual void Advance ()=0

  Advances stream to the next element.
- virtual const T & Value () const =0

  Returns current element of the stream.
- virtual T TakeValue ()=0

  Move out the current element of the stream.
- virtual bool IsExhausted () const =0

  Checks if the stream is exhausted.
- virtual bool IsEmptyOriginalStorage () const =0

  Checks if original storage is empty.

8.9.1 Detailed Description

template<typename T>
class io::IInputStream< T >

Interface for input stream of elements of type T.

Template Parameters

T	type of elements in stream

Definition at line 20 of file interfaces.hpp.

8.9.2 Constructor & Destructor Documentation

8.9.2.1 ~IInputStream()

```
template<typename T >
virtual io::IInputStream< T >::~IInputStream ( )  [virtual], [default]
```

8.9.3 Member Function Documentation

8.9.3.1 Advance()

```
template<typename T >
virtual void io::IInputStream< T >::Advance ( )  [pure virtual]
```

Advances stream to the next element.

Implemented in io::FileInputStream< T >, and io::InMemoryInputStream< T >.

Referenced by main().

Here is the caller graph for this function:

8.9.3.2 IsEmptyOriginalStorage()

```
template<typename T >
virtual bool io::IInputStream< T >::IsEmptyOriginalStorage ( ) const  [pure virtual]
```

Checks if original storage is empty.

Returns

true, if original storage doesn't contains eny elements

Implemented in io::FileInputStream< T >, and io::InMemoryInputStream< T >.

8.9.3.3 IsExhausted()

```
template<typename T >
virtual bool io::IInputStream< T >::IsExhausted ( ) const  [pure virtual]
```

Checks if the stream is exhausted.

Returns

true, if there are no more elements to read

Implemented in io::FileInputStream< T >, and io::InMemoryInputStream< T >.

Referenced by main().

Here is the caller graph for this function:

8.9.3.4 TakeValue()

```
template<typename T >
virtual T io::IInputStream< T >::TakeValue ( )  [pure virtual]
```

Move out the current element of the stream.

Returns the current element by value using move semantics where possible. Does not advance the stream; call Advance() to move to the next element.

Returns

Current element by value

Exceptions

std::logic_error	if the stream is exhausted

Implemented in io::FileInputStream< T >, and io::InMemoryInputStream< T >.

Referenced by main().

Here is the caller graph for this function:

8.9.3.5 Value()

```
template<typename T >
virtual const T & io::IInputStream< T >::Value ( ) const  [pure virtual]
```

Returns current element of the stream.

Returns

Constant reference to the current element of the stream

Exceptions

std::logic_error	if the stream is exhausted

Implemented in io::FileInputStream< T >, and io::InMemoryInputStream< T >.

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

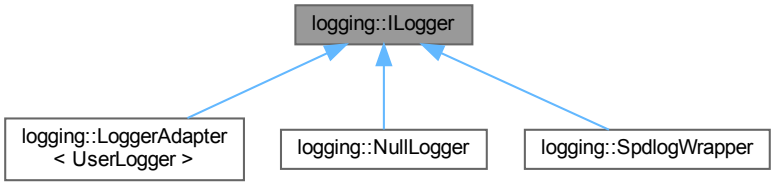
- io/include/interfaces.hpp

8.10 logging::ILogger Class Reference

Abstract base class defining the logger interface.

`#include <ILogger.hpp>`

Inheritance diagram for logging::ILogger:



Public Member Functions

- virtual ~ILogger ()=default
- virtual void LogInfo (const std::string &message)=0

  Log an informational message.
- virtual void LogWarning (const std::string &message)=0

  Log a warning message.
- virtual void LogError (const std::string &message)=0

  Log an error message.

8.10.1 Detailed Description

Abstract base class defining the logger interface.

All logger implementations must inherit from this interface and implement the three logging methods for different severity levels: info, warning, and error. This allows the library to work with any logging backend through polymorphism.

Definition at line 20 of file ILogger.hpp.

8.10.2 Constructor & Destructor Documentation

8.10.2.1 ~ILogger()

```
virtual logging::ILogger::~ILogger ( )  [virtual], [default]
```

8.10.3 Member Function Documentation

8.10.3.1 LogError()

```
virtual void logging::ILogger::LogError (
            const std::string & message )  [pure virtual]
```

Log an error message.

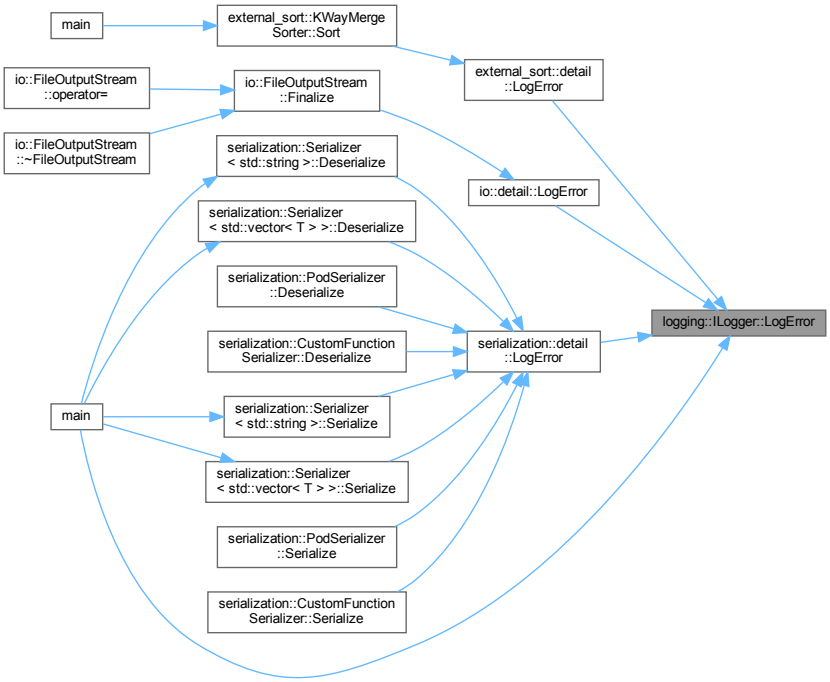
Parameters

message	The message to log

Implemented in logging::LoggerAdapter< UserLogger >, logging::NullLogger, and logging::SpdlogWrapper.

Referenced by external_sort::detail::LogError(), io::detail::LogError(), serialization::detail::LogError(), and main().

Here is the caller graph for this function:



8.10.3.2 LogInfo()

```
virtual void logging::ILogger::LogInfo (
            const std::string & message )  [pure virtual]
```

Log an informational message.

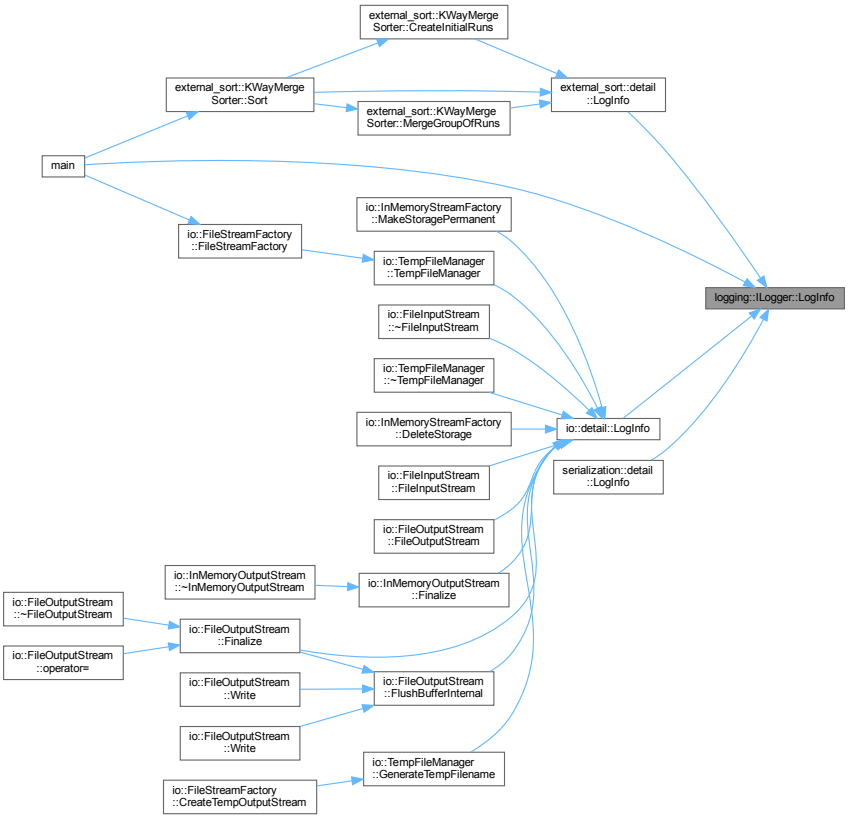
Parameters

message	The message to log

Implemented in logging::LoggerAdapter< UserLogger >, logging::NullLogger, and logging::SpdlogWrapper.

Referenced by external_sort::detail::LogInfo(), io::detail::LogInfo(), serialization::detail::LogInfo(), and main().

Here is the caller graph for this function:



8.10.3.3 LogWarning()

```
virtual void logging::ILogger::LogWarning (
            const std::string & message )  [pure virtual]
```

Log a warning message.

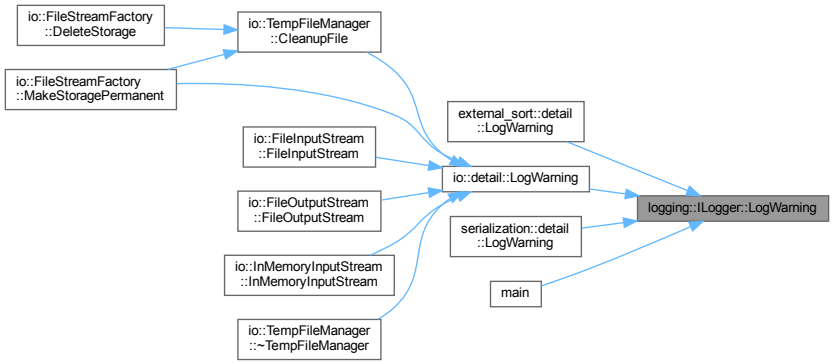
Parameters

message	The message to log

Implemented in logging::LoggerAdapter< UserLogger >, logging::NullLogger, and logging::SpdlogWrapper.

Referenced by external_sort::detail::LogWarning(), io::detail::LogWarning(), serialization::detail::LogWarning(), and main().

Here is the caller graph for this function:



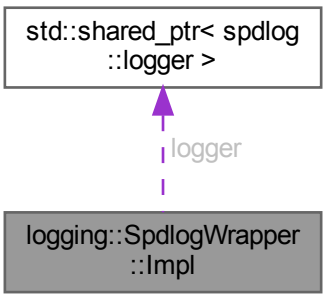
The documentation for this class was generated from the following file:

- logging/include/ILogger.hpp

8.11 logging::SpdlogWrapper::Impl Class Reference

Implementation class hiding spdlog dependencies.

Collaboration diagram for logging::SpdlogWrapper::Impl:



Public Member Functions

- Impl ()=delete
- Impl (const std::string &name, SpdlogSinkType sink_type, const std::string &filename)

  Construct with specified sink configuration.

Public Attributes

- std::shared_ptr< spdlog::logger > logger

  The underlying spdlog logger.

8.11.1 Detailed Description

Implementation class hiding spdlog dependencies.

This nested class contains the actual spdlog logger instance, keeping spdlog headers out of the public interface.

Definition at line 27 of file SpdlogWrapper.cpp.

8.11.2 Constructor & Destructor Documentation

8.11.2.1 Impl() [1/2]

```
logging::SpdlogWrapper::Impl::Impl ( )  [delete]
```

8.11.2.2 Impl() [2/2]

```
logging::SpdlogWrapper::Impl::Impl (
            const std::string & name,
            SpdlogSinkType sink_type,
            const std::string & filename )  [inline]
```

Construct with specified sink configuration.

Parameters

sink_type	Type of sink to create
filename	Filename for file sink (if applicable)

Definition at line 36 of file SpdlogWrapper.cpp.

```
00036                                                                                {
00037         std::vector<spdlog::sink_ptr> sinks;
00038
00039         switch (sink_type) {
00040             case SpdlogSinkType::Console: {
00041                 auto console_sink = std::make_shared<spdlog::sinks::stdout_color_sink_mt>();
00042                 sinks.push_back(console_sink);
00043                 break;
00044             }
00045             case SpdlogSinkType::File: {
00046                 if (filename.empty()) {
00047                     throw std::invalid_argument("Filename required for File sink type");
00048                 }
00049                 auto file_sink =
00050                     std::make_shared<spdlog::sinks::basic_file_sink_mt>(filename, true);
00051                 sinks.push_back(file_sink);
00052                 break;
00053             }
00054             case SpdlogSinkType::Both: {
00055                 if (filename.empty()) {
00056                     throw std::invalid_argument("Filename required for Both sink type");
00057                 }
00058                 auto console_sink = std::make_shared<spdlog::sinks::stdout_color_sink_mt>();
```

```
00059                 auto file_sink =
00060                     std::make_shared<spdlog::sinks::basic_file_sink_mt>(filename, true);
00061                 sinks.push_back(console_sink);
00062                 sinks.push_back(file_sink);
00063                 break;
00064             }
00065         }
00066 
00067         logger = std::make_shared<spdlog::logger>(name + "_" + std::to_string(logger_counter++),
00068                                                   sinks.begin(), sinks.end());
00069         logger->set_level(spdlog::level::trace);
00070     }
```

References logging::Both, logging::Console, and logging::File.

8.11.3 Member Data Documentation

8.11.3.1 logger

`std::shared_ptr<spdlog::logger> logging::SpdlogWrapper::Impl::logger`

The underlying spdlog logger.

Definition at line 72 of file SpdlogWrapper.cpp.

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

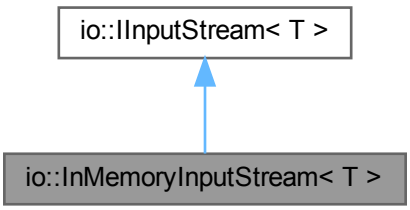
- logging/src/SpdlogWrapper.cpp

8.12 io::InMemoryInputStream< T > Class Template Reference

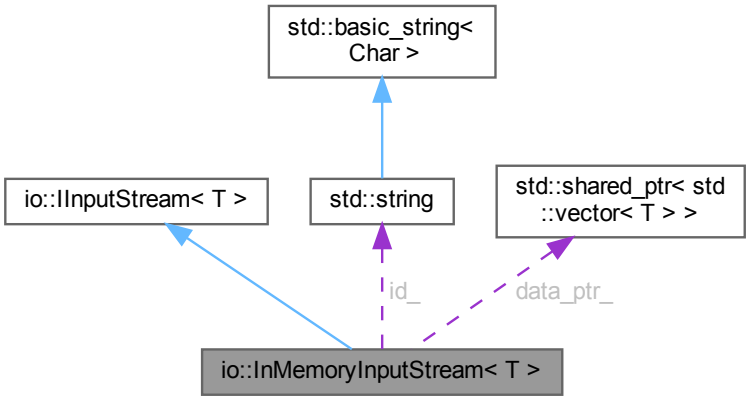
In-memory implementation of input stream.

`#include <memory_stream.hpp>`

Inheritance diagram for io::InMemoryInputStream< T >:



Collaboration diagram for io::InMemoryInputStream< T >:



Public Member Functions

- InMemoryInputStream (StorageId id, std::shared_ptr< std::vector< T > > data_vec_ptr, uint64_t actual_↩ storage_size, uint64_t buffer_capacity)

  Constructor for in-memory input stream.
- ~InMemoryInputStream () override=default

  Destructor.
- void Advance () override

  Advances the stream to the next element.
- const T & Value () const override

  Returns the current element of the stream.
- T TakeValue () override

  Takes the current element of the stream (removing it from the stream)
- bool IsExhausted () const override

  Checks if the stream is exhausted.
- bool IsEmptyOriginalStorage () const override

  Checks if the original storage was empty.

Public Member Functions inherited from io::IInputStream< T >

- virtual ~IInputStream ()=default

Private Attributes

- StorageId id_

  Stream identifier.
- std::shared_ptr< std::vector< T > > data_ptr_

  Pointer to data (modifiable to allow move-out of elements)
- uint64_t total_elements_in_storage_

Total number of elements in storage.

- uint64_t read_cursor_ = 0

  Read cursor.

- T current_value_ {}

  Current element.

- bool has_valid_value_ = false

  Current element validity flag.

- bool is_exhausted_ = false

  Stream exhaustion flag.

8.12.1 Detailed Description

template<typename T>
class io::InMemoryInputStream< T >

In-memory implementation of input stream.

Implements the IInputStream interface for reading data from memory. Reads data from std::vector.

Template Parameters

T	Type of elements in the stream

Definition at line 102 of file memory_stream.hpp.

8.12.2 Constructor & Destructor Documentation

8.12.2.1 InMemoryInputStream()

```
template<typename T >
io::InMemoryInputStream< T >::InMemoryInputStream (
            StorageId id,
            std::shared_ptr< std::vector< T > > data_vec_ptr,
            uint64_t actual_storage_size,
            uint64_t buffer_capacity )
```

Constructor for in-memory input stream.

Parameters

id	Stream identifier
data_vec_ptr	Pointer to data vector
actual_storage_size	Actual data size
buffer_capacity	Buffer capacity (not used in in-memory)

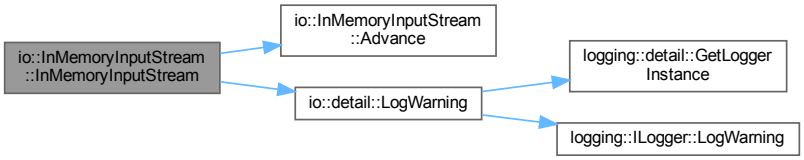
Definition at line 324 of file memory_stream.hpp.

```
00327     : id_(std::move(id))
00328     , data_ptr_(std::move(data_vec_ptr))
00329     , total_elements_in_storage_(actual_storage_size) {
```

```
00330     if (total_elements_in_storage_ > data_ptr_->size()) {
00331         detail::LogWarning("Warning: InMemoryInputStream " + id_ +
00332                            " declared size (" + std::to_string(total_elements_in_storage_) +
00333                            ") > actual vector size (" + std::to_string(data_ptr_->size()) +
00334                            "). Clamping to actual size.");
00335         total_elements_in_storage_ = data_ptr_->size();
00336     }
00337
00338     if (total_elements_in_storage_ == 0) {
00339         is_exhausted_ = true;
00340         has_valid_value_ = false;
00341     } else {
00342         Advance();
00343     }
00344 }
```

References io::InMemoryInputStream< T >::Advance(), io::InMemoryInputStream< T >::data_ptr_, io::InMemoryInputStream< T >::id_, io::InMemoryInputStream< T >::is_exhausted_, io::detail::LogWarning(), and io::InMemoryInputStream< T >::total_elements_in_storage_.

Here is the call graph for this function:



8.12.2.2 ~InMemoryInputStream()

```
template<typename T >
io::InMemoryInputStream< T >::~InMemoryInputStream ( )  [override], [default]
```

Destructor.

8.12.3 Member Function Documentation

8.12.3.1 Advance()

```
template<typename T >
void io::InMemoryInputStream< T >::Advance ( )  [override], [virtual]
```

Advances the stream to the next element.

Implements io::IInputStream< T >.

Definition at line 347 of file memory_stream.hpp.

```
00347                                {
00348     if (is_exhausted_ || read_cursor_ >= total_elements_in_storage_) {
00349         has_valid_value_ = false;
00350         is_exhausted_ = true;
00351         return;
00352     }
00353     // Move the element out of the underlying storage to avoid copies.
00354     current_value_ = std::move((*data_ptr_)[read_cursor_]);
00355     read_cursor_++;
```

```
00356     has_valid_value_ = true;
00357     if (read_cursor_ >= total_elements_in_storage_) {
00358         is_exhausted_ = true;
00359     }
00360 }
```

References io::InMemoryInputStream< T >::current_value_, io::InMemoryInputStream< T >::data_ptr_, io::InMemoryInputStream< io::InMemoryInputStream< T >::is_exhausted_, io::InMemoryInputStream< T >::read_cursor_, and io::InMemoryInputStream< T >

Referenced by io::InMemoryInputStream< T >::InMemoryInputStream().

Here is the caller graph for this function:

8.12.3.2 IsEmptyOriginalStorage()

```
template<typename T >
bool io::InMemoryInputStream< T >::IsEmptyOriginalStorage ( ) const  [override], [virtual]
```

Checks if the original storage was empty.

Returns

true if the storage contained no elements

Implements io::IInputStream< T >.

Definition at line 385 of file memory_stream.hpp.

```
00385                                                 {
00386     return total_elements_in_storage_ == 0;
00387 }
```

References io::InMemoryInputStream< T >::total_elements_in_storage_.

8.12.3.3 IsExhausted()

```
template<typename T >
bool io::InMemoryInputStream< T >::IsExhausted ( ) const  [override], [virtual]
```

Checks if the stream is exhausted.

Returns

true if there are no more elements to read

Implements io::IInputStream< T >.

Definition at line 380 of file memory_stream.hpp.

```
00380                                           {
00381     return is_exhausted_ && !has_valid_value_;
00382 }
```

References io::InMemoryInputStream< T >::has_valid_value_, and io::InMemoryInputStream< T >::is_exhausted_.

8.12.3.4 TakeValue()

```
template<typename T >
T io::InMemoryInputStream< T >::TakeValue ( )  [override], [virtual]
```

Takes the current element of the stream (removing it from the stream)

Returns

The current element

Exceptions

std::logic_error	if the stream is exhausted

Implements io::IInputStream< T >.

Definition at line 371 of file memory_stream.hpp.

```
00371                                       {
00372     if (!has_valid_value_) {
00373         throw std::logic_error("TakeValue from exhausted InMemoryInputStream: " + id_);
00374     }
00375     T tmp = std::move(current_value_);
00376     return tmp;
00377 }
```

References io::InMemoryInputStream< T >::current_value_, io::InMemoryInputStream< T >::has_valid_value_, and io::InMemoryInputStream< T >::id_.

8.12.3.5 Value()

```
template<typename T >
const T & io::InMemoryInputStream< T >::Value ( ) const  [override], [virtual]
```

Returns the current element of the stream.

Returns

Constant reference to the current element

Exceptions

std::logic_error	if the stream is exhausted

Implements io::IInputStream< T >.

Definition at line 363 of file memory_stream.hpp.

```
00363                                             {
00364     if (!has_valid_value_) {
00365         throw std::logic_error("Value from exhausted InMemoryInputStream: " + id_);
00366     }
00367     return current_value_;
00368 }
```

References io::InMemoryInputStream< T >::current_value_, io::InMemoryInputStream< T >::has_valid_value_, and io::InMemoryInputStream< T >::id_.

8.12.4 Member Data Documentation

8.12.4.1 current_value_

```
template<typename T >
T io::InMemoryInputStream< T >::current_value_ {}  [private]
```

Current element.

Definition at line 108 of file memory_stream.hpp.
```
00108 {};
```

Referenced by io::InMemoryInputStream< T >::Advance(), io::InMemoryInputStream< T >::TakeValue(), and io::InMemoryInputStream< T >::Value().

8.12.4.2 data_ptr_

```
template<typename T >
std::shared_ptr<std::vector<T> > io::InMemoryInputStream< T >::data_ptr_  [private]
```

Pointer to data (modifiable to allow move-out of elements)

Definition at line 105 of file memory_stream.hpp.

Referenced by io::InMemoryInputStream< T >::Advance(), and io::InMemoryInputStream< T >::InMemoryInputStream().

8.12.4.3 has_valid_value_

```
template<typename T >
bool io::InMemoryInputStream< T >::has_valid_value_ = false  [private]
```

Current element validity flag.

Definition at line 109 of file memory_stream.hpp.

Referenced by io::InMemoryInputStream< T >::Advance(), io::InMemoryInputStream< T >::InMemoryInputStream(), io::InMemoryInputStream< T >::IsExhausted(), io::InMemoryInputStream< T >::TakeValue(), and io::InMemoryInputStream< T >::

8.12.4.4 id_

```
template<typename T >
StorageId io::InMemoryInputStream< T >::id_  [private]
```

Stream identifier.

Definition at line 104 of file memory_stream.hpp.

Referenced by io::InMemoryInputStream< T >::InMemoryInputStream(), io::InMemoryInputStream< T >::TakeValue(), and io::InMemoryInputStream< T >::Value().

8.12.4.5 is_exhausted_

```
template<typename T >
bool io::InMemoryInputStream< T >::is_exhausted_ = false  [private]
```

Stream exhaustion flag.

Definition at line 110 of file memory_stream.hpp.

Referenced by io::InMemoryInputStream< T >::Advance(), io::InMemoryInputStream< T >::InMemoryInputStream(), and io::InMemoryInputStream< T >::IsExhausted().

8.12.4.6 read_cursor_

```
template<typename T >
uint64_t io::InMemoryInputStream< T >::read_cursor_ = 0  [private]
```

Read cursor.

Definition at line 107 of file memory_stream.hpp.

Referenced by io::InMemoryInputStream< T >::Advance().

8.12.4.7 total_elements_in_storage_

```
template<typename T >
uint64_t io::InMemoryInputStream< T >::total_elements_in_storage_  [private]
```

Total number of elements in storage.

Definition at line 106 of file memory_stream.hpp.

Referenced by io::InMemoryInputStream< T >::Advance(), io::InMemoryInputStream< T >::InMemoryInputStream(), and io::InMemoryInputStream< T >::IsEmptyOriginalStorage().

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

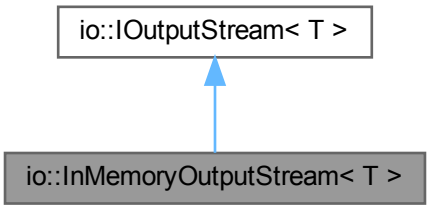
- io/include/memory_stream.hpp

8.13 io::InMemoryOutputStream< T > Class Template Reference

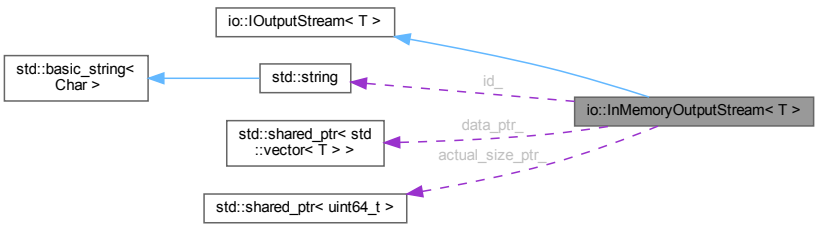
In-memory implementation of output stream.

`#include <memory_stream.hpp>`

Inheritance diagram for io::InMemoryOutputStream< T >:



Collaboration diagram for io::InMemoryOutputStream< T >:



Public Member Functions

- InMemoryOutputStream (StorageId id, std::shared_ptr< std::vector< T > > data_vec_ptr, std::shared_ptr< uint64_t > size_ptr, uint64_t buffer_capacity)

  Constructor for in-memory output stream.
- ~InMemoryOutputStream () override

  Destructor that finalizes the stream.
- void Write (const T &value) override

  Writes an element to the stream.
- void Write (T &&value) override

  Writes an rvalue element to the stream.
- void Finalize () override

  Finalizes the stream.
- uint64_t GetTotalElementsWritten () const override

  Returns the total number of elements written.
- uint64_t GetTotalBytesWritten () const override

  Returns the total number of bytes that would be written if serialized.
- StorageId GetId () const override

  Returns the stream identifier.

Public Member Functions inherited from io::IOutputStream< T >

- virtual ~IOutputStream ()=default

Private Attributes

- StorageId id_

  Stream identifier.
- std::shared_ptr< std::vector< T > > data_ptr_

  Pointer to data.
- std::shared_ptr< uint64_t > actual_size_ptr_

  Pointer to size ("header")
- uint64_t elements_written_ = 0

  Number of elements written.
- bool finalized_ = false

  Finalization flag.

8.13.1 Detailed Description

template<typename T>
class io::InMemoryOutputStream< T >

In-memory implementation of output stream.

Implements the IOutputStream interface for writing data to memory. All data is stored in std::vector.

Note: Supports move-friendly Write(T&&) so callers can avoid copies when they have rvalue elements or use std::move().

Template Parameters

T	Type of elements in the stream

Definition at line 30 of file memory_stream.hpp.

8.13.2 Constructor & Destructor Documentation

8.13.2.1 InMemoryOutputStream()

```
template<typename T >
io::InMemoryOutputStream< T >::InMemoryOutputStream (
            StorageId id,
            std::shared_ptr< std::vector< T > > data_vec_ptr,
            std::shared_ptr< uint64_t > size_ptr,
            uint64_t buffer_capacity )
```

Constructor for in-memory output stream.

Parameters

id	Stream identifier
data_vec_ptr	Pointer to data vector
size_ptr	Pointer to data size
buffer_capacity	Buffer capacity (not used in in-memory)

Definition at line 255 of file memory_stream.hpp.

```
00258     : id_(std::move(id))
00259     , data_ptr_(std::move(data_vec_ptr))
00260     , actual_size_ptr_(std::move(size_ptr)) {
00261     data_ptr_->clear();
00262     *actual_size_ptr_ = 0;
00263 }
```

References io::InMemoryOutputStream< T >::actual_size_ptr_, io::InMemoryOutputStream< T >::data_ptr_, and io::InMemoryOutputStream< T >::id_.

8.13.2.2 ~InMemoryOutputStream()

```
template<typename T >
io::InMemoryOutputStream< T >::~InMemoryOutputStream ( )  [override]
```

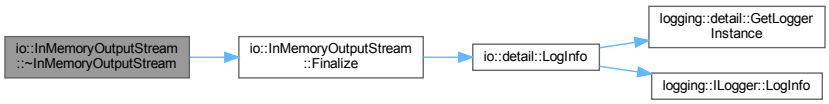
Destructor that finalizes the stream.

Definition at line 266 of file memory_stream.hpp.

```
00266                                                {
00267     Finalize();
00268 }
```

References io::InMemoryOutputStream< T >::Finalize().

Here is the call graph for this function:



8.13.3 Member Function Documentation

8.13.3.1 Finalize()

```
template<typename T >
void io::InMemoryOutputStream< T >::Finalize ( )  [override], [virtual]
```

Finalizes the stream.

Implements io::IOutputStream< T >.

Definition at line 289 of file memory_stream.hpp.

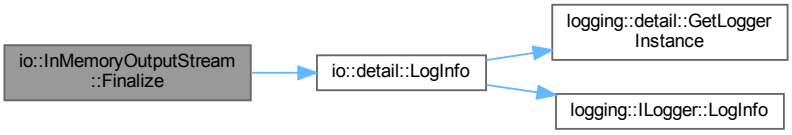
```
00289                                          {
00290     if (finalized_) {
```

```
00291         return;
00292     }
00293     *actual_size_ptr_ = elements_written_;
00294     finalized_ = true;
00295     detail::LogInfo("InMemoryOutputStream: Finalized " + id_ +
00296                     ". Elements: " + std::to_string(*actual_size_ptr_));
00297 }
```

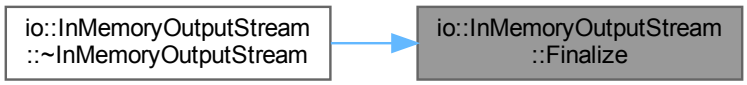
References io::InMemoryOutputStream< T >::actual_size_ptr_, io::InMemoryOutputStream< T >::elements_written_, io::InMemoryOutputStream< T >::finalized_, io::InMemoryOutputStream< T >::id_, and io::detail::LogInfo().

Referenced by io::InMemoryOutputStream< T >::~InMemoryOutputStream().

Here is the call graph for this function:



Here is the caller graph for this function:



8.13.3.2 GetId()

```
template<typename T >
StorageId io::InMemoryOutputStream< T >::GetId ( ) const  [override], [virtual]
```

Returns the stream identifier.

Returns

Stream StorageId

Implements io::IOutputStream< T >.

Definition at line 319 of file memory_stream.hpp.

```
00319                                           {
00320     return id_;
00321 }
```

References io::InMemoryOutputStream< T >::id_.

8.13.3.3 GetTotalBytesWritten()

```
template<typename T >
uint64_t io::InMemoryOutputStream< T >::GetTotalBytesWritten ( ) const  [override], [virtual]
```

Returns the total number of bytes that would be written if serialized.

Returns

Estimated total bytes for in-memory data

Implements io::IOutputStream< T >.

Definition at line 305 of file memory_stream.hpp.

```
00305                                                       {
00306     // For in-memory streams, calculate the serialized size of all elements
00307     // This includes the "header" (element count) that would be written to a file
00308     uint64_t total_bytes = sizeof(uint64_t);  // Header with element count
00309
00310     auto serializer = serialization::CreateSerializer<T>();
00311     for (size_t i = 0; i < elements_written_; ++i) {
00312         total_bytes += serializer->GetSerializedSize((*data_ptr_)[i]);
00313     }
00314
00315     return total_bytes;
00316 }
```

References io::InMemoryOutputStream< T >::data_ptr_, and io::InMemoryOutputStream< T >::elements_written_.

8.13.3.4 GetTotalElementsWritten()

```
template<typename T >
uint64_t io::InMemoryOutputStream< T >::GetTotalElementsWritten ( ) const  [override], [virtual]
```

Returns the total number of elements written.

Returns

Number of elements written to the stream

Implements io::IOutputStream< T >.

Definition at line 300 of file memory_stream.hpp.

```
00300                                                       {
00301     return elements_written_;
00302 }
```

References io::InMemoryOutputStream< T >::elements_written_.

8.13.3.5 Write() [1/2]

```
template<typename T >
void io::InMemoryOutputStream< T >::Write (
            const T & value )  [override], [virtual]
```

Writes an element to the stream.

Parameters

value	Element to write

Exceptions

std::logic_error	if the stream is finalized

Implements io::IOutputStream< T >.

Definition at line 271 of file memory_stream.hpp.

```
00271                                                 {
00272     if (finalized_) {
00273         throw std::logic_error("Write to finalized InMemoryOutputStream: " + id_);
00274     }
00275     data_ptr_->push_back(value);
00276     elements_written_++;
00277 }
```

References io::InMemoryOutputStream< T >::data_ptr_, io::InMemoryOutputStream< T >::elements_written_, io::InMemoryOutputStream< T >::finalized_, and io::InMemoryOutputStream< T >::id_.

8.13.3.6 Write() [2/2]

```
template<typename T >
void io::InMemoryOutputStream< T >::Write (
            T && value )  [override], [virtual]
```

Writes an rvalue element to the stream.

Parameters

value	Rvalue element to write

Exceptions

std::logic_error	if the stream is finalized

Implements io::IOutputStream< T >.

Definition at line 280 of file memory_stream.hpp.

```
00280                                                 {
00281     if (finalized_) {
00282         throw std::logic_error("Write to finalized InMemoryOutputStream: " + id_);
00283     }
00284     data_ptr_->push_back(std::move(value));
00285     elements_written_++;
00286 }
```

References io::InMemoryOutputStream< T >::data_ptr_, io::InMemoryOutputStream< T >::elements_written_, io::InMemoryOutputStream< T >::finalized_, and io::InMemoryOutputStream< T >::id_.

8.13.4 Member Data Documentation

8.13.4.1 actual_size_ptr_

```
template<typename T >
```

```
std::shared_ptr<uint64_t> io::InMemoryOutputStream< T >::actual_size_ptr_  [private]
```

Pointer to size ("header")

Definition at line 34 of file memory_stream.hpp.

Referenced by io::InMemoryOutputStream< T >::Finalize(), and io::InMemoryOutputStream< T >::InMemoryOutputStream().

8.13.4.2 data_ptr_

```
template<typename T >
std::shared_ptr<std::vector<T> > io::InMemoryOutputStream< T >::data_ptr_  [private]
```

Pointer to data.

Definition at line 33 of file memory_stream.hpp.

Referenced by io::InMemoryOutputStream< T >::GetTotalBytesWritten(), io::InMemoryOutputStream< T >::InMemoryOutputStream
io::InMemoryOutputStream< T >::Write(), and io::InMemoryOutputStream< T >::Write().

8.13.4.3 elements_written_

```
template<typename T >
uint64_t io::InMemoryOutputStream< T >::elements_written_ = 0  [private]
```

Number of elements written.

Definition at line 35 of file memory_stream.hpp.

Referenced by io::InMemoryOutputStream< T >::Finalize(), io::InMemoryOutputStream< T >::GetTotalBytesWritten(), io::InMemoryOutputStream< T >::GetTotalElementsWritten(), io::InMemoryOutputStream< T >::Write(), and io::InMemoryOutputStream< T >::Write().

8.13.4.4 finalized_

```
template<typename T >
bool io::InMemoryOutputStream< T >::finalized_ = false  [private]
```

Finalization flag.

Definition at line 36 of file memory_stream.hpp.

Referenced by io::InMemoryOutputStream< T >::Finalize(), io::InMemoryOutputStream< T >::Write(), and io::InMemoryOutputStream< T >::Write().

8.13.4.5 id_

```
template<typename T >
StorageId io::InMemoryOutputStream< T >::id_  [private]
```

Stream identifier.

Definition at line 32 of file memory_stream.hpp.

Referenced by io::InMemoryOutputStream< T >::Finalize(), io::InMemoryOutputStream< T >::GetId(), io::InMemoryOutputStream< io::InMemoryOutputStream< T >::Write(), and io::InMemoryOutputStream< T >::Write().

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

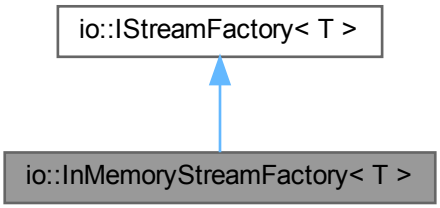
- io/include/memory_stream.hpp

8.14 io::InMemoryStreamFactory< T > Class Template Reference

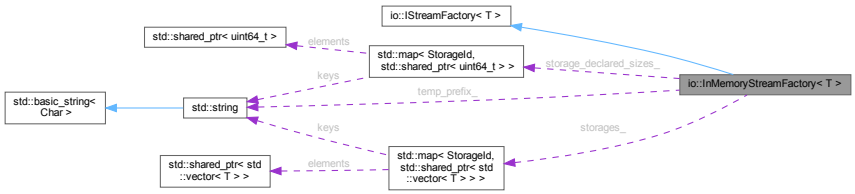
Factory for in-memory streams.

```
#include <memory_stream.hpp>
```

Inheritance diagram for io::InMemoryStreamFactory< T >:



Collaboration diagram for io::InMemoryStreamFactory< T >:



Public Member Functions

- InMemoryStreamFactory ()=default

  Constructor for in-memory stream factory.
- std::unique_ptr< IInputStream< T > > CreateInputStream (const StorageId &id, uint64_t buffer_capacity_elements) override

  Creates an input stream for reading from memory.
- std::unique_ptr< IOutputStream< T > > CreateOutputStream (const StorageId &id, uint64_t buffer_capacity_elements) override

  Creates an output stream for writing to memory.
- std::unique_ptr< IOutputStream< T > > CreateTempOutputStream (StorageId &out_temp_id, uint64_t buffer_capacity_elements) override

  Creates an output stream for writing to temporary storage.
- void DeleteStorage (const StorageId &id) override

  Deletes a storage from memory.
- void MakeStoragePermanent (const StorageId &temp_id, const StorageId &final_id) override

  Makes a temporary storage permanent.
- bool StorageExists (const StorageId &id) const override

  Checks if a storage exists.
- StorageId GetTempStorageContextId () const override

  Returns the temporary storage context.
- std::shared_ptr< const std::vector< T > > GetStorageData (const StorageId &id) const

  Returns storage data (for tests)
- uint64_t GetStorageDeclaredSize (const StorageId &id) const

  Returns declared storage size (for tests)

Public Member Functions inherited from io::IStreamFactory< T >

- virtual ~IStreamFactory ()=default

Private Attributes

- std::map< StorageId, std::shared_ptr< std::vector< T > > > storages_

  Data storages.
- std::map< StorageId, std::shared_ptr< uint64_t > > storage_declared_sizes_

  Declared storage sizes.
- uint64_t temp_id_counter_ = 0

  Temporary ID counter.
- const std::string temp_prefix_ = "in_memory_temp_run_"

  Temporary ID prefix.

8.14.1 Detailed Description

template<typename T>
class io::InMemoryStreamFactory< T >

Factory for in-memory streams.

Implements the IStreamFactory interface for creating streams that work with data in memory.

Template Parameters

T	Type of elements in the streams

Definition at line 170 of file memory_stream.hpp.

8.14.2 Constructor & Destructor Documentation

8.14.2.1 InMemoryStreamFactory()

```
template<typename T >
io::InMemoryStreamFactory< T >::InMemoryStreamFactory ( )  [default]
```

Constructor for in-memory stream factory.

8.14.3 Member Function Documentation

8.14.3.1 CreateInputStream()

```
template<typename T >
std::unique_ptr< IInputStream< T > > io::InMemoryStreamFactory< T >::CreateInputStream (
            const StorageId & id,
            uint64_t buffer_capacity_elements )  [override], [virtual]
```

Creates an input stream for reading from memory.

Parameters

id	Storage identifier
buffer_capacity_elements	Buffer capacity (not used)

Returns

Unique pointer to InMemoryInputStream

Exceptions

std::runtime_error	if storage is not found

Implements io::IStreamFactory< T >.

Definition at line 390 of file memory_stream.hpp.

```
00391                                                                {
00392     auto it_data = storages_.find(id);
00393     auto it_size = storage_declared_sizes_.find(id);
00394     if (it_data == storages_.end() || it_size == storage_declared_sizes_.end()) {
00395         throw std::runtime_error("InMemoryStreamFactory: Storage ID not found for input: " + id);
00396     }
00397     return std::make_unique<InMemoryInputStream<T»(
00398         id, it_data->second, *(it_size->second), buffer_capacity_elements);
```

```
00399 }
```

References io::InMemoryStreamFactory< T >::storage_declared_sizes_, and io::InMemoryStreamFactory< T >::storages_.

8.14.3.2 CreateOutputStream()

```
template<typename T >
std::unique_ptr< IOutputStream< T > > io::InMemoryStreamFactory< T >::CreateOutputStream (
            const StorageId & id,
            uint64_t buffer_capacity_elements )  [override], [virtual]
```

Creates an output stream for writing to memory.

Parameters

id	Storage identifier
buffer_capacity_elements	Buffer capacity (not used)

Returns

Unique pointer to InMemoryOutputStream

Implements io::IStreamFactory< T >.

Definition at line 402 of file memory_stream.hpp.

```
00403                                                                    {
00404     auto data_ptr = std::make_shared<std::vector<T>>();
00405     auto size_ptr = std::make_shared<uint64_t>(0);
00406     storages_[id] = data_ptr;
00407     storage_declared_sizes_[id] = size_ptr;
00408     return std::make_unique<InMemoryOutputStream<T>>(
00409         id, data_ptr, size_ptr, buffer_capacity_elements);
00410 }
```

References io::InMemoryStreamFactory< T >::storage_declared_sizes_, and io::InMemoryStreamFactory< T >::storages_.

8.14.3.3 CreateTempOutputStream()

```
template<typename T >
std::unique_ptr< IOutputStream< T > > io::InMemoryStreamFactory< T >::CreateTempOutputStream
(
            StorageId & out_temp_id,
            uint64_t buffer_capacity_elements )  [override], [virtual]
```

Creates an output stream for writing to temporary storage.

Parameters

out_temp_id	Output parameter for temporary storage ID
buffer_capacity_elements	Buffer capacity (not used)

Returns

Unique pointer to InMemoryOutputStream

Implements io::IStreamFactory< T >.

Definition at line 413 of file memory_stream.hpp.

```
00414                                                                    {
00415     out_temp_id = temp_prefix_ + std::to_string(temp_id_counter_++);
00416     auto data_ptr = std::make_shared<std::vector<T»();
00417     auto size_ptr = std::make_shared<uint64_t>(0);
00418     storages_[out_temp_id] = data_ptr;
00419     storage_declared_sizes_[out_temp_id] = size_ptr;
00420     return std::make_unique<InMemoryOutputStream<T»(
00421         out_temp_id, data_ptr, size_ptr, buffer_capacity_elements);
00422 }
```

References io::InMemoryStreamFactory< T >::storage_declared_sizes_, io::InMemoryStreamFactory< T >::storages_, io::InMemoryStreamFactory< T >::temp_id_counter_, and io::InMemoryStreamFactory< T >::temp_prefix_.

8.14.3.4 DeleteStorage()

```
template<typename T >
void io::InMemoryStreamFactory< T >::DeleteStorage (
            const StorageId & id )  [override], [virtual]
```

Deletes a storage from memory.

Parameters

id	Storage identifier to delete

Implements io::IStreamFactory< T >.

Definition at line 425 of file memory_stream.hpp.

```
00425                                                                    {
00426     storages_.erase(id);
00427     storage_declared_sizes_.erase(id);
00428     detail::LogInfo("InMemoryStreamFactory: Deleted storage " + id);
00429 }
```

References io::detail::LogInfo(), io::InMemoryStreamFactory< T >::storage_declared_sizes_, and io::InMemoryStreamFactory< T >

Here is the call graph for this function:

8.14.3.5 GetStorageData()

```
template<typename T >
std::shared_ptr< const std::vector< T > > io::InMemoryStreamFactory< T >::GetStorageData (
            const StorageId & id ) const
```

Returns storage data (for tests)

Parameters

id	Storage identifier

Returns

Pointer to data or nullptr if not found

Definition at line 462 of file memory_stream.hpp.

```
00463                                                 {
00464     auto it = storages_.find(id);
00465     return (it != storages_.end()) ? it->second : nullptr;
00466 }
```

References io::InMemoryStreamFactory< T >::storages_.

8.14.3.6 GetStorageDeclaredSize()

```
template<typename T >
uint64_t io::InMemoryStreamFactory< T >::GetStorageDeclaredSize (
            const StorageId & id ) const
```

Returns declared storage size (for tests)

Parameters

id	Storage identifier

Returns

Declared size or 0 if not found

Definition at line 469 of file memory_stream.hpp.

```
00469                                                                                 {
00470     auto it = storage_declared_sizes_.find(id);
00471     return (it != storage_declared_sizes_.end()) ? *(it->second) : 0;
00472 }
```

References io::InMemoryStreamFactory< T >::storage_declared_sizes_.

8.14.3.7 GetTempStorageContextId()

```
template<typename T >
StorageId io::InMemoryStreamFactory< T >::GetTempStorageContextId ( ) const  [override], [virtual]
```

Returns the temporary storage context.

Returns

Temporary ID prefix

Implements io::IStreamFactory< T >.

Definition at line 457 of file memory_stream.hpp.

```
00457                                                                 {
00458     return temp_prefix_;
00459 }
```

References io::InMemoryStreamFactory< T >::temp_prefix_.

8.14.3.8 MakeStoragePermanent()

```
template<typename T >
void io::InMemoryStreamFactory< T >::MakeStoragePermanent (
            const StorageId & temp_id,
            const StorageId & final_id )  [override], [virtual]
```

Makes a temporary storage permanent.

Parameters

temp_id	Temporary storage identifier
final_id	Final storage identifier

Exceptions

std::runtime_error	if temporary storage is not found

Implements io::IStreamFactory< T >.

Definition at line 432 of file memory_stream.hpp.

```
00433                                                         {
00434     if (temp_id == final_id) {
00435         return;
00436     }
00437
00438     auto it_temp_data = storages_.find(temp_id);
00439     auto it_temp_size = storage_declared_sizes_.find(temp_id);
00440     if (it_temp_data == storages_.end() || it_temp_size == storage_declared_sizes_.end()) {
00441         throw std::runtime_error("InMemoryStreamFactory: Temp ID not found: " + temp_id);
00442     }
00443     storages_[final_id] = it_temp_data->second;
00444     storage_declared_sizes_[final_id] = it_temp_size->second;
00445
00446     storages_.erase(it_temp_data);
00447     storage_declared_sizes_.erase(it_temp_size);
00448     detail::LogInfo("InMemoryStreamFactory: Made " + temp_id + " permanent as " + final_id);
00449 }
```

References io::detail::LogInfo(), io::InMemoryStreamFactory< T >::storage_declared_sizes_, and io::InMemoryStreamFactory< T >

Here is the call graph for this function:

8.14.3.9 StorageExists()

```
template<typename T >
bool io::InMemoryStreamFactory< T >::StorageExists (
            const StorageId & id ) const  [override], [virtual]
```

Checks if a storage exists.

Parameters

id	Storage identifier

Returns

true if the storage exists

Implements io::IStreamFactory< T >.

Definition at line 452 of file memory_stream.hpp.

```
00452                                                                    {
00453     return storages_.count(id);
00454 }
```

References io::InMemoryStreamFactory< T >::storages_.

8.14.4 Member Data Documentation

8.14.4.1 storage_declared_sizes_

```
template<typename T >
std::map<StorageId, std::shared_ptr<uint64_t> > io::InMemoryStreamFactory< T >::storage_↩
declared_sizes_  [private]
```

Declared storage sizes.

Definition at line 174 of file memory_stream.hpp.

Referenced by io::InMemoryStreamFactory< T >::CreateInputStream(), io::InMemoryStreamFactory< T >::CreateOutputStream(), io::InMemoryStreamFactory< T >::CreateTempOutputStream(), io::InMemoryStreamFactory< T >::DeleteStorage(), io::InMemoryStreamFactory< T >::GetStorageDeclaredSize(), and io::InMemoryStreamFactory< T >::MakeStoragePermanent().

8.14.4.2 storages_

```
template<typename T >
std::map<StorageId, std::shared_ptr<std::vector<T> > > io::InMemoryStreamFactory< T >↩
::storages_ [private]
```

Data storages.

Definition at line 172 of file memory_stream.hpp.

Referenced by io::InMemoryStreamFactory< T >::CreateInputStream(), io::InMemoryStreamFactory< T >::CreateOutputStream(), io::InMemoryStreamFactory< T >::CreateTempOutputStream(), io::InMemoryStreamFactory< T >::DeleteStorage(), io::InMemoryStreamFactory< T >::GetStorageData(), io::InMemoryStreamFactory< T >::MakeStoragePermanent(), and io::InMemoryStreamFactory< T >::StorageExists().

8.14.4.3 temp_id_counter_

```
template<typename T >
uint64_t io::InMemoryStreamFactory< T >::temp_id_counter_ = 0 [private]
```

Temporary ID counter.

Definition at line 175 of file memory_stream.hpp.

Referenced by io::InMemoryStreamFactory< T >::CreateTempOutputStream().

8.14.4.4 temp_prefix_

```
template<typename T >
const std::string io::InMemoryStreamFactory< T >::temp_prefix_ = "in_memory_temp_run_" [private]
```

Temporary ID prefix.

Definition at line 176 of file memory_stream.hpp.

Referenced by io::InMemoryStreamFactory< T >::CreateTempOutputStream(), and io::InMemoryStreamFactory< T >::GetTempSto

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

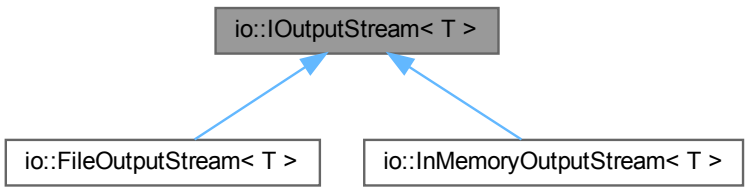
- io/include/memory_stream.hpp

8.15 io::IOutputStream< T > Class Template Reference

Interface for output stream of elements of type T.

`#include <interfaces.hpp>`

Inheritance diagram for io::IOutputStream< T >:



Public Member Functions

- virtual ~IOutputStream ()=default
- virtual void Write (const T &value)=0

  Writes an element to the stream.
- virtual void Write (T &&value)=0

  Writes an element to the stream using move semantics.
- virtual void Finalize ()=0

  Finalizes the stream, writing all buffered data.
- virtual uint64_t GetTotalElementsWritten () const =0

  Returns the total number of elements written.
- virtual uint64_t GetTotalBytesWritten () const =0

  Returns the total number of bytes written to the storage.
- virtual StorageId GetId () const =0

  Returns the identifier of the storage.

8.15.1 Detailed Description

template<typename T>
class io::IOutputStream< T >

Interface for output stream of elements of type T.

Template Parameters

T	type of elements in stream

Definition at line 64 of file interfaces.hpp.

8.15.2 Constructor & Destructor Documentation

8.15.2.1 ~IOutputStream()

```
template<typename T >
virtual io::IOutputStream< T >::~IOutputStream ( )  [virtual], [default]
```

8.15.3 Member Function Documentation

8.15.3.1 Finalize()

```
template<typename T >
virtual void io::IOutputStream< T >::Finalize ( )  [pure virtual]
```

Finalizes the stream, writing all buffered data.

Implemented in io::FileOutputStream< T >, and io::InMemoryOutputStream< T >.

Referenced by main().

Here is the caller graph for this function:

8.15.3.2 GetId()

```
template<typename T >
virtual StorageId io::IOutputStream< T >::GetId ( ) const  [pure virtual]
```

Returns the identifier of the storage.

Returns

The StorageId of the storage associated with the stream

Implemented in io::FileOutputStream< T >, and io::InMemoryOutputStream< T >.

8.15.3.3 GetTotalBytesWritten()

```
template<typename T >
virtual uint64_t io::IOutputStream< T >::GetTotalBytesWritten ( ) const  [pure virtual]
```

Returns the total number of bytes written to the storage.

This method returns the actual size in bytes of the serialized data written to the storage. For complex types (e.g., types with dynamic memory like std::string), this accounts for the actual serialized size, not just sizeof(T) * element_count.

This is particularly useful for:

- Tracking actual disk usage for file-based streams
- Generating data files of specific target sizes in benchmarks
- Monitoring memory usage for in-memory streams
- Estimating storage requirements for complex data types

Returns

The total number of bytes written (including any headers/metadata)

Note

For POD types, this typically equals: sizeof(header) + sizeof(T) * GetTotalElementsWritten()

For complex types (std::string, custom types), this reflects the actual serialized size

This value includes file headers (e.g., element count header in FileOutputStream)

Example:

```
auto output_stream = factory->CreateOutputStream("data.bin", 1024);
for (const auto& person : people) {
    output_stream->Write(person);
}
uint64_t total_bytes = output_stream->GetTotalBytesWritten();
// total_bytes will be the actual file size including all string data
```

Implemented in io::FileOutputStream< T >, and io::InMemoryOutputStream< T >.

8.15.3.4 GetTotalElementsWritten()

```
template<typename T >
virtual uint64_t io::IOutputStream< T >::GetTotalElementsWritten ( ) const  [pure virtual]
```

Returns the total number of elements written.

Returns

The number of elements written to the stream

Implemented in io::FileOutputStream< T >, and io::InMemoryOutputStream< T >.

8.15.3.5 Write() [1/2]

```
template<typename T >
virtual void io::IOutputStream< T >::Write (
            const T & value )  [pure virtual]
```

Writes an element to the stream.

Parameters

value	The element to write

Exceptions

std::logic_error	if the stream has been finalized

Implemented in io::FileOutputStream< T >, and io::InMemoryOutputStream< T >.

8.15.3.6 Write() [2/2]

```
template<typename T >
```

```
virtual void io::IOutputStream< T >::Write (
            T && value )  [pure virtual]
```

Writes an element to the stream using move semantics.

Parameters

value	The element to write (moved)

Exceptions

std::logic_error	if the stream has been finalized

Implemented in io::FileOutputStream< T >, and io::InMemoryOutputStream< T >.

Referenced by main().

Here is the caller graph for this function:

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

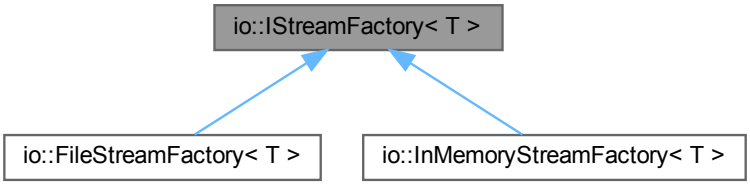
- io/include/interfaces.hpp

8.16 io::IStreamFactory< T > Class Template Reference

Interface for a stream factory.

`#include <interfaces.hpp>`

Inheritance diagram for io::IStreamFactory< T >:



Public Member Functions

- virtual ~IStreamFactory ()=default
- virtual std::unique_ptr< IInputStream< T > > CreateInputStream (const StorageId &id, uint64_t buffer_↩ capacity_elements)=0

  Creates an input stream to read from an existing storage.
- virtual std::unique_ptr< IOutputStream< T > > CreateOutputStream (const StorageId &id, uint64_t buffer↩ _capacity_elements)=0

  Creates an output stream to write to a new or existing storage.
- virtual std::unique_ptr< IOutputStream< T > > CreateTempOutputStream (StorageId &out_temp_id, uint64_t buffer_capacity_elements)=0

  Creates an output stream to write to a new temporary storage.
- virtual void DeleteStorage (const StorageId &id)=0

  Deletes a storage.
- virtual void MakeStoragePermanent (const StorageId &temp_id, const StorageId &final_id)=0

  Makes a temporary storage permanent under a new (or the same) ID.
- virtual bool StorageExists (const StorageId &id) const =0

  Checks if a storage with the given ID exists.
- virtual StorageId GetTempStorageContextId () const =0

  Gets an identifier representing the base path or context for temporary storages This is used to prevent conflicts (e.g., writing output to the temporary directory itself).

8.16.1 Detailed Description

template<typename T>
class io::IStreamFactory< T >

Interface for a stream factory.

Manages creation, deletion, and finalization of data streams, which can be either file-based or in-memory.

Template Parameters

T	The type of elements in the streams

Definition at line 141 of file interfaces.hpp.

8.16.2 Constructor & Destructor Documentation

8.16.2.1 ~IStreamFactory()

```
template<typename T >
virtual io::IStreamFactory< T >::~IStreamFactory ( )  [virtual], [default]
```

8.16.3 Member Function Documentation

8.16.3.1 CreateInputStream()

```
template<typename T >
virtual std::unique_ptr< IInputStream< T > > io::IStreamFactory< T >::CreateInputStream (
```

```
            const StorageId & id,
            uint64_t buffer_capacity_elements )  [pure virtual]
```

Creates an input stream to read from an existing storage.

Parameters

id	The identifier of the storage to read from
buffer_capacity_elements	The capacity of the stream's internal buffer

Returns

A unique pointer to an IInputStream

Exceptions

std::runtime_error	if the storage cannot be opened or does not exist

Implemented in io::FileStreamFactory< T >, and io::InMemoryStreamFactory< T >.

8.16.3.2 CreateOutputStream()

```
template<typename T >
virtual std::unique_ptr< IOutputStream< T > > io::IStreamFactory< T >::CreateOutputStream (
            const StorageId & id,
            uint64_t buffer_capacity_elements )  [pure virtual]
```

Creates an output stream to write to a new or existing storage.

If the storage already exists, its content will be overwritten.

Parameters

id	The identifier of the storage to write to
buffer_capacity_elements	The capacity of the stream's internal buffer

Returns

A unique pointer to an IOutputStream

Exceptions

std::runtime_error	if the storage cannot be created or opened for writing

Implemented in io::FileStreamFactory< T >, and io::InMemoryStreamFactory< T >.

8.16.3.3 CreateTempOutputStream()

```
template<typename T >
virtual std::unique_ptr< IOutputStream< T > > io::IStreamFactory< T >::CreateTempOutput↩
Stream (
            StorageId & out_temp_id,
            uint64_t buffer_capacity_elements )  [pure virtual]
```

Creates an output stream to write to a new temporary storage.

The factory assigns a unique ID to this temporary storage.

Parameters

out_temp_id	Output parameter that will receive the ID of the created temporary storage
buffer_capacity_elements	The capacity of the stream's internal buffer

Returns

A unique pointer to an IOutputStream for the temporary storage

Implemented in io::FileStreamFactory< T >, and io::InMemoryStreamFactory< T >.

8.16.3.4 DeleteStorage()

```
template<typename T >
virtual void io::IStreamFactory< T >::DeleteStorage (
            const StorageId & id )  [pure virtual]
```

Deletes a storage.

Parameters

id	The identifier of the storage to delete

Implemented in io::FileStreamFactory< T >, and io::InMemoryStreamFactory< T >.

8.16.3.5 GetTempStorageContextId()

```
template<typename T >
virtual StorageId io::IStreamFactory< T >::GetTempStorageContextId ( ) const  [pure virtual]
```

Gets an identifier representing the base path or context for temporary storages This is used to prevent conflicts (e.g., writing output to the temporary directory itself).

Returns

A StorageId representing the context for temporary storages

Implemented in io::FileStreamFactory< T >, and io::InMemoryStreamFactory< T >.

8.16.3.6 MakeStoragePermanent()

```
template<typename T >
virtual void io::IStreamFactory< T >::MakeStoragePermanent (
            const StorageId & temp_id,
            const StorageId & final_id )  [pure virtual]
```

Makes a temporary storage permanent under a new (or the same) ID.

This typically involves renaming or copying. The original temporary storage (temp_id) is usually deleted or becomes inaccessible after this operation.

Parameters

temp↩ _id	The identifier of the source temporary storage
final_id	The identifier for the destination permanent storage

Exceptions

std::runtime_error	if the operation fails

Implemented in io::FileStreamFactory< T >, and io::InMemoryStreamFactory< T >.

8.16.3.7 StorageExists()

```
template<typename T >
virtual bool io::IStreamFactory< T >::StorageExists (
            const StorageId & id ) const  [pure virtual]
```

Checks if a storage with the given ID exists.

Parameters

id	The identifier of the storage to check

Returns

true if the storage exists, false otherwise

Implemented in io::FileStreamFactory< T >, and io::InMemoryStreamFactory< T >.

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

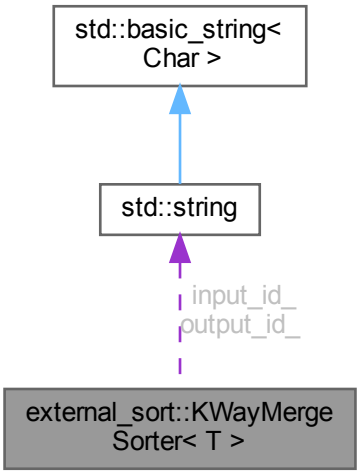
- io/include/interfaces.hpp

8.17 external_sort::KWayMergeSorter< T > Class Template Reference

Template class for performing k-way external merge sort.

`#include <k_way_merge_sorter.hpp>`

Collaboration diagram for external_sort::KWayMergeSorter< T >:



Classes

- struct MergeSourceComparator

  Comparator for merge sources.

Public Member Functions

- KWayMergeSorter (io::IStreamFactory< T > &factory, io::StorageId input_id, io::StorageId output_id, uint64_t mem_bytes, uint64_t k_degree, uint64_t io_buf_elems, bool sort_ascending=true)

  Constructor for k-way merge sorter.
- void Sort ()

  Performs sorting.

Private Member Functions

- std::vector< io::StorageId > CreateInitialRuns ()

  Creates initial sorted runs from input stream.
- void MergeGroupOfRuns (const std::vector< io::StorageId > &group_run_ids, const io::StorageId &output↩_run_id)

  Merges a group of runs into one.

Private Attributes

- io::IStreamFactory< T > & stream_factory_

  Reference to stream factory.
- io::StorageId input_id_

  Input storage identifier.
- io::StorageId output_id_

  Output storage identifier.
- uint64_t memory_for_runs_bytes_

  Memory for creating initial runs (bytes)
- uint64_t k_way_degree_

  Degree of k-way merging.
- uint64_t file_io_buffer_elements_

  Size of I/O buffer in elements.
- bool ascending_

  Ascending sort flag.

8.17.1 Detailed Description

template<typename T>
class external_sort::KWayMergeSorter< T >

Template class for performing k-way external merge sort.

The class is abstracted from specific storage implementation and works through IStreamFactory, io::IInputStream and io::IOutputStream interfaces.

Template Parameters

T	Type of elements to sort

Definition at line 56 of file k_way_merge_sorter.hpp.

8.17.2 Constructor & Destructor Documentation

8.17.2.1 KWayMergeSorter()

```
template<typename T >
external_sort::KWayMergeSorter< T >::KWayMergeSorter (
            io::IStreamFactory< T > & factory,
            io::StorageId input_id,
            io::StorageId output_id,
            uint64_t mem_bytes,
            uint64_t k_degree,
            uint64_t io_buf_elems,
            bool sort_ascending = true )
```

Constructor for k-way merge sorter.

Parameters

factory	Reference to stream factory
input_id	Input storage identifier
output_id	Output storage identifier
mem_bytes	Memory for creating runs in bytes
k_degree	Degree of k-way merging (minimum 2)
io_buf_elems	Size of I/O buffer in elements
sort_ascending	Sort direction (default ascending)

Exceptions

std::invalid_argument	if k_degree < 2
std::runtime_error	if output_id is in temporary storage context

Definition at line 263 of file k_way_merge_sorter.hpp.

```
00266     : stream_factory_(factory),
00267       input_id_(std::move(input_id)),
00268       output_id_(std::move(output_id)),
00269       memory_for_runs_bytes_(mem_bytes),
00270       k_way_degree_(k_degree),
00271       file_io_buffer_elements_(io_buf_elems),
00272       ascending_(sort_ascending) {
00273     if (k_way_degree_ < 2) {
00274         throw std::invalid_argument("KWayMergeSorter: k_way_degree must be at least 2.");
00275     }
00276     io::StorageId temp_context_id = stream_factory_.GetTempStorageContextId();
00277     if (!temp_context_id.empty() && output_id_.rfind(temp_context_id, 0) == 0 &&
00278         output_id_.length() > temp_context_id.length()) {
00279         throw std::runtime_error("KWayMergeSorter: Output storage ID '" + output_id_ +
00280                                  "' seems to be inside the temporary storage context '" +
00281                                  temp_context_id + "'.");
00282     }
00283 }
```

References external_sort::KWayMergeSorter< T >::ascending_, external_sort::KWayMergeSorter< T >::file_io_buffer_elements_, external_sort::KWayMergeSorter< T >::input_id_, external_sort::KWayMergeSorter< T >::k_way_degree_, external_sort::KWayMergeSorter< T >::memory_for_runs_bytes_, external_sort::KWayMergeSorter< T >::output_id_, and external_sort::KWayMergeSorter< T >::stream_factory_.

Referenced by main().

Here is the caller graph for this function:

8.17.3 Member Function Documentation

8.17.3.1 CreateInitialRuns()

```
template<typename T >
std::vector< io::StorageId > external_sort::KWayMergeSorter< T >::CreateInitialRuns ( )  [private]
```

Creates initial sorted runs from input stream.

Returns

Vector of created run identifiers

Note

Performance tip: When sorting non-trivial or move-only types prefer using stream implementations that support TakeValue() and rvalue Write(T&&). The sorter uses these methods internally (when available) to minimize copies by moving elements between streams and temporary buffers.

Definition at line 141 of file k_way_merge_sorter.hpp.

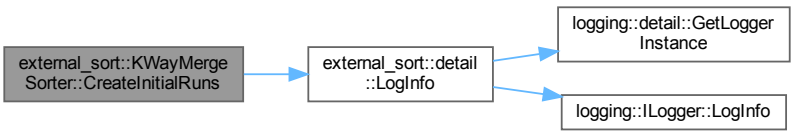
```
00141                                                                 {
00142     detail::LogInfo("»> CreateInitialRuns started for input: '" + input_id_ +
00143                     "' with memory limit: " + std::to_string(memory_for_runs_bytes_ / (1024.0 *
      1024.0)) + " MB");
00144     detail::LogInfo("    sizeof(T) = " + std::to_string(sizeof(T)) + " bytes");
00145
00146     std::unique_ptr<io::IInputStream<T» input_stream =
00147         stream_factory_.CreateInputStream(input_id_, file_io_buffer_elements_);
00148
00149     if (input_stream->IsEmptyOriginalStorage()) {
00150         detail::LogInfo("KWayMergeSorter: Input storage is empty. No runs created.");
00151         return {};
00152     }
00153
00154     auto serializer = serialization::CreateSerializer<T>();
00155     std::vector<io::StorageId> run_ids;
00156     std::vector<T> run_buffer;
00157     int run_counter = 1;
00158
00159     while (!input_stream->IsExhausted()) {
00160         run_buffer.clear();
00161         uint64_t current_run_mem_usage = 0;
00162         int element_counter = 0;
00163
00164         detail::LogInfo("--- Starting Run #" + std::to_string(run_counter++) + " ---");
00165
00166         while (!input_stream->IsExhausted()) {
00167             const T& current_element = input_stream->Value();
00168
00169             uint64_t serialized_size = serializer->GetSerializedSize(current_element);
00170             uint64_t estimated_element_footprint;
00171
00172             if constexpr (std::is_trivial_v<T> && std::is_standard_layout_v<T>) {
00173                 estimated_element_footprint = sizeof(T);
00174             } else {
00175                 estimated_element_footprint = serialized_size + sizeof(T);
00176             }
00177
00178             detail::LogInfo("[Loop " + std::to_string(element_counter++) + "]: "
00179                             "current_mem=" + std::to_string(current_run_mem_usage) +
00180                             ", next_elem_footprint=" + std::to_string(estimated_element_footprint) +
00181                             " (ser_size=" + std::to_string(serialized_size) + ")" +
00182                             ", limit=" + std::to_string(memory_for_runs_bytes_) +
00183                             ", vec_size=" + std::to_string(run_buffer.size()) +
00184                             ", vec_cap=" + std::to_string(run_buffer.capacity()));
00185
00186             if (run_buffer.empty()) {
00187                 if (estimated_element_footprint > memory_for_runs_bytes_) {
00188                     throw std::runtime_error("KWayMergeSorter: Memory limit is too small for a single
      element.");
00189                 }
00190             }
00191             else if (current_run_mem_usage + estimated_element_footprint > memory_for_runs_bytes_) {
00192                 detail::LogInfo("    Limit reached. Breaking loop to finalize run.");
00193                 break;
00194             }
00195
00196             current_run_mem_usage += estimated_element_footprint;
00197             run_buffer.push_back(input_stream->TakeValue());
00198             input_stream->Advance();
00199         }
00200
00201         if (!run_buffer.empty()) {
00202             detail::LogInfo("    Sorting " + std::to_string(run_buffer.size()) + " elements...");
00203             if (ascending_) {
00204                 std::sort(run_buffer.begin(), run_buffer.end());
00205             } else {
00206                 std::sort(run_buffer.begin(), run_buffer.end(), std::greater<T>());
00207             }
00208
```

```
00209             io::StorageId run_id;
00210             std::unique_ptr<io::IOutputStream<T» out_run =
00211                 stream_factory_.CreateTempOutputStream(run_id, file_io_buffer_elements_);
00212             for (T& val : run_buffer) {
00213                 out_run->Write(std::move(val));
00214             }
00215             out_run->Finalize();
00216             run_ids.push_back(std::move(run_id));
00217             detail::LogInfo("    Run created: " + run_id + " with " +
00218                             std::to_string(out_run->GetTotalElementsWritten()) +
00219                             " elements. Estimated mem usage: " + std::to_string(current_run_mem_usage)
      + " bytes.");
00220         }
00221     }
00222     return run_ids;
00223 }
```

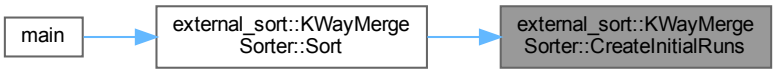
References external_sort::KWayMergeSorter< T >::ascending_, external_sort::KWayMergeSorter< T >::file_io_buffer_elements_, external_sort::KWayMergeSorter< T >::input_id_, external_sort::detail::LogInfo(), external_sort::KWayMergeSorter< T >::memory_ and external_sort::KWayMergeSorter< T >::stream_factory_.

Referenced by external_sort::KWayMergeSorter< T >::Sort().

Here is the call graph for this function:



Here is the caller graph for this function:



8.17.3.2 MergeGroupOfRuns()

```
template<typename T >
void external_sort::KWayMergeSorter< T >::MergeGroupOfRuns (
            const std::vector< io::StorageId > & group_run_ids,
            const io::StorageId & output_run_id )  [private]
```

Merges a group of runs into one.

Parameters

group_run_ids	Identifiers of runs to merge
output_run_id	Output run identifier

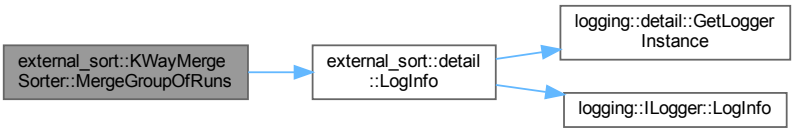
Definition at line 226 of file k_way_merge_sorter.hpp.

```
00227                                                                                {
00228     detail::LogInfo("KWayMergeSorter: Merging " + std::to_string(group_run_ids.size()) +
00229                     " runs into " + output_run_id);
00230
00231     std::priority_queue<MergeSource<T>, std::vector<MergeSource<T», MergeSourceComparator> pq{
00232         MergeSourceComparator(ascending_)};
00233     std::vector<std::unique_ptr<io::IInputStream<T»> input_streams_store;
00234     input_streams_store.reserve(group_run_ids.size());
00235
00236     for (const auto& run_id : group_run_ids) {
00237         auto stream_ptr = stream_factory_.CreateInputStream(run_id, file_io_buffer_elements_);
00238         if (!stream_ptr->IsExhausted()) {
00239             pq.push({stream_ptr.get()});
00240         }
00241         input_streams_store.push_back(std::move(stream_ptr));
00242     }
00243
00244     std::unique_ptr<io::IOutputStream<T» output_stream =
00245         stream_factory_.CreateOutputStream(output_run_id, file_io_buffer_elements_);
00246
00247     while (!pq.empty()) {
00248         MergeSource<T> current_source = pq.top();
00249         pq.pop();
00250         T val = current_source.stream->TakeValue();
00251         output_stream->Write(std::move(val));
00252         current_source.stream->Advance();
00253         if (!current_source.stream->IsExhausted()) {
00254             pq.push({current_source.stream});
00255         }
00256     }
00257     output_stream->Finalize();
00258     detail::LogInfo("KWayMergeSorter: Merged group into " + output_run_id + " with " +
00259                     std::to_string(output_stream->GetTotalElementsWritten()) + " elements.");
00260 }
```

References external_sort::KWayMergeSorter< T >::ascending_, external_sort::KWayMergeSorter< T >::file_io_buffer_elements_, external_sort::detail::LogInfo(), and external_sort::KWayMergeSorter< T >::stream_factory_.

Referenced by external_sort::KWayMergeSorter< T >::Sort().

Here is the call graph for this function:



Here is the caller graph for this function:

8.17.3.3 Sort()

```
template<typename T >
void external_sort::KWayMergeSorter< T >::Sort ( )
```

Performs sorting.

Main method that performs complete k-way external sorting:

1. Creates initial sorted runs
2. Repeatedly merges runs until one remains
3. Makes final run permanent with output_id name

Definition at line 286 of file k_way_merge_sorter.hpp.

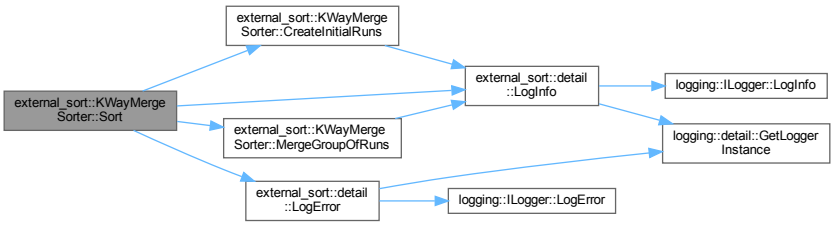
```
00286                                {
00287     std::vector<io::StorageId> current_run_ids = CreateInitialRuns();
00288
00289     if (current_run_ids.empty()) {
00290         detail::LogInfo("KWayMergeSorter: No initial runs. Creating empty output " + output_id_);
00291         auto empty_out = stream_factory_.CreateOutputStream(output_id_, file_io_buffer_elements_);
00292         empty_out->Finalize();
00293         return;
00294     }
00295
00296     std::vector<io::StorageId> runs_to_delete_this_pass;
00297     while (current_run_ids.size() > 1) {
00298         std::vector<io::StorageId> next_pass_run_ids;
00299         runs_to_delete_this_pass.clear();
00300         detail::LogInfo("KWayMergeSorter: Merge pass with " +
00301                         std::to_string(current_run_ids.size()) + " runs.");
00302
00303         for (uint64_t i = 0; i < current_run_ids.size(); i += k_way_degree_) {
00304             std::vector<io::StorageId> group_to_merge;
00305             for (uint64_t j = 0; j < k_way_degree_ && (i + j) < current_run_ids.size(); ++j) {
00306                 group_to_merge.push_back(current_run_ids[i + j]);
00307             }
00308             if (group_to_merge.empty()) {
00309                 continue;
00310             }
00311
00312             io::StorageId merged_run_id;
00313             bool is_final_merge_to_output = (current_run_ids.size() <= k_way_degree_) && (i == 0);
00314
00315             if (is_final_merge_to_output) {
00316                 merged_run_id = output_id_;
00317                 detail::LogInfo("KWayMergeSorter: Merging to final output: " + output_id_);
00318             } else {
00319                 auto temp_stream =
00320                     stream_factory_.CreateTempOutputStream(merged_run_id, file_io_buffer_elements_);
00321                 temp_stream->Finalize();
00322             }
00323
00324             MergeGroupOfRuns(group_to_merge, merged_run_id);
00325             next_pass_run_ids.push_back(std::move(merged_run_id));
00326
00327             for (const auto& id_to_del : group_to_merge) {
00328                 runs_to_delete_this_pass.push_back(id_to_del);
00329             }
00330         }
00331         current_run_ids = next_pass_run_ids;
00332         for (const auto& id_del : runs_to_delete_this_pass) {
00333             if (id_del != output_id_) {
00334                 stream_factory_.DeleteStorage(id_del);
00335             }
00336         }
00337     }
00338
00339     if (current_run_ids.size() == 1) {
00340         if (current_run_ids[0] != output_id_) {
00341             detail::LogInfo("KWayMergeSorter: Finalizing " + current_run_ids[0] + " as " +
00342                             output_id_);
00343             stream_factory_.MakeStoragePermanent(current_run_ids[0], output_id_);
00344         } else {
00345             detail::LogInfo("KWayMergeSorter: Output is already in " + output_id_);
00346         }
00347     } else if (current_run_ids.empty() && !stream_factory_.StorageExists(output_id_)) {
```

```
00348         detail::LogInfo("KWayMergeSorter: No runs left and output " + output_id_ +
00349                         " does not exist. Creating empty.");
00350         auto empty_out = stream_factory_.CreateOutputStream(output_id_, file_io_buffer_elements_);
00351         empty_out->Finalize();
00352     } else if (!current_run_ids.empty()) {
00353         detail::LogError("Error: KWayMergeSorter finished with unexpected runs: " +
00354                          std::to_string(current_run_ids.size()));
00355     }
00356 }
```

References external_sort::KWayMergeSorter< T >::CreateInitialRuns(), external_sort::KWayMergeSorter< T >::file_io_buffer_elem
external_sort::KWayMergeSorter< T >::k_way_degree_, external_sort::detail::LogError(), external_sort::detail::LogInfo(),
external_sort::KWayMergeSorter< T >::MergeGroupOfRuns(), external_sort::KWayMergeSorter< T >::output_id_,
and external_sort::KWayMergeSorter< T >::stream_factory_.

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:

8.17.4 Member Data Documentation

8.17.4.1 ascending_

```
template<typename T >
bool external_sort::KWayMergeSorter< T >::ascending_  [private]
```

Ascending sort flag.

Definition at line 64 of file k_way_merge_sorter.hpp.

Referenced by external_sort::KWayMergeSorter< T >::CreateInitialRuns(), external_sort::KWayMergeSor
and external_sort::KWayMergeSorter< T >::MergeGroupOfRuns().

8.17.4.2 file_io_buffer_elements_

```
template<typename T >
uint64_t external_sort::KWayMergeSorter< T >::file_io_buffer_elements_  [private]
```

Size of I/O buffer in elements.

Definition at line 63 of file k_way_merge_sorter.hpp.

Referenced by external_sort::KWayMergeSorter< T >::CreateInitialRuns(), external_sort::KWayMergeSorter< T >::KWayMergeSor
external_sort::KWayMergeSorter< T >::MergeGroupOfRuns(), and external_sort::KWayMergeSorter< T >::Sort().

8.17.4.3 input_id_

```
template<typename T >
io::StorageId external_sort::KWayMergeSorter< T >::input_id_  [private]
```

Input storage identifier.

Definition at line 59 of file k_way_merge_sorter.hpp.

Referenced by external_sort::KWayMergeSorter< T >::CreateInitialRuns(), and external_sort::KWayMerge

8.17.4.4 k_way_degree_

```
template<typename T >
uint64_t external_sort::KWayMergeSorter< T >::k_way_degree_  [private]
```

Degree of k-way merging.

Definition at line 62 of file k_way_merge_sorter.hpp.

Referenced by external_sort::KWayMergeSorter< T >::KWayMergeSorter(), and external_sort::KWayMergeSorter< T >::Sort().

8.17.4.5 memory_for_runs_bytes_

```
template<typename T >
uint64_t external_sort::KWayMergeSorter< T >::memory_for_runs_bytes_  [private]
```

Memory for creating initial runs (bytes)

Definition at line 61 of file k_way_merge_sorter.hpp.

Referenced by external_sort::KWayMergeSorter< T >::CreateInitialRuns(), and external_sort::KWayMerge

8.17.4.6 output_id_

```
template<typename T >
io::StorageId external_sort::KWayMergeSorter< T >::output_id_  [private]
```

Output storage identifier.

Definition at line 60 of file k_way_merge_sorter.hpp.

Referenced by external_sort::KWayMergeSorter< T >::KWayMergeSorter(), and external_sort::KWayMergeSorter< T >::Sort().

8.17.4.7 stream_factory_

```
template<typename T >
io::IStreamFactory<T>& external_sort::KWayMergeSorter< T >::stream_factory_  [private]
```

Reference to stream factory.

Definition at line 58 of file k_way_merge_sorter.hpp.

Referenced by external_sort::KWayMergeSorter< T >::CreateInitialRuns(), external_sort::KWayMergeSorter< T >::KWayMergeSor external_sort::KWayMergeSorter< T >::MergeGroupOfRuns(), and external_sort::KWayMergeSorter< T >::Sort().

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

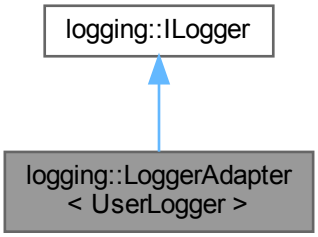
- external_sort/include/k_way_merge_sorter.hpp

8.18 logging::LoggerAdapter< UserLogger > Class Template Reference

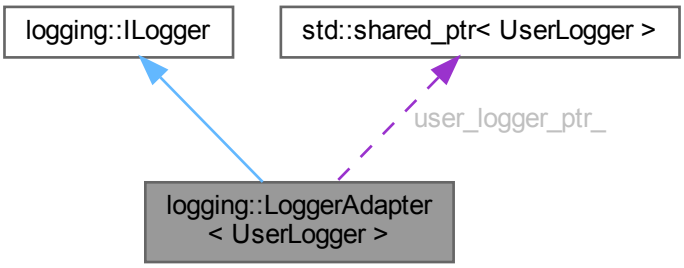
Adapter class that wraps a user-defined logger to conform to the ILogger interface.

`#include <LoggerAdapter.hpp>`

Inheritance diagram for logging::LoggerAdapter< UserLogger >:



Collaboration diagram for logging::LoggerAdapter< UserLogger >:



Public Member Functions

- template<typename T >
  requires std::copy_constructible<std::decay_t<T>> && (!std::same_as<std::decay_t<T>, std::shared_ptr<UserLogger>>)
  LoggerAdapter (T &&logger)

  Construct adapter for copyable logger by forwarding the user logger.
- LoggerAdapter (std::shared_ptr< UserLogger > logger)

  Construct adapter for non-copyable logger via shared_ptr.
- void LogInfo (const std::string &message) override

  Log an informational message using the wrapped logger.
- void LogWarning (const std::string &message) override

  Log a warning message using the wrapped logger.
- void LogError (const std::string &message) override

  Log an error message using the wrapped logger.

Public Member Functions inherited from logging::ILogger

- virtual ~ILogger ()=default

Private Attributes

- std::optional< UserLogger > user_logger_

  Logger stored by value (for copyable types)
- std::shared_ptr< UserLogger > user_logger_ptr_

  Logger stored by pointer (for non-copyable)

8.18.1 Detailed Description

template<typename UserLogger>
class logging::LoggerAdapter< UserLogger >

Adapter class that wraps a user-defined logger to conform to the ILogger interface.

This template class allows any custom logger implementation to be used with current library by adapting its interface to match the ILogger contract. The user logger must provide info(), warn(), and error() methods that accept string messages.

The adapter automatically handles both copyable and non-copyable loggers:

- Copyable loggers are stored by value
- Non-copyable loggers are stored via shared_ptr

Template Parameters

UserLogger	The type of the user-defined logger to be adapted

Example usage with copyable logger:

```
struct MyLogger {
    void info(const std::string& msg) { std::cout « "[INFO] " « msg « '\n'; }
    void warn(const std::string& msg) { std::cout « "[WARN] " « msg « '\n'; }
    void error(const std::string& msg) { std::cerr « "[ERROR] " « msg « '\n'; }
};

auto adapter = std::make_shared<LoggerAdapter<MyLogger»(MyLogger{});
```

Example usage with non-copyable logger stored via shared_ptr:

```
struct HeavyLogger {
    HeavyLogger() = default;
    HeavyLogger(const HeavyLogger&) = delete;
    void info(const std::string& msg) { }
    void warn(const std::string& msg) { }
    void error(const std::string& msg) { }
};

auto adapter = std::make_shared<LoggerAdapter<HeavyLogger»(
    std::make_shared<HeavyLogger>());
```

Definition at line 57 of file LoggerAdapter.hpp.

8.18.2 Constructor & Destructor Documentation

8.18.2.1 LoggerAdapter() [1/2]

```
template<typename UserLogger >
template<typename T >
requires std::copy_constructible<std::decay_t<T>> && (!std::same_as<std::decay_t<T>, std↩
::shared_ptr<UserLogger>>)
logging::LoggerAdapter< UserLogger >::LoggerAdapter (
            T && logger )  [inline], [explicit]
```

Construct adapter for copyable logger by forwarding the user logger.

Template Parameters

T	Deduced type for perfect forwarding

Parameters

logger	The user-defined logger instance to wrap (by value)

Definition at line 67 of file LoggerAdapter.hpp.

```
00067                                     : user_logger_(std::forward<T>(logger)) {
00068     }
```

References logging::LoggerAdapter< UserLogger >::user_logger_.

8.18.2.2 LoggerAdapter() [2/2]

```
template<typename UserLogger >
logging::LoggerAdapter< UserLogger >::LoggerAdapter (
            std::shared_ptr< UserLogger > logger )  [inline], [explicit]
```

Construct adapter for non-copyable logger via shared_ptr.

Parameters

logger	Shared pointer to the user-defined logger instance

Definition at line 74 of file LoggerAdapter.hpp.

```
00075         : user_logger_ptr_(std::move(logger)) {
00076     }
```

References logging::LoggerAdapter< UserLogger >::user_logger_ptr_.

8.18.3 Member Function Documentation

8.18.3.1 LogError()

```
template<typename UserLogger >
void logging::LoggerAdapter< UserLogger >::LogError (
             const std::string & message )  [inline], [override], [virtual]
```

Log an error message using the wrapped logger.

Parameters

message	The message to log

Implements logging::ILogger.

Definition at line 106 of file LoggerAdapter.hpp.

```
00106                                                    {
00107         if (user_logger_ptr_) {
00108             user_logger_ptr_->error(message);
00109         } else {
00110             user_logger_->error(message);
00111         }
00112     }
```

References logging::LoggerAdapter< UserLogger >::user_logger_, and logging::LoggerAdapter< UserLogger >::user_logger_ptr_.

8.18.3.2 LogInfo()

```
template<typename UserLogger >
void logging::LoggerAdapter< UserLogger >::LogInfo (
             const std::string & message )  [inline], [override], [virtual]
```

Log an informational message using the wrapped logger.

Parameters

message	The message to log

Implements logging::ILogger.

Definition at line 82 of file LoggerAdapter.hpp.

```
00082                                                   {
00083         if (user_logger_ptr_) {
00084             user_logger_ptr_->info(message);
00085         } else {
00086             user_logger_->info(message);
00087         }
00088     }
```

References logging::LoggerAdapter< UserLogger >::user_logger_, and logging::LoggerAdapter< UserLogger >::user_logger_ptr_.

8.18.3.3 LogWarning()

```
template<typename UserLogger >
void logging::LoggerAdapter< UserLogger >::LogWarning (
            const std::string & message )  [inline], [override], [virtual]
```

Log a warning message using the wrapped logger.

Parameters

message	The message to log

Implements logging::ILogger.

Definition at line 94 of file LoggerAdapter.hpp.

```
00094                                                   {
00095         if (user_logger_ptr_) {
00096             user_logger_ptr_->warn(message);
00097         } else {
00098             user_logger_->warn(message);
00099         }
00100     }
```

References logging::LoggerAdapter< UserLogger >::user_logger_, and logging::LoggerAdapter< UserLogger >::user_logger_ptr_.

8.18.4 Member Data Documentation

8.18.4.1 user_logger_

```
template<typename UserLogger >
std::optional<UserLogger> logging::LoggerAdapter< UserLogger >::user_logger_  [private]
```

Logger stored by value (for copyable types)

Definition at line 115 of file LoggerAdapter.hpp.

Referenced by logging::LoggerAdapter< UserLogger >::LogError(), logging::LoggerAdapter< UserLogger >::LoggerAdapter(), logging::LoggerAdapter< UserLogger >::LogInfo(), and logging::LoggerAdapter< UserLogger >::LogWarning().

8.18.4.2 user_logger_ptr_

```
template<typename UserLogger >
std::shared_ptr<UserLogger> logging::LoggerAdapter< UserLogger >::user_logger_ptr_  [private]
```

Logger stored by pointer (for non-copyable)

Definition at line 116 of file LoggerAdapter.hpp.

Referenced by logging::LoggerAdapter< UserLogger >::LogError(), logging::LoggerAdapter< UserLogger >::LoggerAdapter(), logging::LoggerAdapter< UserLogger >::LogInfo(), and logging::LoggerAdapter< UserLogger >::LogWarning().

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

- logging/include/LoggerAdapter.hpp

8.19 external_sort::MergeSource< T > Struct Template Reference

Structure representing a data source for merging.

`#include <k_way_merge_sorter.hpp>`

Public Attributes

- io::IInputStream< T > * stream
  Pointer to source stream.

8.19.1 Detailed Description

template<typename T>
struct external_sort::MergeSource< T >

Structure representing a data source for merging.

Used in priority queue for k-way merging.

Template Parameters

T	Type of data elements

Definition at line 27 of file k_way_merge_sorter.hpp.

8.19.2 Member Data Documentation

8.19.2.1 stream

```
template<typename T >
io::IInputStream<T>* external_sort::MergeSource< T >::stream
```

Pointer to source stream.

Definition at line 28 of file k_way_merge_sorter.hpp.

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

- external_sort/include/k_way_merge_sorter.hpp

8.20 external_sort::KWayMergeSorter< T >::MergeSourceComparator Struct Reference

Comparator for merge sources.

Public Member Functions

- MergeSourceComparator (bool asc)
  Comparator constructor.
- bool operator() (const MergeSource< T > &a, const MergeSource< T > &b) const
  Comparison operator.

Public Attributes

- bool ascending
  Sort direction flag.

8.20.1 Detailed Description

template<typename T>
struct external_sort::KWayMergeSorter< T >::MergeSourceComparator

Comparator for merge sources.

Definition at line 69 of file k_way_merge_sorter.hpp.

8.20.2 Constructor & Destructor Documentation

8.20.2.1 MergeSourceComparator()

```
template<typename T >
external_sort::KWayMergeSorter< T >::MergeSourceComparator::MergeSourceComparator (
            bool asc )  [inline], [explicit]
```

Comparator constructor.

Parameters

asc	Sort direction (true - ascending)

Definition at line 76 of file k_way_merge_sorter.hpp.

```
00076                                           : ascending(asc) {
00077         }
```

References external_sort::KWayMergeSorter< T >::MergeSourceComparator::ascending.

8.20.3 Member Function Documentation

8.20.3.1 operator()()

```
template<typename T >
bool external_sort::KWayMergeSorter< T >::MergeSourceComparator::operator() (
```

```
const MergeSource< T > & a,
const MergeSource< T > & b ) const  [inline]
```

Comparison operator.

Parameters

a	First source
b	Second source

Returns

Comparison result for priority queue

Definition at line 85 of file k_way_merge_sorter.hpp.

```
00085                                                                                {
00086             const T& va = a.stream->Value();
00087             const T& vb = b.stream->Value();
00088             return ascending ? (va > vb) : (va < vb);
00089         }
```

References external_sort::KWayMergeSorter< T >::MergeSourceComparator::ascending.

8.20.4 Member Data Documentation

8.20.4.1 ascending

```
template<typename T >
bool external_sort::KWayMergeSorter< T >::MergeSourceComparator::ascending
```

Sort direction flag.

Definition at line 70 of file k_way_merge_sorter.hpp.

Referenced by external_sort::KWayMergeSorter< T >::MergeSourceComparator::MergeSourceComparator(), and external_sort::KWayMergeSorter< T >::MergeSourceComparator::operator()().

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

- external_sort/include/k_way_merge_sorter.hpp

8.21 external_sort::MergeSourceComparatorStatic< T > Struct Template Reference

Comparator for merge sources that compares the current value of the streams.

```
#include <k_way_merge_sorter.hpp>
```

Public Member Functions

- MergeSourceComparatorStatic (bool asc)
- bool operator() (const MergeSource< T > &a, const MergeSource< T > &b) const

Public Attributes

- bool ascending

8.21.1 Detailed Description

template<typename T>
struct external_sort::MergeSourceComparatorStatic< T >

Comparator for merge sources that compares the current value of the streams.

Definition at line 35 of file k_way_merge_sorter.hpp.

8.21.2 Constructor & Destructor Documentation

8.21.2.1 MergeSourceComparatorStatic()

```
template<typename T >
external_sort::MergeSourceComparatorStatic< T >::MergeSourceComparatorStatic (
            bool asc )  [inline], [explicit]
```

Definition at line 37 of file k_way_merge_sorter.hpp.

```
00037                                                : ascending(asc) {
00038     }
```

References external_sort::MergeSourceComparatorStatic< T >::ascending.

8.21.3 Member Function Documentation

8.21.3.1 operator()()

```
template<typename T >
bool external_sort::MergeSourceComparatorStatic< T >::operator() (
            const MergeSource< T > & a,
            const MergeSource< T > & b ) const  [inline]
```

Definition at line 40 of file k_way_merge_sorter.hpp.

```
00040                                                                        {
00041         const T& va = a.stream->Value();
00042         const T& vb = b.stream->Value();
00043         return ascending ? (va > vb) : (va < vb);
00044     }
```

References external_sort::MergeSourceComparatorStatic< T >::ascending.

8.21.4 Member Data Documentation

8.21.4.1 ascending

```
template<typename T >
bool external_sort::MergeSourceComparatorStatic< T >::ascending
```

Definition at line 36 of file k_way_merge_sorter.hpp.

Referenced by external_sort::MergeSourceComparatorStatic< T >::MergeSourceComparatorStatic(), and external_sort::MergeSourceComparatorStatic< T >::operator()().

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

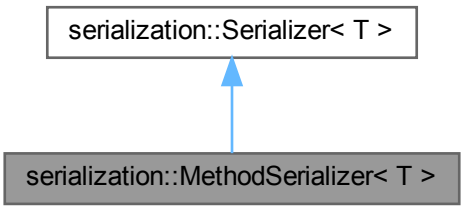
- external_sort/include/k_way_merge_sorter.hpp

8.22 serialization::MethodSerializer< T > Class Template Reference

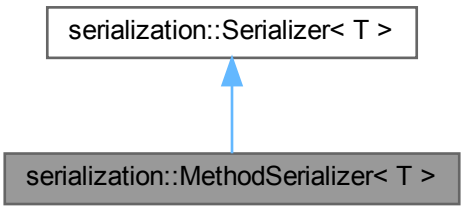
Serializer for types with method serialization.

```
#include <serializers.hpp>
```

Inheritance diagram for serialization::MethodSerializer< T >:



Collaboration diagram for serialization::MethodSerializer< T >:



Public Member Functions

- bool Serialize (const T &obj, FILE *file) override

  Serialize object into file.
- bool Deserialize (T &obj, FILE *file) override

  Deserialize object from file.
- uint64_t GetSerializedSize (const T &obj) override

  Calculate the serialized size of an object in bytes.

Public Member Functions inherited from serialization::Serializer< T >

- virtual ~Serializer ()=default
- Serializer ()=default
- Serializer (const Serializer &)=default
- Serializer & operator= (const Serializer &)=default
- Serializer (Serializer &&)=default
- Serializer & operator= (Serializer &&)=default

Additional Inherited Members

Public Types inherited from serialization::Serializer< T >

- using Specialized = std::false_type

8.22.1 Detailed Description

template<MethodSerializable T>
class serialization::MethodSerializer< T >

Serializer for types with method serialization.

This serializer is used for types that provide Serialize() and Deserialize() methods as member functions. It supports two performance modes for size calculation:

Optimized Mode (Recommended): If the type provides a `GetSerializedSize() const` method returning uint64_t, it will be called directly for fast size calculation without I/O overhead.

Fallback Mode: If GetSerializedSize() is not provided, the serializer will perform actual serialization to /dev/null to count bytes. This works but is slower.

Performance Tip:

For best performance, especially in benchmarks or when processing large datasets, add a GetSerializedSize() method to your type:

```
struct MyType {
    bool Serialize(FILE* file) const { ... }
    bool Deserialize(FILE* file) { ... }

    // Add this for optimal performance:
    uint64_t GetSerializedSize() const {
        return ...; // Calculate total size
    }
};
```

Template Parameters

T	Type with Serialize() and Deserialize() member methods

See also

serialization::MethodSerializable

Definition at line 408 of file serializers.hpp.

8.22.2 Member Function Documentation

8.22.2.1 Deserialize()

```
template<MethodSerializable T>
bool serialization::MethodSerializer< T >::Deserialize (
            T & obj,
            FILE * file )  [inline], [override], [virtual]
```

Deserialize object from file.

Parameters

obj	Object to deserialize to
file	Pointer to file to read from

Returns

true, if deserialization is successful

Implements serialization::Serializer< T >.

Definition at line 414 of file serializers.hpp.

```
00414                                                   {
00415         return obj.Deserialize(file);
00416     }
```

8.22.2.2 GetSerializedSize()

```
template<MethodSerializable T>
uint64_t serialization::MethodSerializer< T >::GetSerializedSize (
            const T & obj )  [inline], [override], [virtual]
```

Calculate the serialized size of an object in bytes.

This method computes the exact number of bytes that would be written to a file when serializing the given object. This is useful for:

- Estimating storage requirements before serialization
- Tracking actual data size for complex types (e.g., types with dynamic memory)
- Generating correct amount of data to match target file sizes in benchmarks

Parameters

obj	Object to calculate serialized size for

Returns

Number of bytes that would be written when serializing this object

Note

For POD types, this returns sizeof(T). For complex types (std::string, std::vector, custom types), this computes the actual serialized size including all dynamic data.

Performance Optimization for Custom Types:

For types using MethodSerializer (types with Serialize/Deserialize methods), there are two implementation paths:

1. **Optimized path (RECOMMENDED)**: If your type provides a `GetSerializedSize()` const method, it will be called directly. This is **much faster** as it only calculates the size without actual serialization.
2. **Fallback path**: If `GetSerializedSize()` is not provided, the serializer will perform actual serialization to /dev/null and count bytes. This works but is **significantly slower** (involves I/O operations).

Example (Basic usage):

```
auto serializer = serialization::CreateSerializer<Person>();
Person p{"Alice", 30, 170.5, 65, "123 Main St"};
uint64_t size = serializer->GetSerializedSize(p);
// size includes: sizeof(name length) + name data + age + height + weight +
//                sizeof(address length) + address data
```

Example (Optimized custom type):

```
struct Person {
    std::string name;
    int32_t age;
    std::string address;

    bool Serialize(FILE* file) const { ... }
    bool Deserialize(FILE* file) { ... }

    // Add this method for optimal performance!
    uint64_t GetSerializedSize() const {
        return sizeof(size_t) + name.size()
             + sizeof(int32_t)
             + sizeof(size_t) + address.size();
    }
};
```

Warning

If your type is used in performance-critical code (e.g., benchmarks, large datasets), strongly consider implementing GetSerializedSize() method to avoid I/O overhead.

Implements serialization::Serializer< T >.

Definition at line 418 of file serializers.hpp.

```
00418                                                                 {
00419         // For method-based serialization, first check if the type provides GetSerializedSize() method
00420         if constexpr (requires(const T& t) { { t.GetSerializedSize() } ->
      std::convertible_to<uint64_t>; }) {
00421             return obj.GetSerializedSize();
00422         } else {
00423             // Fallback: calculate size by actually serializing to /dev/null
00424             return detail::CalculateSizeBySerializing(obj, [](const T& o, FILE* f) {
00425                 return o.Serialize(f);
00426             });
00427         }
00428     }
```

8.22.2.3 Serialize()

```
template<MethodSerializable T>
bool serialization::MethodSerializer< T >::Serialize (
            const T & obj,
            FILE * file )  [inline], [override], [virtual]
```

Serialize object into file.

Parameters

obj	Object ro serialize
file	Pointer to file to write to

Returns

true, if serialization is successful

Implements serialization::Serializer< T >.

Definition at line 410 of file serializers.hpp.

```
00410                                                       {
00411         return obj.Serialize(file);
00412     }
```

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

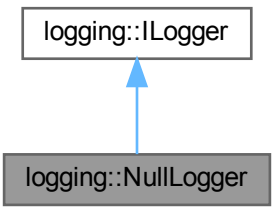
- serialization/include/serializers.hpp

8.23 logging::NullLogger Class Reference

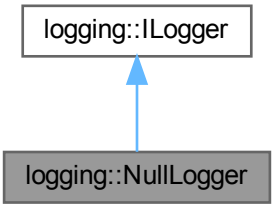
A logger implementation that discards all log messages.

```
#include <NullLogger.hpp>
```

Inheritance diagram for logging::NullLogger:



Collaboration diagram for logging::NullLogger:



Public Member Functions

- void LogInfo (const std::string &message) override
  Log an informational message (no-op)
- void LogWarning (const std::string &message) override
  Log a warning message (no-op)
- void LogError (const std::string &message) override
  Log an error message (no-op)

Public Member Functions inherited from logging::ILogger

- virtual ~ILogger ()=default

8.23.1 Detailed Description

A logger implementation that discards all log messages.

This is the default logger used when no custom logger is set. All logging methods are no-ops, making it suitable for production code where logging overhead should be minimal or when logging is not desired.

Definition at line 21 of file NullLogger.hpp.

8.23.2 Member Function Documentation

8.23.2.1 LogError()

```
void logging::NullLogger::LogError (
            const std::string & message )  [inline], [override], [virtual]
```

Log an error message (no-op)

Parameters

message	The message to discard (unused)

Implements logging::ILogger.

Definition at line 43 of file NullLogger.hpp.

```
00043                                                 {
00044         (void)message;  // Suppress unused parameter warning
00045     }
```

8.23.2.2 LogInfo()

```
void logging::NullLogger::LogInfo (
            const std::string & message )  [inline], [override], [virtual]
```

Log an informational message (no-op)

Parameters

message	The message to discard (unused)

Implements logging::ILogger.

Definition at line 27 of file NullLogger.hpp.

```
00027                                               {
00028         (void)message;  // Suppress unused parameter warning
00029     }
```

8.23.2.3 LogWarning()

```
void logging::NullLogger::LogWarning (
            const std::string & message )  [inline], [override], [virtual]
```

Log a warning message (no-op)

Parameters

message	The message to discard (unused)

Implements logging::ILogger.

Definition at line 35 of file NullLogger.hpp.

```
00035                                                 {
00036         (void)message;  // Suppress unused parameter warning
00037     }
```

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

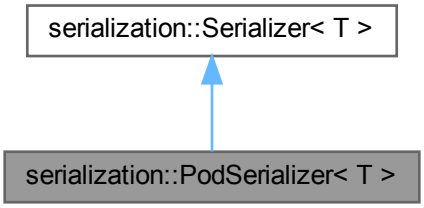
- logging/include/NullLogger.hpp

8.24 serialization::PodSerializer< T > Class Template Reference

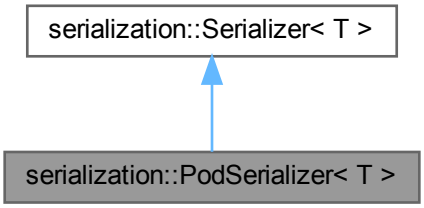
Serializer for POD types.

`#include <serializers.hpp>`

Inheritance diagram for serialization::PodSerializer< T >:



Collaboration diagram for serialization::PodSerializer< T >:



Public Member Functions

- bool Serialize (const T &obj, FILE *file) override
  Serialize object into file.
- bool Deserialize (T &obj, FILE *file) override
  Deserialize object from file.
- uint64_t GetSerializedSize (const T &obj) override
  Calculate the serialized size of an object in bytes.

Public Member Functions inherited from serialization::Serializer< T >

- virtual ~Serializer ()=default
- Serializer ()=default
- Serializer (const Serializer &)=default
- Serializer & operator= (const Serializer &)=default
- Serializer (Serializer &&)=default
- Serializer & operator= (Serializer &&)=default

Additional Inherited Members

Public Types inherited from serialization::Serializer< T >

- using Specialized = std::false_type

8.24.1 Detailed Description

template<PodSerializable T>
class serialization::PodSerializer< T >

Serializer for POD types.

See also

serialization::PodSerializable

Definition at line 304 of file serializers.hpp.

8.24.2 Member Function Documentation

8.24.2.1 Deserialize()

```
template<PodSerializable T>
bool serialization::PodSerializer< T >::Deserialize (
            T & obj,
            FILE * file )  [inline], [override], [virtual]
```

Deserialize object from file.

Parameters

obj	Object to deserialize to
file	Pointer to file to read from

Returns

true, if deserialization is successful

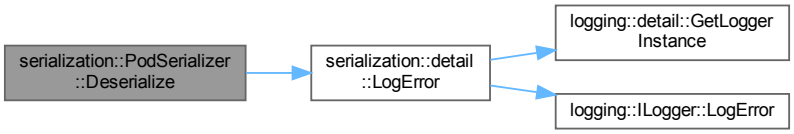
Implements serialization::Serializer< T >.

Definition at line 314 of file serializers.hpp.

```
00314                                                {
00315         bool result = fread(&obj, sizeof(T), 1, file) == 1;
00316         if (!result) {
00317             detail::LogError("Failed to deserialize POD type of size " + std::to_string(sizeof(T)));
00318         }
00319         return result;
00320     }
```

References serialization::detail::LogError().

Here is the call graph for this function:



8.24.2.2 GetSerializedSize()

```
template<PodSerializable T>
uint64_t serialization::PodSerializer< T >::GetSerializedSize (
            const T & obj )  [inline], [override], [virtual]
```

Calculate the serialized size of an object in bytes.

This method computes the exact number of bytes that would be written to a file when serializing the given object. This is useful for:

- Estimating storage requirements before serialization
- Tracking actual data size for complex types (e.g., types with dynamic memory)
- Generating correct amount of data to match target file sizes in benchmarks

Parameters

obj	Object to calculate serialized size for

Returns

Number of bytes that would be written when serializing this object

Note

For POD types, this returns sizeof(T). For complex types (std::string, std::vector, custom types), this computes the actual serialized size including all dynamic data.

Performance Optimization for Custom Types:

For types using MethodSerializer (types with Serialize/Deserialize methods), there are two implementation paths:

1. **Optimized path (RECOMMENDED)**: If your type provides a `GetSerializedSize()` const method, it will be called directly. This is **much faster** as it only calculates the size without actual serialization.
2. **Fallback path**: If `GetSerializedSize()` is not provided, the serializer will perform actual serialization to /dev/null and count bytes. This works but is **significantly slower** (involves I/O operations).

Example (Basic usage):

```
auto serializer = serialization::CreateSerializer<Person>();
Person p{"Alice", 30, 170.5, 65, "123 Main St"};
uint64_t size = serializer->GetSerializedSize(p);
// size includes: sizeof(name length) + name data + age + height + weight +
//                sizeof(address length) + address data
```

Example (Optimized custom type):

```
struct Person {
    std::string name;
    int32_t age;
    std::string address;

    bool Serialize(FILE* file) const { ... }
    bool Deserialize(FILE* file) { ... }

    // Add this method for optimal performance!
    uint64_t GetSerializedSize() const {
        return sizeof(size_t) + name.size()
             + sizeof(int32_t)
             + sizeof(size_t) + address.size();
    }
};
```

Warning

If your type is used in performance-critical code (e.g., benchmarks, large datasets), strongly consider implementing GetSerializedSize() method to avoid I/O overhead.

Implements serialization::Serializer< T >.

Definition at line 322 of file serializers.hpp.

```
00322                                                                    {
00323         return sizeof(T);
00324     }
```

8.24.2.3 Serialize()

```
template<PodSerializable T>
bool serialization::PodSerializer< T >::Serialize (
            const T & obj,
            FILE * file )  [inline], [override], [virtual]
```

Serialize object into file.

Parameters

obj	Object ro serialize
file	Pointer to file to write to

Returns

true, if serialization is successful

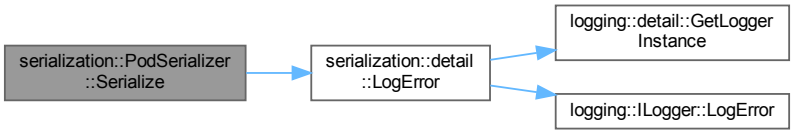
Implements serialization::Serializer< T >.

Definition at line 306 of file serializers.hpp.

```
00306                                                   {
00307         bool result = fwrite(&obj, sizeof(T), 1, file) == 1;
00308         if (!result) {
00309             detail::LogError("Failed to serialize POD type of size " + std::to_string(sizeof(T)));
00310         }
00311         return result;
00312     }
```

References serialization::detail::LogError().

Here is the call graph for this function:



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

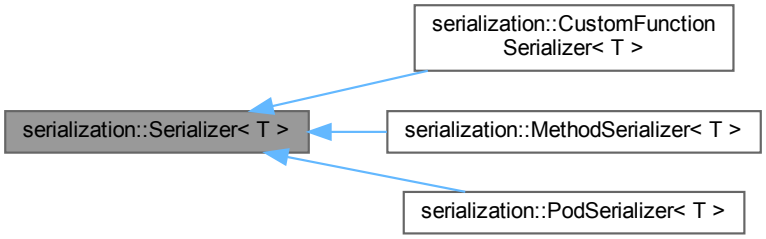
- serialization/include/serializers.hpp

8.25 serialization::Serializer< T > Class Template Reference

Base interface for serializable types.

`#include <serializers.hpp>`

Inheritance diagram for serialization::Serializer< T >:



Public Types

- using Specialized = std::false_type

Public Member Functions

- virtual ~Serializer ()=default
- Serializer ()=default
- Serializer (const Serializer &)=default
- Serializer & operator= (const Serializer &)=default
- Serializer (Serializer &&)=default
- Serializer & operator= (Serializer &&)=default
- virtual bool Serialize (const T &obj, FILE *file)=0

  Serialize object into file.
- virtual bool Deserialize (T &obj, FILE *file)=0

  Deserialize object from file.
- virtual uint64_t GetSerializedSize (const T &obj)=0

  Calculate the serialized size of an object in bytes.

8.25.1 Detailed Description

template<typename T>
class serialization::Serializer< T >

Base interface for serializable types.

Definition at line 205 of file serializers.hpp.

8.25.2 Member Typedef Documentation

8.25.2.1 Specialized

```
template<typename T >
using serialization::Serializer< T >::Specialized = std::false_type
```

Definition at line 207 of file serializers.hpp.

8.25.3 Constructor & Destructor Documentation

8.25.3.1 ~Serializer()

```
template<typename T >
virtual serialization::Serializer< T >::~Serializer ( )  [virtual], [default]
```

8.25.3.2 Serializer() [1/3]

```
template<typename T >
serialization::Serializer< T >::Serializer ( )  [default]
```

8.25.3.3 Serializer() [2/3]

```
template<typename T >
serialization::Serializer< T >::Serializer (
            const Serializer< T > &  )  [default]
```

8.25.3.4 Serializer() [3/3]

```
template<typename T >
serialization::Serializer< T >::Serializer (
            Serializer< T > &&  )  [default]
```

8.25.4 Member Function Documentation

8.25.4.1 Deserialize()

```
template<typename T >
virtual bool serialization::Serializer< T >::Deserialize (
            T & obj,
            FILE * file )  [pure virtual]
```

Deserialize object from file.

Parameters

obj	Object to deserialize to
file	Pointer to file to read from

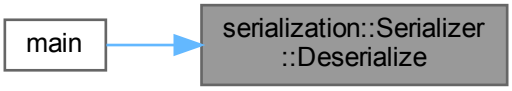
Returns

true, if deserialization is successful

Implemented in serialization::PodSerializer< T >, serialization::CustomFunctionSerializer< T >, and serialization::MethodSerializer<

Referenced by main().

Here is the caller graph for this function:



8.25.4.2 GetSerializedSize()

```
template<typename T >
virtual uint64_t serialization::Serializer< T >::GetSerializedSize (
            const T & obj )  [pure virtual]
```

Calculate the serialized size of an object in bytes.

This method computes the exact number of bytes that would be written to a file when serializing the given object. This is useful for:

- Estimating storage requirements before serialization
- Tracking actual data size for complex types (e.g., types with dynamic memory)
- Generating correct amount of data to match target file sizes in benchmarks

Parameters

obj	Object to calculate serialized size for

Returns

Number of bytes that would be written when serializing this object

Note

For POD types, this returns sizeof(T). For complex types (std::string, std::vector, custom types), this computes the actual serialized size including all dynamic data.

Performance Optimization for Custom Types:

For types using MethodSerializer (types with Serialize/Deserialize methods), there are two implementation paths:

1. **Optimized path (RECOMMENDED)**: If your type provides a `GetSerializedSize()` const method, it will be called directly. This is **much faster** as it only calculates the size without actual serialization.
2. **Fallback path**: If `GetSerializedSize()` is not provided, the serializer will perform actual serialization to /dev/null and count bytes. This works but is **significantly slower** (involves I/O operations).

Example (Basic usage):

```
auto serializer = serialization::CreateSerializer<Person>();
Person p{"Alice", 30, 170.5, 65, "123 Main St"};
uint64_t size = serializer->GetSerializedSize(p);
// size includes: sizeof(name length) + name data + age + height + weight +
//                sizeof(address length) + address data
```

Example (Optimized custom type):

```
struct Person {
    std::string name;
    int32_t age;
    std::string address;

    bool Serialize(FILE* file) const { ... }
    bool Deserialize(FILE* file) { ... }

    // Add this method for optimal performance!
    uint64_t GetSerializedSize() const {
        return sizeof(size_t) + name.size()
             + sizeof(int32_t)
             + sizeof(size_t) + address.size();
    }
};
```

Warning

If your type is used in performance-critical code (e.g., benchmarks, large datasets), strongly consider implementing GetSerializedSize() method to avoid I/O overhead.

Implemented in serialization::PodSerializer< T >, serialization::CustomFunctionSerializer< T >, and serialization::MethodSerializer<

8.25.4.3 operator=() [1/2]

```
template<typename T >
Serializer & serialization::Serializer< T >::operator= (
            const Serializer< T > &  )  [default]
```

8.25.4.4 operator=() [2/2]

```
template<typename T >
Serializer & serialization::Serializer< T >::operator= (
            Serializer< T > &&  )  [default]
```

8.25.4.5 Serialize()

```
template<typename T >
virtual bool serialization::Serializer< T >::Serialize (
            const T & obj,
            FILE * file )  [pure virtual]
```

Serialize object into file.

Parameters

obj	Object ro serialize
file	Pointer to file to write to

Returns

true, if serialization is successful

Implemented in serialization::PodSerializer< T >, serialization::CustomFunctionSerializer< T >, and serialization::MethodSerializer<

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

- serialization/include/serializers.hpp

8.26 serialization::Serializer< std::string > Class Reference

`#include <serializers.hpp>`

Public Types

- using Specialized = std::true_type

Public Member Functions

- bool Serialize (const std::string &obj, FILE *file)
- bool Deserialize (std::string &obj, FILE *file)
- uint64_t GetSerializedSize (const std::string &obj)

8.26.1 Detailed Description

Definition at line 453 of file serializers.hpp.

8.26.2 Member Typedef Documentation

8.26.2.1 Specialized

```
using serialization::Serializer< std::string >::Specialized = std::true_type
```

Definition at line 455 of file serializers.hpp.

8.26.3 Member Function Documentation

8.26.3.1 Deserialize()

```
bool serialization::Serializer< std::string >::Deserialize (
            std::string & obj,
            FILE * file )  [inline]
```

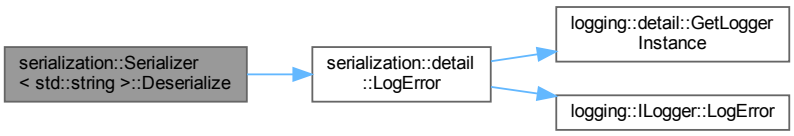
Definition at line 469 of file serializers.hpp.

```
00469                                                {
00470         uint64_t length;
00471         if (fread(&length, sizeof(uint64_t), 1, file) != 1) {
00472             detail::LogError("Failed to read string length");
00473             return false;
00474         }
00475         obj.resize(length);
00476         if (fread(&obj[0], sizeof(char), length, file) != length) {
00477             detail::LogError("Failed to read string data of length: " + std::to_string(length));
00478             return false;
00479         }
00480         return true;
00481     }
```

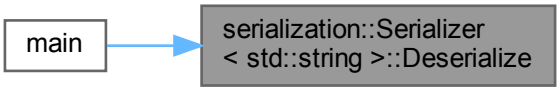
References serialization::detail::LogError().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:



8.26.3.2 GetSerializedSize()

```
uint64_t serialization::Serializer< std::string >::GetSerializedSize (
            const std::string & obj )  [inline]
```

Definition at line 483 of file serializers.hpp.

```
00483                                                      {
00484         // Size = sizeof(length field) + actual string data
00485         return sizeof(uint64_t) + obj.length() * sizeof(char);
00486     }
```

8.26.3.3 Serialize()

```
bool serialization::Serializer< std::string >::Serialize (
            const std::string & obj,
            FILE * file )  [inline]
```

Definition at line 456 of file serializers.hpp.

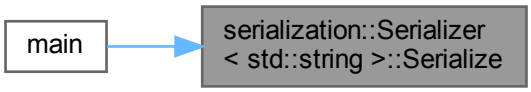
```
00456                                                {
00457         uint64_t length = obj.length();
00458         if (fwrite(&length, sizeof(uint64_t), 1, file) != 1) {
00459             detail::LogError("Failed to write string length: " + std::to_string(length));
00460             return false;
00461         }
00462         if (fwrite(obj.data(), sizeof(char), length, file) != length) {
00463             detail::LogError("Failed to write string data of length: " + std::to_string(length));
00464             return false;
00465         }
00466         return true;
00467     }
```

References serialization::detail::LogError().

Referenced by main().

Here is the call graph for this function:

Here is the caller graph for this function:



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

- serialization/include/serializers.hpp

8.27 serialization::Serializer< std::vector< T > > Class Template Reference

```
#include <serializers.hpp>
```

Public Types

- using Specialized = std::true_type

Public Member Functions

- bool Serialize (const std::vector< T > &obj, FILE *file)
- bool Deserialize (std::vector< T > &obj, FILE *file)
- uint64_t GetSerializedSize (const std::vector< T > &obj)

8.27.1 Detailed Description

template<typename T>
class serialization::Serializer< std::vector< T > >

Definition at line 490 of file serializers.hpp.

8.27.2 Member Typedef Documentation

8.27.2.1 Specialized

```
template<typename T >
using serialization::Serializer< std::vector< T > >::Specialized = std::true_type
```

Definition at line 492 of file serializers.hpp.

8.27.3 Member Function Documentation

8.27.3.1 Deserialize()

```
template<typename T >
bool serialization::Serializer< std::vector< T > >::Deserialize (
            std::vector< T > & obj,
            FILE * file )  [inline]
```

Definition at line 511 of file serializers.hpp.

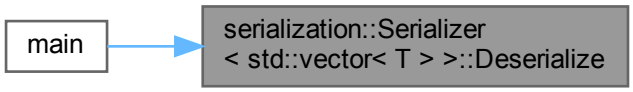
```
00511                                                   {
00512         uint64_t size;
00513         if (fread(&size, sizeof(uint64_t), 1, file) != 1) {
00514             detail::LogError("Failed to read vector size");
00515             return false;
00516         }
00517 
00518         obj.resize(size);
00519         auto item_serializer = CreateSerializer<T>();
00520         for (size_t i = 0; i < size; ++i) {
00521             if (!item_serializer->Deserialize(obj[i], file)) {
00522                 detail::LogError("Failed to deserialize vector element at index: " +
      std::to_string(i));
00523                 return false;
00524             }
00525         }
00526         return true;
00527     }
```

References serialization::detail::LogError().

Referenced by main().

Here is the call graph for this function:

Here is the caller graph for this function:



8.27.3.2 GetSerializedSize()

```
template<typename T >
uint64_t serialization::Serializer< std::vector< T > >::GetSerializedSize (
            const std::vector< T > & obj )  [inline]
```

Definition at line 529 of file serializers.hpp.

```
00529                                                       {
00530         // Size = sizeof(size field) + sum of element sizes
00531         uint64_t total_size = sizeof(uint64_t);
00532         auto item_serializer = CreateSerializer<T>();
00533         for (const auto& item : obj) {
00534             total_size += item_serializer->GetSerializedSize(item);
00535         }
00536         return total_size;
00537     }
```

8.27.3.3 Serialize()

```
template<typename T >
bool serialization::Serializer< std::vector< T > >::Serialize (
            const std::vector< T > & obj,
            FILE * file )  [inline]
```

Definition at line 494 of file serializers.hpp.

```
00494                                                       {
00495         uint64_t size = obj.size();
00496         if (fwrite(&size, sizeof(uint64_t), 1, file) != 1) {
00497             detail::LogError("Failed to write vector size: " + std::to_string(size));
00498             return false;
00499         }
```

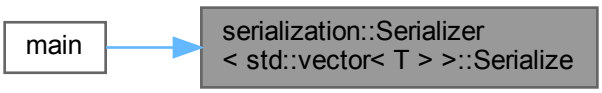
```
00500
00501         auto item_serializer = CreateSerializer<T>();
00502         for (size_t i = 0; i < obj.size(); ++i) {
00503             if (!item_serializer->Serialize(obj[i], file)) {
00504                 detail::LogError("Failed to serialize vector element at index: " + std::to_string(i));
00505                 return false;
00506             }
00507         }
00508         return true;
00509     }
```

References serialization::detail::LogError().

Referenced by main().

Here is the call graph for this function:

Here is the caller graph for this function:



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

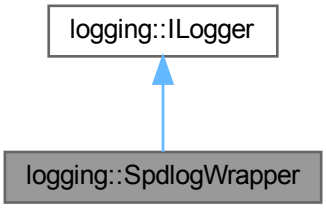
- serialization/include/serializers.hpp

8.28 logging::SpdlogWrapper Class Reference

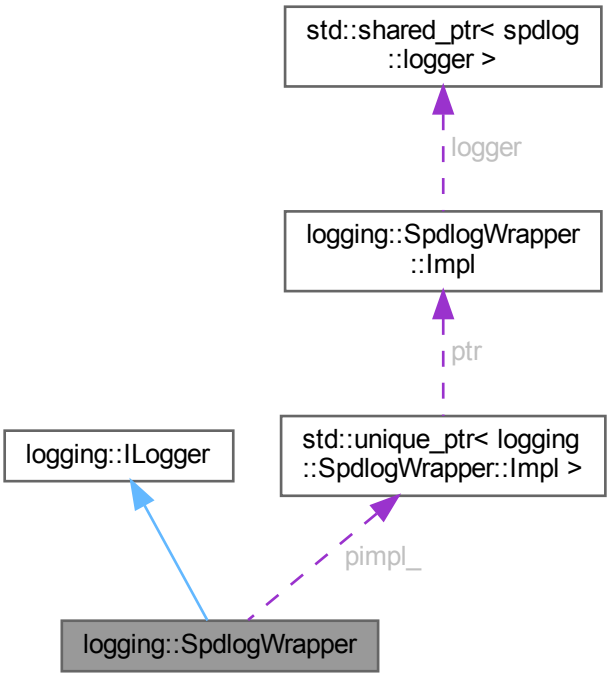
Concrete logger implementation wrapping the spdlog library.

`#include <SpdlogWrapper.hpp>`

Inheritance diagram for logging::SpdlogWrapper:



Collaboration diagram for logging::SpdlogWrapper:



Classes

- class Impl

  Implementation class hiding spdlog dependencies.

Public Member Functions

- SpdlogWrapper (const std::string &name="spdlog", SpdlogSinkType sink_type=SpdlogSinkType::Console, const std::string &filename="logs.log")

  Construct with specified sink type.
- ~SpdlogWrapper ()
- SpdlogWrapper (SpdlogWrapper &&) noexcept
- SpdlogWrapper & operator= (SpdlogWrapper &&) noexcept
- SpdlogWrapper (const SpdlogWrapper &)=delete
- SpdlogWrapper & operator= (const SpdlogWrapper &)=delete
- void LogInfo (const std::string &message) override

  Log an informational message.
- void LogWarning (const std::string &message) override

  Log a warning message.
- void LogError (const std::string &message) override

  Log an error message.

Public Member Functions inherited from logging::ILogger

- virtual ~ILogger ()=default

Private Attributes

- std::unique_ptr< Impl > pimpl_

  Forward declaration of implementation class.

8.28.1 Detailed Description

Concrete logger implementation wrapping the spdlog library.

Uses the PIMPL idiom to hide spdlog dependencies from the header, reducing compilation dependencies and allowing the logging library to be used without requiring spdlog to be installed.

Supports multiple output targets:

- Console only (default)
- File only
- Both console and file simultaneously

Definition at line 35 of file SpdlogWrapper.hpp.

8.28.2 Constructor & Destructor Documentation

8.28.2.1 SpdlogWrapper() [1/3]

```
logging::SpdlogWrapper::SpdlogWrapper (
            const std::string & name = "spdlog",
            SpdlogSinkType sink_type = SpdlogSinkType::Console,
            const std::string & filename = "logs.log" )  [explicit]
```

Construct with specified sink type.

Parameters

sink_type	Type of sink to use (Console, File, or Both)
filename	Filename for file sink (required if sink_type is File or Both)

Example:

```
// Console only
auto console_logger = std::make_shared<SpdlogWrapper>(SpdlogSinkType::Console);

// File only
auto file_logger = std::make_shared<SpdlogWrapper>(SpdlogSinkType::File, "app.log");

// Both console and file
auto both_logger = std::make_shared<SpdlogWrapper>(SpdlogSinkType::Both, "app.log");
```

Definition at line 75 of file SpdlogWrapper.cpp.

```
00077     : pimpl_(std::make_unique<Impl>(name, sink_type, filename)) {
00078 }
```

References pimpl_.

8.28.2.2 ~SpdlogWrapper()

```
logging::SpdlogWrapper::~SpdlogWrapper ( )  [default]
```

8.28.2.3 SpdlogWrapper() [2/3]

```
logging::SpdlogWrapper::SpdlogWrapper (
            SpdlogWrapper &&  )  [default], [noexcept]
```

8.28.2.4 SpdlogWrapper() [3/3]

```
logging::SpdlogWrapper::SpdlogWrapper (
            const SpdlogWrapper &  )  [delete]
```

8.28.3 Member Function Documentation

8.28.3.1 LogError()

```
void logging::SpdlogWrapper::LogError (
            const std::string & message )  [override], [virtual]
```

Log an error message.

Parameters

message	The message to log

Implements logging::ILogger.

Definition at line 94 of file SpdlogWrapper.cpp.

```
00094                                                   {
00095     pimpl_->logger->error(message);
00096 }
```

References pimpl_.

8.28.3.2 LogInfo()

```
void logging::SpdlogWrapper::LogInfo (
            const std::string & message )  [override], [virtual]
```

Log an informational message.

Parameters

message	The message to log

Implements logging::ILogger.

Definition at line 86 of file SpdlogWrapper.cpp.

```
00086                                                   {
00087     pimpl_->logger->info(message);
00088 }
```

References pimpl_.

8.28.3.3 LogWarning()

```
void logging::SpdlogWrapper::LogWarning (
            const std::string & message )  [override], [virtual]
```

Log a warning message.

Parameters

message	The message to log

Implements logging::ILogger.

Definition at line 90 of file SpdlogWrapper.cpp.

```
00090                                                   {
00091     pimpl_->logger->warn(message);
00092 }
```

References pimpl_.

8.28.3.4 operator=() [1/2]

```
SpdlogWrapper & logging::SpdlogWrapper::operator= (
            const SpdlogWrapper &  )  [delete]
```

8.28.3.5 operator=() [2/2]

```
SpdlogWrapper & logging::SpdlogWrapper::operator= (
            SpdlogWrapper &&  )  [default], [noexcept]
```

8.28.4 Member Data Documentation

8.28.4.1 pimpl_

```
std::unique_ptr<Impl> logging::SpdlogWrapper::pimpl_  [private]
```

Forward declaration of implementation class.

Pointer to implementation

Definition at line 84 of file SpdlogWrapper.hpp.

Referenced by LogError(), LogInfo(), LogWarning(), and SpdlogWrapper().

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

- logging/include/SpdlogWrapper.hpp
- logging/src/SpdlogWrapper.cpp

8.29 io::TempFileManager Class Reference

A manager for temporary files.

```
#include <temp_file_manager.hpp>
```

Public Member Functions

- TempFileManager (const std::string &base_dir_name="ts")
  Constructs a new TempFileManager object.
- ~TempFileManager ()
  Destructor that cleans up all temporary files.
- TempFileManager (const TempFileManager &)=delete
- TempFileManager & operator= (const TempFileManager &)=delete
- TempFileManager (TempFileManager &&)=default
- TempFileManager & operator= (TempFileManager &&)=default
- std::string GenerateTempFilename (const std::string &prefix="tmp", const std::string &extension=".b")
  Generates a unique temporary filename.
- void CleanupFile (const std::string &filename_str)
  Deletes the specified file.
- const std::filesystem::path & GetBaseDirPath () const
  Returns the path to the base directory.

Private Attributes

- std::filesystem::path base_temp_dir_path_

  Base directory for temporary files.
- uint64_t temp_file_counter_ = 0

  Counter for generating unique names.
- bool owns_directory_ {}

  Flag to indicate if this instance owns the directory.

8.29.1 Detailed Description

A manager for temporary files.

The class is responsible for creating, tracking, and deleting temporary files used during the external sorting process.

Definition at line 22 of file temp_file_manager.hpp.

8.29.2 Constructor & Destructor Documentation

8.29.2.1 TempFileManager() [1/3]

```
io::TempFileManager::TempFileManager (
            const std::string & base_dir_name = "ts" )  [explicit]
```

Constructs a new TempFileManager object.

Parameters

base_dir_name	The name of the base directory for temporary files

Definition at line 68 of file temp_file_manager.hpp.

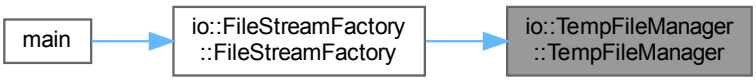
```
00068                                                                   {
00069     base_temp_dir_path_ = std::filesystem::current_path() / base_dir_name;
00070     if (!std::filesystem::exists(base_temp_dir_path_)) {
00071         std::error_code ec;
00072         std::filesystem::create_directories(base_temp_dir_path_, ec);
00073         if (ec) {
00074             throw std::runtime_error("TempFileManager failed to create temp directory: " +
00075                                      base_temp_dir_path_.string() + " Error: " + ec.message());
00076         }
00077         owns_directory_ = true;
00078         detail::LogInfo("TempFileManager created temporary directory: " +
      base_temp_dir_path_.string());
00079     } else {
00080         owns_directory_ = false;
00081         detail::LogInfo("TempFileManager using existing temporary directory: "
00082                         + base_temp_dir_path_.string());
00083     }
00084 }
```

References base_temp_dir_path_, io::detail::LogInfo(), and owns_directory_.

Referenced by io::FileStreamFactory< T >::FileStreamFactory().

Here is the call graph for this function:

Here is the caller graph for this function:



8.29.2.2 ~TempFileManager()

```
io::TempFileManager::~TempFileManager ( )
```

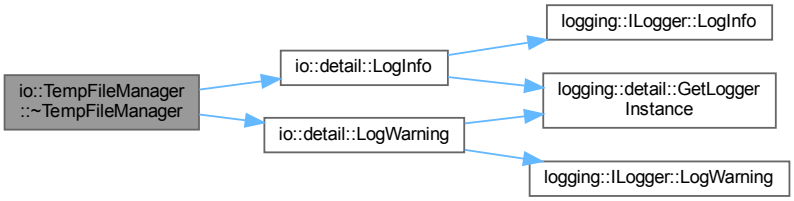
Destructor that cleans up all temporary files.

Definition at line 86 of file temp_file_manager.hpp.

```
00086                                   {
00087     detail::LogInfo("TempFileManager destructor: Attempting to clean up temporary directory...");
00088     if (owns_directory_ && std::filesystem::exists(base_temp_dir_path_)) {
00089         std::error_code ec;
00090         std::filesystem::remove_all(base_temp_dir_path_, ec);
00091         if (ec) {
00092             detail::LogWarning("Warning: TempFileManager failed to remove_all temp directory "
00093                                + base_temp_dir_path_.string() + ": " + ec.message());
00094         } else {
00095             detail::LogInfo("TempFileManager removed temp directory and all its contents: "
00096                             + base_temp_dir_path_.string());
00097         }
00098     } else if (std::filesystem::exists(base_temp_dir_path_)) {
00099         detail::LogInfo("TempFileManager: Temporary directory "
00100                         + base_temp_dir_path_.string()
00101                         + " exists but was not created by this instance, not removing.");
00102     }
00103 }
```

References base_temp_dir_path_, io::detail::LogInfo(), io::detail::LogWarning(), and owns_directory_.

Here is the call graph for this function:



8.29.2.3 TempFileManager() [2/3]

```
io::TempFileManager::TempFileManager (
            const TempFileManager &  )  [delete]
```

8.29.2.4 TempFileManager() [3/3]

```
io::TempFileManager::TempFileManager (
            TempFileManager &&  )  [default]
```

8.29.3 Member Function Documentation

8.29.3.1 CleanupFile()

```
void io::TempFileManager::CleanupFile (
            const std::string & filename_str )
```

Deletes the specified file.

Parameters

filename_str	Path to the file to be deleted

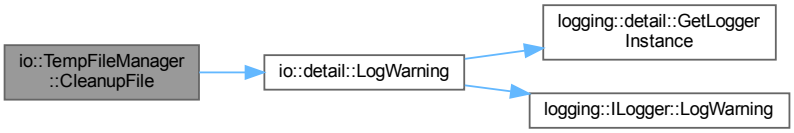
Definition at line 113 of file temp_file_manager.hpp.

```
00113                                                                   {
00114     std::filesystem::path file_path(filename_str);
00115     if (std::filesystem::exists(file_path)) {
00116         std::error_code ec;
00117         std::filesystem::remove(file_path, ec);
00118         if (ec) {
00119             detail::LogWarning("CleanupFile failed to remove " + filename_str + ": "
00120                                + ec.message());
00121         }
00122     }
00123 }
```

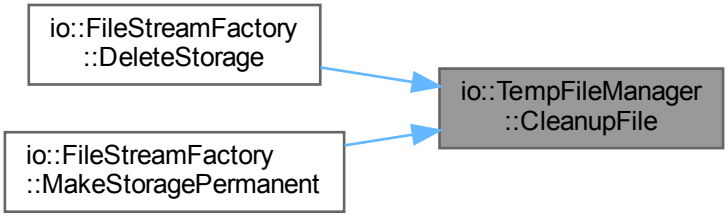
References io::detail::LogWarning().

Referenced by io::FileStreamFactory< T >::DeleteStorage(), and io::FileStreamFactory< T >::MakeStoragePermanent().

Here is the call graph for this function:



Here is the caller graph for this function:



8.29.3.2 GenerateTempFilename()

```
std::string io::TempFileManager::GenerateTempFilename (
            const std::string & prefix = "tmp",
            const std::string & extension = ".b" )
```

Generates a unique temporary filename.

Parameters

prefix	The prefix for the filename
extension	The file extension

Returns

The full path to the temporary file

Definition at line 105 of file temp_file_manager.hpp.

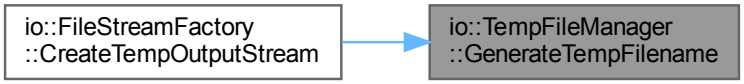
```
00106                                                                                {
00107     std::filesystem::path file_path =
00108         base_temp_dir_path_ / (prefix + std::to_string(temp_file_counter_++) + extension);
00109     detail::LogInfo("TempFileManager generated temp filename: " + file_path.string());
00110     return file_path.string();
00111 }
```

References base_temp_dir_path_, io::detail::LogInfo(), and temp_file_counter_.

Referenced by io::FileStreamFactory< T >::CreateTempOutputStream().

Here is the call graph for this function:

Here is the caller graph for this function:



8.29.3.3 GetBaseDirPath()

```
const std::filesystem::path & io::TempFileManager::GetBaseDirPath ( ) const
```

Returns the path to the base directory.

Returns

A const reference to the base directory path

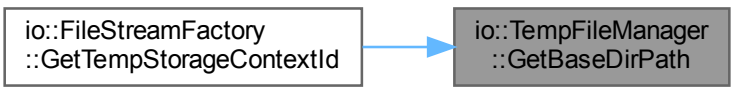
Definition at line 125 of file temp_file_manager.hpp.

```
00125                                                       {
00126     return base_temp_dir_path_;
00127 }
```

References base_temp_dir_path_.

Referenced by io::FileStreamFactory< T >::GetTempStorageContextId().

Here is the caller graph for this function:



8.29.3.4 operator=() [1/2]

```
TempFileManager & io::TempFileManager::operator= (
            const TempFileManager &  )  [delete]
```

8.29.3.5 operator=() [2/2]

```
TempFileManager & io::TempFileManager::operator= (
            TempFileManager &&  )  [default]
```

8.29.4 Member Data Documentation

8.29.4.1 base_temp_dir_path_

```
std::filesystem::path io::TempFileManager::base_temp_dir_path_  [private]
```

Base directory for temporary files.

Definition at line 24 of file temp_file_manager.hpp.

Referenced by GenerateTempFilename(), GetBaseDirPath(), TempFileManager(), and ~TempFileManager().

8.29.4.2 owns_directory_

```
bool io::TempFileManager::owns_directory_ {}  [private]
```

Flag to indicate if this instance owns the directory.

Definition at line 26 of file temp_file_manager.hpp.

```
00026 {};
```

Referenced by TempFileManager(), and ~TempFileManager().

8.29.4.3 temp_file_counter_

```
uint64_t io::TempFileManager::temp_file_counter_ = 0  [private]
```

Counter for generating unique names.

Definition at line 25 of file temp_file_manager.hpp.

Referenced by GenerateTempFilename().

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

- io/include/temp_file_manager.hpp

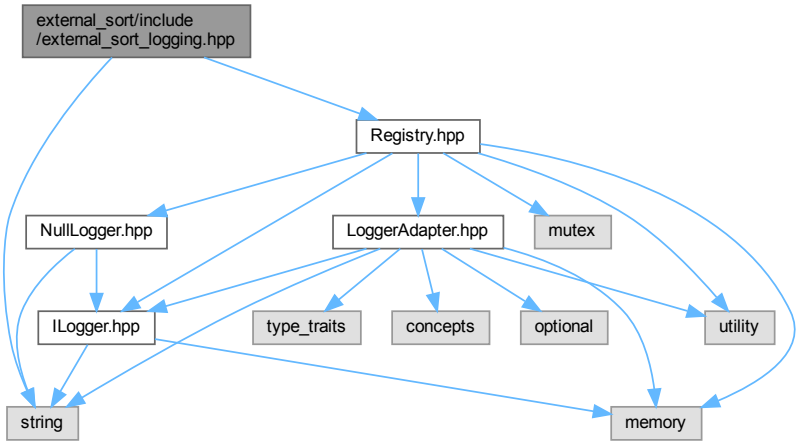
Chapter 9

File Documentation

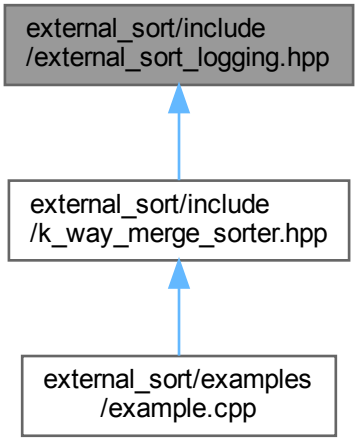
9.1 external_sort/include/external_sort_logging.hpp File Reference

```
#include "Registry.hpp"
#include <string>
```

Include dependency graph for external_sort_logging.hpp:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace external_sort
- namespace external_sort::detail

Functions

- void external_sort::detail::LogError (const std::string &message)

  Log an error message from External Sort operations.
- void external_sort::detail::LogWarning (const std::string &message)

  Log a warning message from io operations.
- void external_sort::detail::LogInfo (const std::string &message)

  Log an informational message from External Sort operations.

9.2 external_sort_logging.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file io_logging.hpp
00003  * @brief Logging integration for External Sort library
00004  */
00005
00006 #pragma once
00007
00008 #include "Registry.hpp"
00009 #include <string>
00010
00011 namespace external_sort {
00012 namespace detail {
00013
00014 /**
00015  * @brief Log an error message from External Sort operations
00016  * @param message The error message External Sort log
00017  */
```

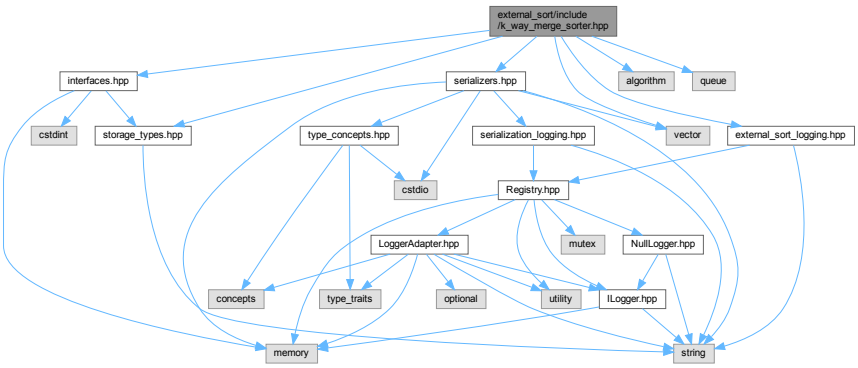
```
00018 inline void LogError(const std::string& message) {
00019     auto& logger = logging::detail::GetLoggerInstance();
00020     if (logger) {
00021         logger->LogError("[External Sort] " + message);
00022     }
00023 }
00024
00025 /**
00026  * @brief Log a warning message from io operations
00027  * @param message The warning message External Sort log
00028  */
00029 inline void LogWarning(const std::string& message) {
00030     auto& logger = logging::detail::GetLoggerInstance();
00031     if (logger) {
00032         logger->LogWarning("[External Sort] " + message);
00033     }
00034 }
00035
00036 /**
00037  * @brief Log an informational message from External Sort operations
00038  * @param message The info message to log
00039  */
00040 inline void LogInfo(const std::string& message) {
00041     auto& logger = logging::detail::GetLoggerInstance();
00042     if (logger) {
00043         logger->LogInfo("[External Sort] " + message);
00044     }
00045 }
00046
00047 }  // namespace detail
00048 }  // namespace external_sort
```

9.3 external_sort/include/k_way_merge_sorter.hpp File Reference

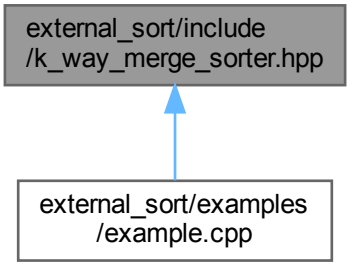
K-way external merge sort.

```
#include "external_sort_logging.hpp"
#include "interfaces.hpp"
#include "serializers.hpp"
#include "storage_types.hpp"
#include <algorithm>
#include <queue>
#include <vector>
```

Include dependency graph for k_way_merge_sorter.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- struct external_sort::MergeSource< T >

  Structure representing a data source for merging.
- struct external_sort::MergeSourceComparatorStatic< T >

  Comparator for merge sources that compares the current value of the streams.
- class external_sort::KWayMergeSorter< T >

  Template class for performing k-way external merge sort.
- struct external_sort::KWayMergeSorter< T >::MergeSourceComparator

  Comparator for merge sources.

Namespaces

- namespace external_sort

9.3.1 Detailed Description

K-way external merge sort.

Definition in file k_way_merge_sorter.hpp.

9.4 k_way_merge_sorter.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file k_way_merge_sorter.hpp
00003  * @brief K-way external merge sort
00004  */
00005
00006 #pragma once
00007
00008 #include "external_sort_logging.hpp"
00009 #include "interfaces.hpp"
00010 #include "serializers.hpp"
00011 #include "storage_types.hpp"
```

```
00012
00013 #include <algorithm>
00014 #include <queue>
00015 #include <vector>
00016
00017 namespace external_sort {
00018
00019 /**
00020  * @brief Structure representing a data source for merging
00021  *
00022  * Used in priority queue for k-way merging.
00023  *
00024  * @tparam T Type of data elements
00025  */
00026 template <typename T>
00027 struct MergeSource {
00028     io::IInputStream<T>* stream;  ///< Pointer to source stream
00029 };
00030
00031 /**
00032  * @brief Comparator for merge sources that compares the current value of the streams
00033  */
00034 template <typename T>
00035 struct MergeSourceComparatorStatic {
00036     bool ascending;
00037     explicit MergeSourceComparatorStatic(bool asc) : ascending(asc) {
00038     }
00039
00040     bool operator()(const MergeSource<T>& a, const MergeSource<T>& b) const {
00041         const T& va = a.stream->Value();
00042         const T& vb = b.stream->Value();
00043         return ascending ? (va > vb) : (va < vb);
00044     }
00045 };
00046
00047 /**
00048  * @brief Template class for performing k-way external merge sort
00049  *
00050  * The class is abstracted from specific storage implementation and works
00051  * through IStreamFactory, io::IInputStream and io::IOutputStream interfaces.
00052  *
00053  * @tparam T Type of elements to sort
00054  */
00055 template <typename T>
00056 class KWayMergeSorter {
00057 private:
00058     io::IStreamFactory<T>& stream_factory_;  ///< Reference to stream factory
00059     io::StorageId input_id_;                 ///< Input storage identifier
00060     io::StorageId output_id_;                ///< Output storage identifier
00061     uint64_t memory_for_runs_bytes_;         ///< Memory for creating initial runs (bytes)
00062     uint64_t k_way_degree_;                  ///< Degree of k-way merging
00063     uint64_t file_io_buffer_elements_;       ///< Size of I/O buffer in elements
00064     bool ascending_;                         ///< Ascending sort flag
00065
00066     /**
00067      * @brief Comparator for merge sources
00068      */
00069     struct MergeSourceComparator {
00070         bool ascending;  ///< Sort direction flag
00071
00072         /**
00073          * @brief Comparator constructor
00074          * @param asc Sort direction (true - ascending)
00075          */
00076         explicit MergeSourceComparator(bool asc) : ascending(asc) {
00077         }
00078
00079         /**
00080          * @brief Comparison operator
00081          * @param a First source
00082          * @param b Second source
00083          * @return Comparison result for priority queue
00084          */
00085         bool operator()(const MergeSource<T>& a, const MergeSource<T>& b) const {
00086             const T& va = a.stream->Value();
00087             const T& vb = b.stream->Value();
00088             return ascending ? (va > vb) : (va < vb);
00089         }
00090     };
00091
00092     /**
00093      * @brief Creates initial sorted runs from input stream
00094      * @return Vector of created run identifiers
00095      */
00096     std::vector<io::StorageId> CreateInitialRuns();
00097
00098     /**
```

```
00099      * @brief Merges a group of runs into one
00100      * @param group_run_ids Identifiers of runs to merge
00101      * @param output_run_id Output run identifier
00102      */
00103     void MergeGroupOfRuns(const std::vector<io::StorageId>& group_run_ids,
00104                           const io::StorageId& output_run_id);
00105
00106 public:
00107     /**
00108      * @brief Constructor for k-way merge sorter
00109      * @param factory Reference to stream factory
00110      * @param input_id Input storage identifier
00111      * @param output_id Output storage identifier
00112      * @param mem_bytes Memory for creating runs in bytes
00113      * @param k_degree Degree of k-way merging (minimum 2)
00114      * @param io_buf_elems Size of I/O buffer in elements
00115      * @param sort_ascending Sort direction (default ascending)
00116      * @throws std::invalid_argument if k_degree < 2
00117      * @throws std::runtime_error if output_id is in temporary storage context
00118      */
00119     KWayMergeSorter(io::IStreamFactory<T>& factory, io::StorageId input_id, io::StorageId output_id,
00120                     uint64_t mem_bytes, uint64_t k_degree, uint64_t io_buf_elems,
00121                     bool sort_ascending = true);
00122
00123     /**
00124      * @brief Performs sorting
00125      *
00126      * Main method that performs complete k-way external sorting:
00127      * 1. Creates initial sorted runs
00128      * 2. Repeatedly merges runs until one remains
00129      * 3. Makes final run permanent with output_id name
00130      */
00131     void Sort();
00132 };
00133
00134 /**
00135  * @note Performance tip: When sorting non-trivial or move-only types prefer
00136  * using stream implementations that support TakeValue() and rvalue Write(T&&).
00137  * The sorter uses these methods internally (when available) to minimize copies
00138  * by moving elements between streams and temporary buffers.
00139  */
00140 template <typename T>
00141 std::vector<io::StorageId> KWayMergeSorter<T>::CreateInitialRuns() {
00142     detail::LogInfo("»» CreateInitialRuns started for input: '" + input_id_ +
00143                     "' with memory limit: " + std::to_string(memory_for_runs_bytes_ / (1024.0 *
      1024.0)) + " MB");
00144     detail::LogInfo("    sizeof(T) = " + std::to_string(sizeof(T)) + " bytes");
00145
00146     std::unique_ptr<io::IInputStream<T> input_stream =
00147         stream_factory_.CreateInputStream(input_id_, file_io_buffer_elements_);
00148
00149     if (input_stream->IsEmptyOriginalStorage()) {
00150         detail::LogInfo("KWayMergeSorter: Input storage is empty. No runs created.");
00151         return {};
00152     }
00153
00154     auto serializer = serialization::CreateSerializer<T>();
00155     std::vector<io::StorageId> run_ids;
00156     std::vector<T> run_buffer;
00157     int run_counter = 1;
00158
00159     while (!input_stream->IsExhausted()) {
00160         run_buffer.clear();
00161         uint64_t current_run_mem_usage = 0;
00162         int element_counter = 0;
00163
00164         detail::LogInfo("--- Starting Run #" + std::to_string(run_counter++) + " ---");
00165
00166         while (!input_stream->IsExhausted()) {
00167             const T& current_element = input_stream->Value();
00168
00169             uint64_t serialized_size = serializer->GetSerializedSize(current_element);
00170             uint64_t estimated_element_footprint;
00171
00172             if constexpr (std::is_trivial_v<T> && std::is_standard_layout_v<T>) {
00173                 estimated_element_footprint = sizeof(T);
00174             } else {
00175                 estimated_element_footprint = serialized_size + sizeof(T);
00176             }
00177
00178             detail::LogInfo("[Loop " + std::to_string(element_counter++) + "]: "
00179                             "current_mem=" + std::to_string(current_run_mem_usage) +
00180                             ", next_elem_footprint=" + std::to_string(estimated_element_footprint) +
00181                             " (ser_size=" + std::to_string(serialized_size) + ")" +
00182                             ", limit=" + std::to_string(memory_for_runs_bytes_) +
00183                             ", vec_size=" + std::to_string(run_buffer.size()) +
00184                             ", vec_cap=" + std::to_string(run_buffer.capacity()));
```

```
00185
00186             if (run_buffer.empty()) {
00187                 if (estimated_element_footprint > memory_for_runs_bytes_) {
00188                     throw std::runtime_error("KWayMergeSorter: Memory limit is too small for a single
      element.");
00189                 }
00190             }
00191             else if (current_run_mem_usage + estimated_element_footprint > memory_for_runs_bytes_) {
00192                 detail::LogInfo("    Limit reached. Breaking loop to finalize run.");
00193                 break;
00194             }
00195
00196             current_run_mem_usage += estimated_element_footprint;
00197             run_buffer.push_back(input_stream->TakeValue());
00198             input_stream->Advance();
00199         }
00200
00201         if (!run_buffer.empty()) {
00202             detail::LogInfo("    Sorting " + std::to_string(run_buffer.size()) + " elements...");
00203             if (ascending_) {
00204                 std::sort(run_buffer.begin(), run_buffer.end());
00205             } else {
00206                 std::sort(run_buffer.begin(), run_buffer.end(), std::greater<T>());
00207             }
00208
00209             io::StorageId run_id;
00210             std::unique_ptr<io::IOutputStream<T» out_run =
00211                 stream_factory_.CreateTempOutputStream(run_id, file_io_buffer_elements_);
00212             for (T& val : run_buffer) {
00213                 out_run->Write(std::move(val));
00214             }
00215             out_run->Finalize();
00216             run_ids.push_back(std::move(run_id));
00217             detail::LogInfo("    Run created: " + run_id + " with " +
00218                             std::to_string(out_run->GetTotalElementsWritten()) +
00219                             " elements. Estimated mem usage: " + std::to_string(current_run_mem_usage)
      + " bytes.");
00220         }
00221     }
00222     return run_ids;
00223 }
00224
00225 template <typename T>
00226 void KWayMergeSorter<T>::MergeGroupOfRuns(const std::vector<io::StorageId>& group_run_ids,
00227                                           const io::StorageId& output_run_id) {
00228     detail::LogInfo("KWayMergeSorter: Merging " + std::to_string(group_run_ids.size()) +
00229                     " runs into " + output_run_id);
00230
00231     std::priority_queue<MergeSource<T>, std::vector<MergeSource<T», MergeSourceComparator> pq{
00232         MergeSourceComparator(ascending_)};
00233     std::vector<std::unique_ptr<io::IInputStream<T»> input_streams_store;
00234     input_streams_store.reserve(group_run_ids.size());
00235
00236     for (const auto& run_id : group_run_ids) {
00237         auto stream_ptr = stream_factory_.CreateInputStream(run_id, file_io_buffer_elements_);
00238         if (!stream_ptr->IsExhausted()) {
00239             pq.push({stream_ptr.get()});
00240         }
00241         input_streams_store.push_back(std::move(stream_ptr));
00242     }
00243
00244     std::unique_ptr<io::IOutputStream<T» output_stream =
00245         stream_factory_.CreateOutputStream(output_run_id, file_io_buffer_elements_);
00246
00247     while (!pq.empty()) {
00248         MergeSource<T> current_source = pq.top();
00249         pq.pop();
00250         T val = current_source.stream->TakeValue();
00251         output_stream->Write(std::move(val));
00252         current_source.stream->Advance();
00253         if (!current_source.stream->IsExhausted()) {
00254             pq.push({current_source.stream});
00255         }
00256     }
00257     output_stream->Finalize();
00258     detail::LogInfo("KWayMergeSorter: Merged group into " + output_run_id + " with " +
00259                     std::to_string(output_stream->GetTotalElementsWritten()) + " elements.");
00260 }
00261
00262 template <typename T>
00263 KWayMergeSorter<T>::KWayMergeSorter(io::IStreamFactory<T>& factory, io::StorageId input_id,
00264                                     io::StorageId output_id, uint64_t mem_bytes, uint64_t k_degree,
00265                                     uint64_t io_buf_elems, bool sort_ascending)
00266     : stream_factory_(factory),
00267       input_id_(std::move(input_id)),
00268       output_id_(std::move(output_id)),
00269       memory_for_runs_bytes_(mem_bytes),
```

```
00270       k_way_degree_(k_degree),
00271       file_io_buffer_elements_(io_buf_elems),
00272       ascending_(sort_ascending) {
00273     if (k_way_degree_ < 2) {
00274         throw std::invalid_argument("KWayMergeSorter: k_way_degree must be at least 2.");
00275     }
00276     io::StorageId temp_context_id = stream_factory_.GetTempStorageContextId();
00277     if (!temp_context_id.empty() && output_id_.rfind(temp_context_id, 0) == 0 &&
00278         output_id_.length() > temp_context_id.length()) {
00279         throw std::runtime_error("KWayMergeSorter: Output storage ID '" + output_id_ +
00280                                  "' seems to be inside the temporary storage context '" +
00281                                  temp_context_id + "'.");
00282     }
00283 }
00284
00285 template <typename T>
00286 void KWayMergeSorter<T>::Sort() {
00287     std::vector<io::StorageId> current_run_ids = CreateInitialRuns();
00288
00289     if (current_run_ids.empty()) {
00290         detail::LogInfo("KWayMergeSorter: No initial runs. Creating empty output " + output_id_);
00291         auto empty_out = stream_factory_.CreateOutputStream(output_id_, file_io_buffer_elements_);
00292         empty_out->Finalize();
00293         return;
00294     }
00295
00296     std::vector<io::StorageId> runs_to_delete_this_pass;
00297     while (current_run_ids.size() > 1) {
00298         std::vector<io::StorageId> next_pass_run_ids;
00299         runs_to_delete_this_pass.clear();
00300         detail::LogInfo("KWayMergeSorter: Merge pass with " +
00301                         std::to_string(current_run_ids.size()) + " runs.");
00302
00303         for (uint64_t i = 0; i < current_run_ids.size(); i += k_way_degree_) {
00304             std::vector<io::StorageId> group_to_merge;
00305             for (uint64_t j = 0; j < k_way_degree_ && (i + j) < current_run_ids.size(); ++j) {
00306                 group_to_merge.push_back(current_run_ids[i + j]);
00307             }
00308             if (group_to_merge.empty()) {
00309                 continue;
00310             }
00311
00312             io::StorageId merged_run_id;
00313             bool is_final_merge_to_output = (current_run_ids.size() <= k_way_degree_) && (i == 0);
00314
00315             if (is_final_merge_to_output) {
00316                 merged_run_id = output_id_;
00317                 detail::LogInfo("KWayMergeSorter: Merging to final output: " + output_id_);
00318             } else {
00319                 auto temp_stream =
00320                     stream_factory_.CreateTempOutputStream(merged_run_id, file_io_buffer_elements_);
00321                 temp_stream->Finalize();
00322             }
00323
00324             MergeGroupOfRuns(group_to_merge, merged_run_id);
00325             next_pass_run_ids.push_back(std::move(merged_run_id));
00326
00327             for (const auto& id_to_del : group_to_merge) {
00328                 runs_to_delete_this_pass.push_back(id_to_del);
00329             }
00330         }
00331         current_run_ids = next_pass_run_ids;
00332         for (const auto& id_del : runs_to_delete_this_pass) {
00333             if (id_del != output_id_) {
00334                 stream_factory_.DeleteStorage(id_del);
00335             }
00336         }
00337     }
00338
00339     if (current_run_ids.size() == 1) {
00340         if (current_run_ids[0] != output_id_) {
00341             detail::LogInfo("KWayMergeSorter: Finalizing " + current_run_ids[0] + " as " +
00342                             output_id_);
00343             stream_factory_.MakeStoragePermanent(current_run_ids[0], output_id_);
00344         } else {
00345             detail::LogInfo("KWayMergeSorter: Output is already in " + output_id_);
00346         }
00347     } else if (current_run_ids.empty() && !stream_factory_.StorageExists(output_id_)) {
00348         detail::LogInfo("KWayMergeSorter: No runs left and output " + output_id_ +
00349                         " does not exist. Creating empty.");
00350         auto empty_out = stream_factory_.CreateOutputStream(output_id_, file_io_buffer_elements_);
00351         empty_out->Finalize();
00352     } else if (!current_run_ids.empty()) {
00353         detail::LogError("Error: KWayMergeSorter finished with unexpected runs: " +
00354                          std::to_string(current_run_ids.size()));
00355     }
00356 }
```

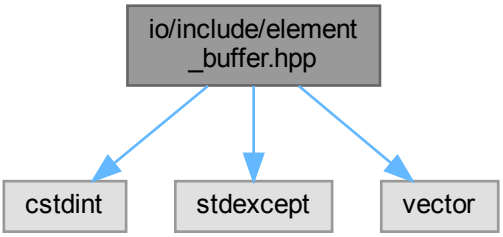
```
00357
00358 }  // namespace external_sort
```

9.5 io/include/element_buffer.hpp File Reference

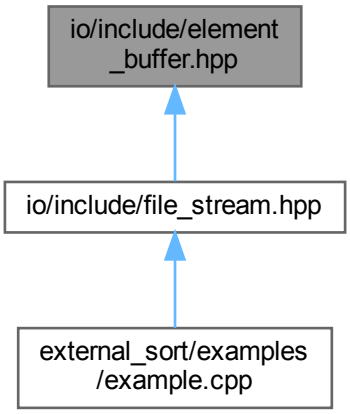
Element buffer template class.

```
#include <cstdint>
#include <stdexcept>
#include <vector>
```

Include dependency graph for element_buffer.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class io::ElementBuffer< T >

  A template class for a buffer of elements of type T.

Namespaces

- namespace io

9.5.1 Detailed Description

Element buffer template class.

Version

1.0

Definition in file element_buffer.hpp.

9.6 element_buffer.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file element_buffer.hpp
00003  * @brief Element buffer template class
00004  * @version 1.0
00005  */
00006
00007 #pragma once
00008
00009 #include <cstdint>
00010 #include <stdexcept>
00011 #include <vector>
00012
00013 namespace io {
00014
00015 /**
00016  * @brief A template class for a buffer of elements of type T
00017  *
00018  * ElementBuffer provides in-memory buffering for reading or writing
00019  * a sequence of elements. It manages the capacity, the current number
00020  * of valid elements, and the read cursor.
00021  *
00022  * @tparam T The type of elements stored in the buffer
00023  */
00024 template <typename T>
00025 class ElementBuffer {
00026 private:
00027     std::vector<T> storage_{};       ///< Internal storage for elements
00028     uint64_t capacity_elements_{};   ///< Maximum buffer capacity in elements
00029     uint64_t num_valid_elements_{};  ///< Current number of valid (filled) elements in the buffer
00030     uint64_t read_cursor_{};         ///< Cursor position for the next read from the buffer
00031
00032 public:
00033     /**
00034      * @brief Constructs a buffer with a given capacity
00035      * @param capacity The buffer capacity in elements (minimum 1)
00036      */
00037     explicit ElementBuffer(uint64_t capacity);
00038
00039     /**
00040      * @brief Adds an element to the end of the buffer
00041      * @param element The element to add
00042      * @return true if the buffer becomes full after adding the element
00043      */
00044     bool PushBack(const T& element);
00045
00046     /**
00047      * @brief Adds an element to the end of the buffer (move overload)
00048      * @param element The element to add (moved)
00049      * @return true if the buffer becomes full after adding the element
00050      */
00051     bool PushBack(T&& element);
00052
00053     /**
00054      * @brief Returns a pointer to the buffer's data (read-only)
```

```
00055      * @return A const pointer to the beginning of the data
00056      */
00057     [[nodiscard]] const T* Data() const;
00058
00059     /**
00060      * @brief Returns the number of valid elements in the buffer
00061      * @return The number of valid elements
00062      */
00063     [[nodiscard]] uint64_t Size() const;
00064
00065     /**
00066      * @brief Returns a pointer to the raw buffer data (for writing)
00067      * @return A pointer to the beginning of the data
00068      */
00069     T* RawDataPtr();
00070
00071     /**
00072      * @brief Sets the number of valid elements in the buffer
00073      * @param count The new number of valid elements
00074      * @throws std::length_error if count exceeds the buffer's capacity
00075      */
00076     void SetValidElementsCount(uint64_t count);
00077
00078     /**
00079      * @brief Reads the next element from the buffer and advances the cursor
00080      *
00081      * For non-trivial types this function returns the element by value and may use
00082      * move semantics to avoid copies. Callers that only need the element once
00083      * should accept the returned value by value to enable moves.
00084      * @return The next element, or T{} if there are no more elements
00085      */
00086     T ReadNext();
00087
00088     /**
00089      * @brief Checks if there are more elements to read
00090      * @return true if there are more elements to read
00091      */
00092     [[nodiscard]] bool HasMoreToRead() const;
00093
00094     /**
00095      * @brief Returns the maximum capacity of the buffer
00096      * @return The buffer capacity in elements
00097      */
00098     [[nodiscard]] uint64_t Capacity() const;
00099
00100     /**
00101      * @brief Checks if the buffer is empty
00102      * @return true if there are no valid elements in the buffer
00103      */
00104     [[nodiscard]] bool IsEmpty() const;
00105
00106     /**
00107      * @brief Checks if the buffer is full
00108      * @return true if the buffer is filled to its maximum capacity
00109      */
00110     [[nodiscard]] bool IsFull() const;
00111
00112     /**
00113      * @brief Clears the buffer, resetting the count of valid elements and the read cursor
00114      */
00115     void Clear();
00116
00117     ElementBuffer(ElementBuffer&&) = default;
00118     ElementBuffer& operator=(ElementBuffer&&) = default;
00119     ElementBuffer(const ElementBuffer&) = delete;
00120     ElementBuffer& operator=(const ElementBuffer&) = delete;
00121     ~ElementBuffer() = default;
00122 };
00123
00124 }  // namespace io
00125
00126 namespace io {
00127
00128 template <typename T>
00129 ElementBuffer<T>::ElementBuffer(uint64_t capacity)
00130     : capacity_elements_(std::max(static_cast<uint64_t>(1), capacity)),
00131       num_valid_elements_(0),
00132       read_cursor_(0) {
00133     storage_.resize(capacity_elements_);
00134 }
00135
00136 template <typename T>
00137 bool ElementBuffer<T>::PushBack(const T& element) {
00138     if (num_valid_elements_ < capacity_elements_) {
00139         storage_[num_valid_elements_] = element;
00140         num_valid_elements_++;
00141         return num_valid_elements_ == capacity_elements_;
```

```
00142     }
00143     return true;
00144 }
00145
00146 template <typename T>
00147 bool ElementBuffer<T>::PushBack(T&& element) {
00148     if (num_valid_elements_ < capacity_elements_) {
00149         storage_[num_valid_elements_] = std::move(element);
00150         num_valid_elements_++;
00151         return num_valid_elements_ == capacity_elements_;
00152     }
00153     return true;
00154 }
00155
00156 template <typename T>
00157 const T* ElementBuffer<T>::Data() const {
00158     return storage_.data();
00159 }
00160
00161 template <typename T>
00162 uint64_t ElementBuffer<T>::Size() const {
00163     return num_valid_elements_;
00164 }
00165
00166 template <typename T>
00167 T* ElementBuffer<T>::RawDataPtr() {
00168     return storage_.data();
00169 }
00170
00171 template <typename T>
00172 void ElementBuffer<T>::SetValidElementsCount(uint64_t count) {
00173     if (count > capacity_elements_) {
00174         throw std::length_error("ElementBuffer: Count exceeds capacity.");
00175     }
00176     num_valid_elements_ = count;
00177     read_cursor_ = 0;
00178 }
00179
00180 template <typename T>
00181 T ElementBuffer<T>::ReadNext() {
00182     if (read_cursor_ < num_valid_elements_) {
00183         return std::move(storage_[read_cursor_++]);
00184     }
00185     return T{};
00186 }
00187
00188 template <typename T>
00189 bool ElementBuffer<T>::HasMoreToRead() const {
00190     return read_cursor_ < num_valid_elements_;
00191 }
00192
00193 template <typename T>
00194 uint64_t ElementBuffer<T>::Capacity() const {
00195     return capacity_elements_;
00196 }
00197
00198 template <typename T>
00199 bool ElementBuffer<T>::IsEmpty() const {
00200     return num_valid_elements_ == 0;
00201 }
00202
00203 template <typename T>
00204 bool ElementBuffer<T>::IsFull() const {
00205     return num_valid_elements_ == capacity_elements_;
00206 }
00207
00208 template <typename T>
00209 void ElementBuffer<T>::Clear() {
00210     num_valid_elements_ = 0;
00211     read_cursor_ = 0;
00212 }
00213
00214 }  // namespace io
```

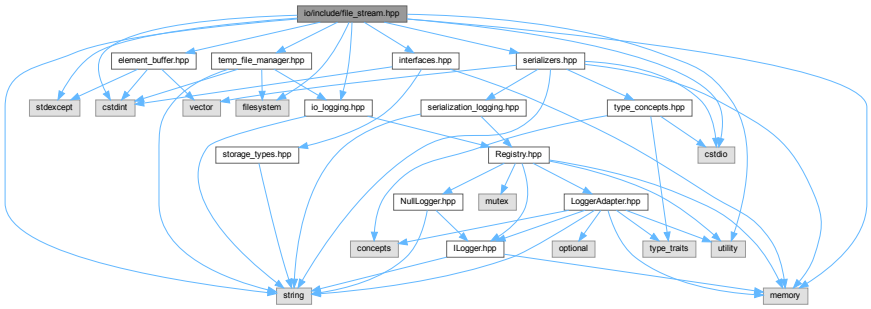
9.7 io/include/file_stream.hpp File Reference

File implementations of input/output streams.

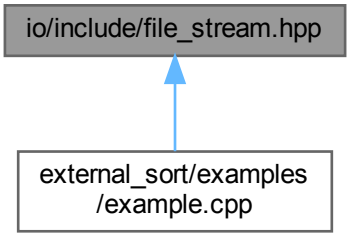
```
#include "element_buffer.hpp"
#include "interfaces.hpp"
```

```
#include "temp_file_manager.hpp"
#include "serializers.hpp"
#include "io_logging.hpp"
#include <cstdint>
#include <cstdio>
#include <filesystem>
#include <memory>
#include <stdexcept>
#include <string>
#include <utility>
```

Include dependency graph for file_stream.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class io::FileInputStream< T >

  File implementation of an input stream.
- class io::FileOutputStream< T >

  File implementation of an output stream.
- class io::FileStreamFactory< T >

  Factory for file streams.

Namespaces

- namespace io

9.7.1 Detailed Description

File implementations of input/output streams.

Definition in file file_stream.hpp.

9.8 file_stream.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file file_stream.hpp
00003  * @brief File implementations of input/output streams
00004  */
00005
00006 #pragma once
00007
00008 #include "element_buffer.hpp"
00009 #include "interfaces.hpp"
00010 #include "temp_file_manager.hpp"
00011 #include "serializers.hpp"
00012 #include "io_logging.hpp"
00013
00014 #include <cstdint>
00015 #include <cstdio>
00016 #include <filesystem>
00017 #include <memory>
00018 #include <stdexcept>
00019 #include <string>
00020 #include <cstdio>
00021 #include <filesystem>
00022 #include <stdexcept>
00023 #include <utility>
00024
00025 namespace io {
00026
00027 /**
00028  * @brief File implementation of an input stream
00029  *
00030  * Implements the IInputStream interface for reading data from a file.
00031  * Uses buffering for efficient reading.
00032  *
00033  * Note: This implementation provides efficient access for non-trivial types
00034  * by supporting TakeValue() (move out) and minimizing copies when possible.
00035  *
00036  * @tparam T The type of elements in the stream
00037  */
00038 template <typename T>
00039 class FileInputStream : public IInputStream<T> {
00040 private:
00041     StorageId id_;                          ///< Storage identifier (file name)
00042     FILE* file_ptr_ = nullptr;              ///< Pointer to the file
00043     ElementBuffer<T> buffer_;               ///< Buffer for reading
00044     uint64_t total_elements_in_file_ = 0;   ///< Total number of elements in the file
00045     uint64_t total_elements_read_ = 0;      ///< Number of elements read so far
00046     bool is_exhausted_ = false;             ///< Flag indicating if the stream is exhausted
00047     T current_value_{};                     ///< The current element
00048     bool has_valid_value_ = false;          ///< Flag indicating if the current element is valid
00049     std::unique_ptr<serialization::Serializer<T»
00050         serializer_member_{};  ///< Serializer for the elements
00051
00052     /**
00053      * @brief Fills the internal buffer with data from the file
00054      */
00055     void FillBuffer();
00056
00057 public:
00058     /**
00059      * @brief Constructor for the file input stream
00060      * @param filename The name of the file to read from
00061      * @param buffer_capacity_elements Buffer capacity in elements
00062      * @throws std::runtime_error if the file cannot be opened
00063      */
00064     FileInputStream(const StorageId& filename, uint64_t buffer_capacity_elements);
00065
00066     /**
00067      * @brief Destructor, closes the file
00068      */
00069     ~FileInputStream() override;
```

```
00070
00071     FileInputStream(const FileInputStream&) = delete;
00072     FileInputStream& operator=(const FileInputStream&) = delete;
00073
00074     FileInputStream(FileInputStream&& other) noexcept;
00075     FileInputStream& operator=(FileInputStream&& other) noexcept;
00076
00077     /**
00078      * @brief Advances the stream to the next element
00079      */
00080     void Advance() override;
00081
00082     /**
00083      * @brief Returns the current element of the stream
00084      * @return A constant reference to the current element
00085      * @throws std::logic_error if the stream is exhausted
00086      */
00087     const T& Value() const override;
00088
00089     /**
00090      * @brief Checks if the stream is exhausted
00091      * @return true if there are no more elements to read
00092      */
00093     bool IsExhausted() const override;
00094
00095     /**
00096      * @brief Checks if the original storage was empty
00097      * @return true if the file contained no elements
00098      */
00099     bool IsEmptyOriginalStorage() const override;
00100
00101     /**
00102      * @brief Transfers the ownership of the current value and resets the state
00103      * @return The current value
00104      * @throws std::logic_error if the stream is exhausted
00105      */
00106     T TakeValue();
00107 };
00108
00109 /**
00110  * @brief File implementation of an output stream
00111  *
00112  * Implements the IOutputStream interface for writing data to a file.
00113  * Uses buffering for efficient writing.
00114  *
00115  * @tparam T The type of elements in the stream
00116  */
00117 template <typename T>
00118 class FileOutputStream : public IOutputStream<T> {
00119 private:
00120     StorageId id_;                          ///< Storage identifier (file name)
00121     FILE* file_ptr_ = nullptr;              ///< Pointer to the file
00122     ElementBuffer<T> buffer_;               ///< Buffer for writing
00123     uint64_t total_elements_written_ = 0;   ///< Number of elements written
00124     uint64_t total_bytes_written_ = 0;      ///< Total bytes written (data + header)
00125     bool finalized_ = false;                ///< Flag indicating if the stream has been finalized
00126     std::unique_ptr<serialization::Serializer<T»
00127         serializer_member_{};  ///< Serializer for the elements
00128
00129     /**
00130      * @brief Flushes the buffer to the file
00131      */
00132     void FlushBufferInternal();
00133
00134 public:
00135     /**
00136      * @brief Constructor for the file output stream
00137      * @param filename The name of the file to write to
00138      * @param buffer_capacity_elements Buffer capacity in elements
00139      * @throws std::runtime_error if the file cannot be created
00140      */
00141     FileOutputStream(const StorageId& filename, uint64_t buffer_capacity_elements);
00142
00143     /**
00144      * @brief Destructor, finalizes the stream
00145      */
00146     ~FileOutputStream() override;
00147
00148     FileOutputStream(const FileOutputStream&) = delete;
00149     FileOutputStream& operator=(const FileOutputStream&) = delete;
00150
00151     FileOutputStream(FileOutputStream&& other) noexcept;
00152     FileOutputStream& operator=(FileOutputStream&& other) noexcept;
00153
00154     /**
00155      * @brief Writes an element to the stream
00156      * @param value The element to write
```

```
00157      * @throws std::logic_error if the stream has been finalized
00158      */
00159     void Write(const T& value) override;
00160
00161     /**
00162      * @brief Writes an rvalue element to the stream
00163      * @param value The rvalue element to write
00164      * @throws std::logic_error if the stream has been finalized
00165      */
00166     void Write(T&& value) override;
00167
00168     /**
00169      * @brief Finalizes the stream, writing all buffered data to the file
00170      */
00171     void Finalize() override;
00172
00173     /**
00174      * @brief Returns the total number of elements written
00175      * @return The number of elements written to the stream
00176      */
00177     uint64_t GetTotalElementsWritten() const override;
00178
00179     /**
00180      * @brief Returns the total number of bytes written to the file
00181      * @return The total number of bytes written (including header)
00182      */
00183     uint64_t GetTotalBytesWritten() const override;
00184
00185     /**
00186      * @brief Returns the file identifier
00187      * @return The StorageId of the file
00188      */
00189     StorageId GetId() const override;
00190 };
00191
00192 /**
00193  * @brief Factory for file streams
00194  *
00195  * Implements the IStreamFactory interface for creating file-based I/O streams
00196  * and managing temporary files.
00197  *
00198  * @tparam T The type of elements in the streams
00199  */
00200 template <typename T>
00201 class FileStreamFactory : public IStreamFactory<T> {
00202 private:
00203     TempFileManager temp_file_manager_;  ///< Temporary file manager
00204
00205 public:
00206     /**
00207      * @brief Constructor for the file stream factory
00208      * @param base_temp_dir_name The name of the base directory for temporary files
00209      */
00210     explicit FileStreamFactory(const std::string& base_temp_dir_name = "temp_files");
00211
00212     /**
00213      * @brief Creates an input stream for reading from a file
00214      * @param id The identifier of the file
00215      * @param buffer_capacity_elements The buffer capacity
00216      * @return A unique pointer to an IInputStream
00217      */
00218     std::unique_ptr<IInputStream<T> CreateInputStream(const StorageId& id,
00219                                                       uint64_t buffer_capacity_elements) override;
00220
00221     /**
00222      * @brief Creates an output stream for writing to a file
00223      * @param id The identifier of the file
00224      * @param buffer_capacity_elements The buffer capacity
00225      * @return A unique pointer to an IOutputStream
00226      */
00227     std::unique_ptr<IOutputStream<T> CreateOutputStream(
00228         const StorageId& id, uint64_t buffer_capacity_elements) override;
00229
00230     /**
00231      * @brief Creates an output stream for writing to a temporary file
00232      * @param out_temp_id Output parameter for the temporary file's ID
00233      * @param buffer_capacity_elements The buffer capacity
00234      * @return A unique pointer to an IOutputStream
00235      */
00236     std::unique_ptr<IOutputStream<T> CreateTempOutputStream(
00237         StorageId& out_temp_id, uint64_t buffer_capacity_elements) override;
00238
00239     /**
00240      * @brief Deletes a file
00241      * @param id The identifier of the file to delete
00242      */
00243     void DeleteStorage(const StorageId& id) override;
```

```
00244
00245     /**
00246      * @brief Makes a temporary file permanent
00247      * @param temp_id The identifier of the temporary file
00248      * @param final_id The identifier of the final file
00249      */
00250     void MakeStoragePermanent(const StorageId& temp_id, const StorageId& final_id) override;
00251
00252     /**
00253      * @brief Checks if a file exists
00254      * @param id The identifier of the file
00255      * @return true if the file exists
00256      */
00257     bool StorageExists(const StorageId& id) const override;
00258
00259     /**
00260      * @brief Returns the path to the temporary files directory
00261      * @return The path to the temporary files directory
00262      */
00263     StorageId GetTempStorageContextId() const override;
00264 };
00265
00266 template <typename T>
00267 void FileInputStream<T>::FillBuffer() {
00268     if (is_exhausted_ ||
00269         (total_elements_in_file_ > 0 && total_elements_read_ >= total_elements_in_file_)) {
00270         buffer_.SetValidElementsCount(0);
00271         is_exhausted_ = true;
00272         has_valid_value_ = false;
00273         return;
00274     }
00275     uint64_t elements_to_read_this_pass =
00276         std::min(buffer_.Capacity(), total_elements_in_file_ - total_elements_read_);
00277
00278     if (elements_to_read_this_pass == 0) {
00279         buffer_.SetValidElementsCount(0);
00280         is_exhausted_ = true;
00281         has_valid_value_ = false;
00282         return;
00283     }
00284
00285     buffer_.Clear();
00286
00287     uint64_t elements_actually_read = 0;
00288     if constexpr (serialization::PodSerializable<T>) {
00289         elements_actually_read =
00290             fread(buffer_.RawDataPtr(), sizeof(T), elements_to_read_this_pass, file_ptr_);
00291         buffer_.SetValidElementsCount(elements_actually_read);
00292     } else {
00293         for (uint64_t i = 0; i < elements_to_read_this_pass; ++i) {
00294             T element;
00295             if (!serializer_member_->Deserialize(element, file_ptr_)) {
00296                 break;
00297             }
00298             buffer_.PushBack(element);
00299             elements_actually_read++;
00300         }
00301     }
00302
00303     if (elements_actually_read < elements_to_read_this_pass && !feof(file_ptr_)) {
00304         if (ferror(file_ptr_)) {
00305             throw std::runtime_error("FileInputStream: Error reading from file: " + id_ +
00306                                      " (ferror set)");
00307         }
00308         throw std::runtime_error("FileInputStream: Failed to read expected elements from file: " +
00309                                  id_);
00310     }
00311     if (elements_actually_read == 0 && total_elements_read_ < total_elements_in_file_ &&
00312         feof(file_ptr_)) {
00313         is_exhausted_ = true;
00314         has_valid_value_ = false;
00315     }
00316 }
00317
00318 template <typename T>
00319 FileInputStream<T>::FileInputStream(const StorageId& filename, uint64_t buffer_capacity_elements)
00320     : id_(filename), buffer_(buffer_capacity_elements) {
00321     if constexpr (!serialization::PodSerializable<T>) {
00322         serializer_member_ = serialization::CreateSerializer<T>();
00323     }
00324     file_ptr_ = fopen(id_.c_str(), "rb");
00325     if (!file_ptr_) {
00326         throw std::runtime_error("FileInputStream: Cannot open input file: " + id_);
00327     }
00328     if (setvbuf(file_ptr_, nullptr, _IONBF, 0) != 0) {
00329         detail::LogWarning("FileInputStream could not disable stdio buffering for " + id_);
00330     }
```

```
00331
00332     if (fread(&total_elements_in_file_, sizeof(uint64_t), 1, file_ptr_) != 1) {
00333         if (feof(file_ptr_) && std::filesystem::file_size(id_) < sizeof(uint64_t)) {
00334             total_elements_in_file_ = 0;
00335         } else {
00336             fclose(file_ptr_);
00337             throw std::runtime_error("FileInputStream: Cannot read size header from file: " + id_);
00338         }
00339     }
00340     detail::LogInfo("FileInputStream: " + id_ +
00341                     " opened. Header elements: " + std::to_string(total_elements_in_file_));
00342
00343     if (total_elements_in_file_ == 0) {
00344         is_exhausted_ = true;
00345         has_valid_value_ = false;
00346     } else {
00347         Advance();
00348     }
00349 }
00350
00351 template <typename T>
00352 FileInputStream<T>::~FileInputStream() {
00353     if (file_ptr_) {
00354         fclose(file_ptr_);
00355         detail::LogInfo("FileInputStream: Closed " + id_);
00356     }
00357 }
00358
00359 template <typename T>
00360 FileInputStream<T>::FileInputStream(FileInputStream&& other) noexcept
00361     : id_(std::move(other.id_)),
00362       file_ptr_(other.file_ptr_),
00363       buffer_(std::move(other.buffer_)),
00364       total_elements_in_file_(other.total_elements_in_file_),
00365       total_elements_read_(other.total_elements_read_),
00366       is_exhausted_(other.is_exhausted_),
00367       current_value_(std::move(other.current_value_)),
00368       has_valid_value_(other.has_valid_value_),
00369       serializer_member_(std::move(other.serializer_member_)) {
00370     other.file_ptr_ = nullptr;
00371 }
00372
00373 template <typename T>
00374 FileInputStream<T>& FileInputStream<T>::operator=(FileInputStream&& other) noexcept {
00375     if (this != &other) {
00376         if (file_ptr_) {
00377             fclose(file_ptr_);
00378         }
00379         id_ = std::move(other.id_);
00380         file_ptr_ = other.file_ptr_;
00381         buffer_ = std::move(other.buffer_);
00382         total_elements_in_file_ = other.total_elements_in_file_;
00383         total_elements_read_ = other.total_elements_read_;
00384         is_exhausted_ = other.is_exhausted_;
00385         current_value_ = std::move(other.current_value_);
00386         has_valid_value_ = other.has_valid_value_;
00387         serializer_member_ = std::move(other.serializer_member_);
00388         other.file_ptr_ = nullptr;
00389     }
00390     return *this;
00391 }
00392
00393 template <typename T>
00394 void FileInputStream<T>::Advance() {
00395     if (is_exhausted_ ||
00396         (total_elements_in_file_ > 0 && total_elements_read_ >= total_elements_in_file_)) {
00397         has_valid_value_ = false;
00398         is_exhausted_ = true;
00399         return;
00400     }
00401     if (!buffer_.HasMoreToRead()) {
00402         FillBuffer();
00403         if (!buffer_.HasMoreToRead()) {
00404             has_valid_value_ = false;
00405             is_exhausted_ = true;
00406             return;
00407         }
00408     }
00409     current_value_ = buffer_.ReadNext();
00410     total_elements_read_++;
00411     has_valid_value_ = true;
00412     if (total_elements_read_ >= total_elements_in_file_) {
00413         is_exhausted_ = true;
00414     }
00415 }
00416
00417 template <typename T>
```

```
00418 const T& FileInputStream<T>::Value() const {
00419     if (!has_valid_value_) {
00420         throw std::logic_error("Value from exhausted FileInputStream: " + id_);
00421     }
00422     return current_value_;
00423 }
00424
00425 template <typename T>
00426 T FileInputStream<T>::TakeValue() {
00427     if (!has_valid_value_) {
00428         throw std::logic_error("TakeValue from exhausted FileInputStream: " + id_);
00429     }
00430     T tmp = std::move(current_value_);
00431     has_valid_value_ = false;
00432     return tmp;
00433 }
00434
00435 template <typename T>
00436 bool FileInputStream<T>::IsExhausted() const {
00437     return is_exhausted_ && !has_valid_value_;
00438 }
00439
00440 template <typename T>
00441 bool FileInputStream<T>::IsEmptyOriginalStorage() const {
00442     return total_elements_in_file_ == 0;
00443 }
00444
00445 template <typename T>
00446 void FileOutputStream<T>::FlushBufferInternal() {
00447     if (buffer_.IsEmpty() || !file_ptr_ || finalized_) {
00448         return;
00449     }
00450
00451     uint64_t successful_writes = 0;
00452     if constexpr (serialization::PodSerializable<T>) {
00453         successful_writes = fwrite(buffer_.Data(), sizeof(T), buffer_.Size(), file_ptr_);
00454         total_bytes_written_ += successful_writes * sizeof(T);
00455     } else {
00456         const T* data = buffer_.Data();
00457         for (uint64_t i = 0; i < buffer_.Size(); ++i) {
00458             uint64_t element_size = serializer_member_->GetSerializedSize(data[i]);
00459             if (!serializer_member_->Serialize(data[i], file_ptr_)) {
00460                 throw std::runtime_error("FileOutputStream: Failed to Serialize element to file: " +
00461                                          id_);
00462             }
00463             total_bytes_written_ += element_size;
00464             successful_writes++;
00465         }
00466     }
00467
00468     if (successful_writes != buffer_.Size()) {
00469         throw std::runtime_error("FileOutputStream: Failed to write full buffer to file: " + id_);
00470     }
00471
00472     detail::LogInfo("FileOutputStream: Flushed " + std::to_string(buffer_.Size()) +
00473                     " elements to " + id_);
00474     buffer_.Clear();
00475 }
00476
00477 template <typename T>
00478 FileOutputStream<T>::FileOutputStream(const StorageId& filename, uint64_t buffer_capacity_elements)
00479     : id_(filename), buffer_(buffer_capacity_elements) {
00480     if constexpr (!serialization::PodSerializable<T>) {
00481         serializer_member_ = serialization::CreateSerializer<T>();
00482     }
00483     file_ptr_ = fopen(id_.c_str(), "wb");
00484     if (!file_ptr_) {
00485         throw std::runtime_error("FileOutputStream: Cannot open output file: " + id_);
00486     }
00487     if (setvbuf(file_ptr_, nullptr, _IONBF, 0) != 0) {
00488         detail::LogWarning("FileOutputStream could not disable stdio buffering for " + id_);
00489     }
00490     uint64_t placeholder_size = 0;
00491     if (fwrite(&placeholder_size, sizeof(uint64_t), 1, file_ptr_) != 1) {
00492         fclose(file_ptr_);
00493         throw std::runtime_error("FileOutputStream: Failed to write placeholder size to " + id_);
00494     }
00495     // Account for the header size in total bytes written
00496     total_bytes_written_ = sizeof(uint64_t);
00497     detail::LogInfo("FileOutputStream: " + id_ + " opened for writing.");
00498 }
00499
00500 template <typename T>
00501 FileOutputStream<T>::~FileOutputStream() {
00502     Finalize();
00503 }
00504
```

```
00505 template <typename T>
00506 FileOutputStream<T>::FileOutputStream(FileOutputStream&& other) noexcept
00507     : id_(std::move(other.id_)),
00508       file_ptr_(other.file_ptr_),
00509       buffer_(std::move(other.buffer_)),
00510       total_elements_written_(other.total_elements_written_),
00511       total_bytes_written_(other.total_bytes_written_),
00512       finalized_(other.finalized_),
00513       serializer_member_(std::move(other.serializer_member_)) {
00514     other.file_ptr_ = nullptr;
00515     other.finalized_ = true;
00516 }
00517
00518 template <typename T>
00519 FileOutputStream<T>& FileOutputStream<T>::operator=(FileOutputStream&& other) noexcept {
00520     if (this != &other) {
00521         Finalize();
00522         id_ = std::move(other.id_);
00523         file_ptr_ = other.file_ptr_;
00524         buffer_ = std::move(other.buffer_);
00525         total_elements_written_ = other.total_elements_written_;
00526         total_bytes_written_ = other.total_bytes_written_;
00527         finalized_ = other.finalized_;
00528         serializer_member_ = std::move(other.serializer_member_);
00529         other.file_ptr_ = nullptr;
00530         other.finalized_ = true;
00531     }
00532     return *this;
00533 }
00534
00535 template <typename T>
00536 void FileOutputStream<T>::Write(const T& value) {
00537     if (finalized_) {
00538         throw std::logic_error("Write to finalized FileOutputStream: " + id_);
00539     }
00540     if (buffer_.PushBack(value)) {
00541         FlushBufferInternal();
00542     }
00543     total_elements_written_++;
00544 }
00545
00546 template <typename T>
00547 void FileOutputStream<T>::Write(T&& value) {
00548     if (finalized_) {
00549         throw std::logic_error("Write to finalized FileOutputStream: " + id_);
00550     }
00551     if (buffer_.PushBack(std::move(value))) {
00552         FlushBufferInternal();
00553     }
00554     total_elements_written_++;
00555 }
00556
00557 template <typename T>
00558 void FileOutputStream<T>::Finalize() {
00559     if (finalized_ || !file_ptr_) {
00560         return;
00561     }
00562     FlushBufferInternal();
00563     if (fseek(file_ptr_, 0, SEEK_SET) != 0) {
00564         detail::LogError("Error: FileOutputStream fseek failed for " + id_);
00565     } else {
00566         if (fwrite(&total_elements_written_, sizeof(uint64_t), 1, file_ptr_) != 1) {
00567             detail::LogError("Error: FileOutputStream fwrite header failed for " + id_);
00568         } else {
00569             detail::LogInfo("FileOutputStream: Finalized " + id_ +
00570                             ". Header elements: " + std::to_string(total_elements_written_));
00571         }
00572     }
00573     fflush(file_ptr_);
00574     fclose(file_ptr_);
00575     file_ptr_ = nullptr;
00576     finalized_ = true;
00577 }
00578
00579 template <typename T>
00580 uint64_t FileOutputStream<T>::GetTotalElementsWritten() const {
00581     return total_elements_written_;
00582 }
00583
00584 template <typename T>
00585 uint64_t FileOutputStream<T>::GetTotalBytesWritten() const {
00586     return total_bytes_written_;
00587 }
00588
00589 template <typename T>
00590 StorageId FileOutputStream<T>::GetId() const {
00591     return id_;
```

```
00592 }
00593
00594 template <typename T>
00595 FileStreamFactory<T>::FileStreamFactory(const std::string& base_temp_dir_name)
00596     : temp_file_manager_(base_temp_dir_name) {
00597 }
00598
00599 template <typename T>
00600 std::unique_ptr<IInputStream<T» FileStreamFactory<T>::CreateInputStream(
00601     const StorageId& id, uint64_t buffer_capacity_elements) {
00602     return std::make_unique<FileInputStream<T»(id, buffer_capacity_elements);
00603 }
00604
00605 template <typename T>
00606
00607 std::unique_ptr<IOutputStream<T» FileStreamFactory<T>::CreateOutputStream(
00608     const StorageId& id, uint64_t buffer_capacity_elements) {
00609     return std::make_unique<FileOutputStream<T»(id, buffer_capacity_elements);
00610 }
00611
00612 template <typename T>
00613
00614 std::unique_ptr<IOutputStream<T» FileStreamFactory<T>::CreateTempOutputStream(
00615     StorageId& out_temp_id, uint64_t buffer_capacity_elements) {
00616     out_temp_id = temp_file_manager_.GenerateTempFilename("r", ".b");
00617     return std::make_unique<FileOutputStream<T»(out_temp_id, buffer_capacity_elements);
00618 }
00619
00620 template <typename T>
00621
00622 void FileStreamFactory<T>::DeleteStorage(const StorageId& id) {
00623     temp_file_manager_.CleanupFile(id);
00624 }
00625
00626 template <typename T>
00627
00628 void FileStreamFactory<T>::MakeStoragePermanent(const StorageId& temp_id,
00629                                                 const StorageId& final_id) {
00630     if (temp_id == final_id) {
00631         return;
00632     }
00633     try {
00634         if (std::filesystem::exists(final_id)) {
00635             std::filesystem::remove(final_id);
00636         }
00637         std::filesystem::rename(temp_id, final_id);
00638     } catch (const std::filesystem::filesystem_error& e) {
00639         detail::LogWarning("MakeStoragePermanent: Rename failed (" + std::string(e.what()) +
00640                            "), attempting copy for " + temp_id + " to " + final_id);
00641         {
00642             FileInputStream<T> src(temp_id, 1024);
00643             FileOutputStream<T> dst(final_id, 1024);
00644             while (!src.IsExhausted()) {
00645                 dst.Write(src.TakeValue());
00646                 src.Advance();
00647             }
00648         }
00649         temp_file_manager_.CleanupFile(temp_id);
00650     }
00651 }
00652
00653 template <typename T>
00654
00655 bool FileStreamFactory<T>::StorageExists(const StorageId& id) const {
00656     return std::filesystem::exists(id);
00657 }
00658
00659 template <typename T>
00660
00661 StorageId FileStreamFactory<T>::GetTempStorageContextId() const {
00662     return temp_file_manager_.GetBaseDirPath().string();
00663 }
00664
00665 }  // namespace io
```

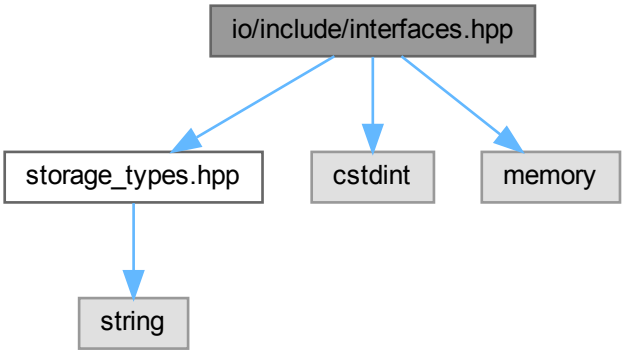
9.9 io/include/interfaces.hpp File Reference

Base interfaces for input and output strems and for stream factories.

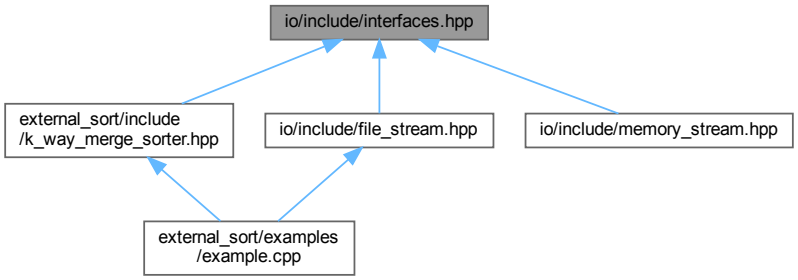
```
#include "storage_types.hpp"
#include <cstdint>
```

```
#include <memory>
```

Include dependency graph for interfaces.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class io::IInputStream< T >

  Interface for input stream of elements of type T.
- class io::IOutputStream< T >

  Interface for output stream of elements of type T.
- class io::IStreamFactory< T >

  Interface for a stream factory.

Namespaces

- namespace io

9.9.1 Detailed Description

Base interfaces for input and output strems and for stream factories.

Definition in file interfaces.hpp.

9.10 interfaces.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file interfaces.hpp
00003  * @brief Base interfaces for input and output strems and for stream factories
00004  */
00005
00006 #pragma once
00007
00008 #include "storage_types.hpp"
00009
00010 #include <cstdint>
00011 #include <memory>
00012
00013 namespace io {
00014
00015 /**
00016  * @brief Interface for input stream of elements of type T
00017  * @tparam T type of elements in stream
00018  */
00019 template <typename T>
00020 class IInputStream {
00021    public:
00022     virtual ~IInputStream() = default;
00023
00024     /**
00025      * @brief Advances stream to the next element
00026      */
00027     virtual void Advance() = 0;
00028
00029     /**
00030      * @brief Returns current element of the stream
00031      * @return Constant reference to the current element of the stream
00032      * @throws std::logic_error if the stream is exhausted
00033      */
00034     virtual const T& Value() const = 0;
00035
00036     /**
00037      * @brief Move out the current element of the stream
00038      *
00039      * Returns the current element by value using move semantics where possible.
00040      * Does not advance the stream; call Advance() to move to the next element.
00041      * @return Current element by value
00042      * @throws std::logic_error if the stream is exhausted
00043      */
00044     virtual T TakeValue() = 0;
00045
00046     /**
00047      * @brief Checks if the stream is exhausted
00048      * @return true, if there are no more elements to read
00049      */
00050     virtual bool IsExhausted() const = 0;
00051
00052     /**
00053      * @brief Checks if original storage is empty
00054      * @return true, if original storage doesn't contains eny elements
00055      */
00056     virtual bool IsEmptyOriginalStorage() const = 0;
00057 };
00058
00059 /**
00060  * @brief Interface for output stream of elements of type T
00061  * @tparam T type of elements in stream
00062  */
00063 template <typename T>
00064 class IOutputStream {
00065    public:
00066     virtual ~IOutputStream() = default;
00067
00068     /**
00069      * @brief Writes an element to the stream
```

```
00070      * @param value The element to write
00071      * @throws std::logic_error if the stream has been finalized
00072      */
00073     virtual void Write(const T& value) = 0;
00074
00075     /**
00076      * @brief Writes an element to the stream using move semantics
00077      * @param value The element to write (moved)
00078      * @throws std::logic_error if the stream has been finalized
00079      */
00080     virtual void Write(T&& value) = 0;
00081
00082     /**
00083      * @brief Finalizes the stream, writing all buffered data
00084      */
00085     virtual void Finalize() = 0;
00086
00087     /**
00088      * @brief Returns the total number of elements written
00089      * @return The number of elements written to the stream
00090      */
00091     virtual uint64_t GetTotalElementsWritten() const = 0;
00092
00093     /**
00094      * @brief Returns the total number of bytes written to the storage
00095      *
00096      * This method returns the actual size in bytes of the serialized data written
00097      * to the storage. For complex types (e.g., types with dynamic memory like
00098      * std::string), this accounts for the actual serialized size, not just
00099      * sizeof(T) * element_count.
00100      *
00101      * This is particularly useful for:
00102      * - Tracking actual disk usage for file-based streams
00103      * - Generating data files of specific target sizes in benchmarks
00104      * - Monitoring memory usage for in-memory streams
00105      * - Estimating storage requirements for complex data types
00106      *
00107      * @return The total number of bytes written (including any headers/metadata)
00108      *
00109      * @note For POD types, this typically equals: sizeof(header) + sizeof(T) *
     GetTotalElementsWritten()
00110      * @note For complex types (std::string, custom types), this reflects the actual serialized size
00111      * @note This value includes file headers (e.g., element count header in FileOutputStream)
00112      *
00113      * @par Example:
00114      * @code
00115      * auto output_stream = factory->CreateOutputStream("data.bin", 1024);
00116      * for (const auto& person : people) {
00117      *     output_stream->Write(person);
00118      * }
00119      * uint64_t total_bytes = output_stream->GetTotalBytesWritten();
00120      * // total_bytes will be the actual file size including all string data
00121      * @endcode
00122      */
00123     virtual uint64_t GetTotalBytesWritten() const = 0;
00124
00125     /**
00126      * @brief Returns the identifier of the storage
00127      * @return The StorageId of the storage associated with the stream
00128      */
00129     virtual StorageId GetId() const = 0;
00130 };
00131
00132 /**
00133  * @brief Interface for a stream factory
00134  *
00135  * Manages creation, deletion, and finalization of data streams,
00136  * which can be either file-based or in-memory.
00137  *
00138  * @tparam T The type of elements in the streams
00139  */
00140 template <typename T>
00141 class IStreamFactory {
00142    public:
00143     virtual ~IStreamFactory() = default;
00144
00145     /**
00146      * @brief Creates an input stream to read from an existing storage
00147      * @param id The identifier of the storage to read from
00148      * @param buffer_capacity_elements The capacity of the stream's internal buffer
00149      * @return A unique pointer to an IInputStream
00150      * @throws std::runtime_error if the storage cannot be opened or does not exist
00151      */
00152     virtual std::unique_ptr<IInputStream<T>> CreateInputStream(
00153         const StorageId& id, uint64_t buffer_capacity_elements) = 0;
00154
00155     /**
```

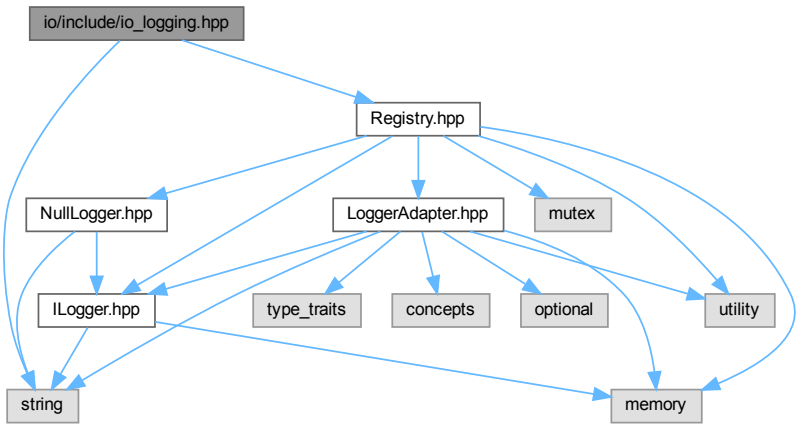
```
00156      * @brief Creates an output stream to write to a new or existing storage
00157      *
00158      * If the storage already exists, its content will be overwritten.
00159      *
00160      * @param id The identifier of the storage to write to
00161      * @param buffer_capacity_elements The capacity of the stream's internal buffer
00162      * @return A unique pointer to an IOutputStream
00163      * @throws std::runtime_error if the storage cannot be created or opened for writing
00164      */
00165     virtual std::unique_ptr<IOutputStream<T> CreateOutputStream(
00166         const StorageId& id, uint64_t buffer_capacity_elements) = 0;
00167
00168     /**
00169      * @brief Creates an output stream to write to a new temporary storage
00170      *
00171      * The factory assigns a unique ID to this temporary storage.
00172      *
00173      * @param out_temp_id Output parameter that will receive the ID of the created temporary storage
00174      * @param buffer_capacity_elements The capacity of the stream's internal buffer
00175      * @return A unique pointer to an IOutputStream for the temporary storage
00176      */
00177     virtual std::unique_ptr<IOutputStream<T> CreateTempOutputStream(
00178         StorageId& out_temp_id, uint64_t buffer_capacity_elements) = 0;
00179
00180     /**
00181      * @brief Deletes a storage
00182      * @param id The identifier of the storage to delete
00183      */
00184     virtual void DeleteStorage(const StorageId& id) = 0;
00185
00186     /**
00187      * @brief Makes a temporary storage permanent under a new (or the same) ID
00188      *
00189      * This typically involves renaming or copying. The original temporary storage (temp_id)
00190      * is usually deleted or becomes inaccessible after this operation.
00191      *
00192      * @param temp_id The identifier of the source temporary storage
00193      * @param final_id The identifier for the destination permanent storage
00194      * @throws std::runtime_error if the operation fails
00195      */
00196     virtual void MakeStoragePermanent(const StorageId& temp_id, const StorageId& final_id) = 0;
00197
00198     /**
00199      * @brief Checks if a storage with the given ID exists
00200      * @param id The identifier of the storage to check
00201      * @return true if the storage exists, false otherwise
00202      */
00203     virtual bool StorageExists(const StorageId& id) const = 0;
00204
00205     /**
00206      * @brief Gets an identifier representing the base path or context for temporary storages
00207      * This is used to prevent conflicts (e.g., writing output to the temporary directory itself).
00208      * @return A StorageId representing the context for temporary storages
00209      */
00210     virtual StorageId GetTempStorageContextId() const = 0;
00211 };
00212
00213 /**
00214  * @brief Move-friendly input/output stream operations
00215  *
00216  * Notes for users:
00217  * - Use TakeValue() when you want to obtain the current element by value
00218  *   and you do not need the original anymore. This allows implementations
00219  *   to move the element out efficiently and is recommended for large or
00220  *   move-only types.
00221  * - Use the rvalue overload Write(T&&) to write elements with move
00222  *   semantics where possible. This avoids copies when the caller has an
00223  *   rvalue or uses std::move().
00224  */
00225
00226 }  // namespace io
```

9.11 io/include/io_logging.hpp File Reference

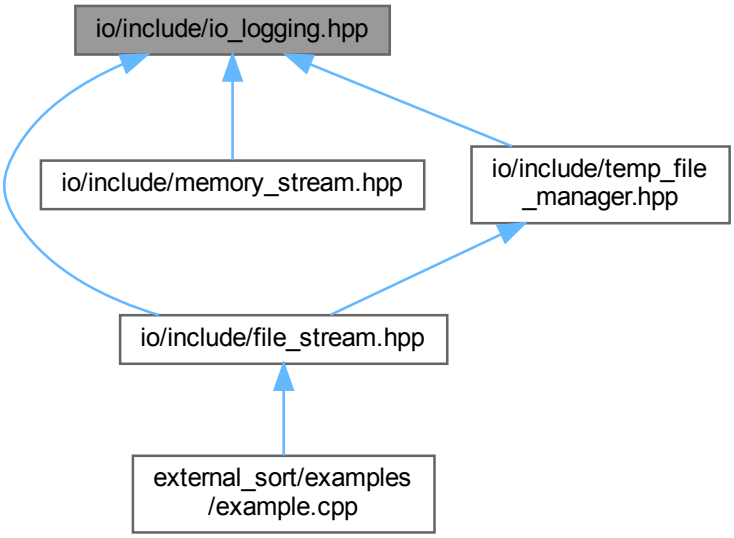
Logging integration for io library.

```
#include "Registry.hpp"
#include <string>
```

Include dependency graph for io_logging.hpp:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace io
- namespace io::detail

Functions

- void io::detail::LogError (const std::string &message)

  Log an error message from io operations.
- void io::detail::LogWarning (const std::string &message)

  Log a warning message from io operations.
- void io::detail::LogInfo (const std::string &message)

  Log an informational message from io operations.

9.11.1 Detailed Description

Logging integration for io library.

Logging integration for External Sort library.

Definition in file io_logging.hpp.

9.12 io_logging.hpp

Go to the documentation of this file.

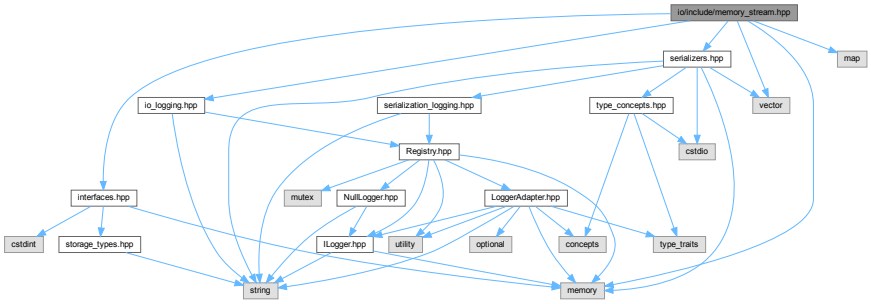
```
00001 /**
00002  * @file io_logging.hpp
00003  * @brief Logging integration for io library
00004  */
00005
00006 #pragma once
00007
00008 #include "Registry.hpp"
00009 #include <string>
00010
00011 namespace io {
00012 namespace detail {
00013
00014 /**
00015  * @brief Log an error message from io operations
00016  * @param message The error message to log
00017  */
00018 inline void LogError(const std::string& message) {
00019     auto& logger = logging::detail::GetLoggerInstance();
00020     if (logger) {
00021         logger->LogError("[IO] " + message);
00022     }
00023 }
00024
00025 /**
00026  * @brief Log a warning message from io operations
00027  * @param message The warning message to log
00028  */
00029 inline void LogWarning(const std::string& message) {
00030     auto& logger = logging::detail::GetLoggerInstance();
00031     if (logger) {
00032         logger->LogWarning("[IO] " + message);
00033     }
00034 }
00035
00036 /**
00037  * @brief Log an informational message from io operations
00038  * @param message The info message to log
00039  */
00040 inline void LogInfo(const std::string& message) {
00041     auto& logger = logging::detail::GetLoggerInstance();
00042     if (logger) {
00043         logger->LogInfo("[IO] " + message);
00044     }
00045 }
00046
00047 }  // namespace detail
00048 }  // namespace io
```

9.13 io/include/memory_stream.hpp File Reference

In-memory implementations of input/output streams.

```
#include "io_logging.hpp"
#include "interfaces.hpp"
#include "serializers.hpp"
#include <map>
#include <memory>
#include <vector>
```

Include dependency graph for memory_stream.hpp:



Classes

- class io::InMemoryOutputStream< T >

  In-memory implementation of output stream.
- class io::InMemoryInputStream< T >

  In-memory implementation of input stream.
- class io::InMemoryStreamFactory< T >

  Factory for in-memory streams.

Namespaces

- namespace io

9.13.1 Detailed Description

In-memory implementations of input/output streams.

Definition in file memory_stream.hpp.

9.14 memory_stream.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file memory_stream.hpp
00003  * @brief In-memory implementations of input/output streams
00004  */
00005
00006 #pragma once
00007
00008 #include "io_logging.hpp"
00009 #include "interfaces.hpp"
00010 #include "serializers.hpp"
00011
00012 #include <map>
00013 #include <memory>
00014 #include <vector>
00015
00016 namespace io {
00017
00018 /**
00019  * @brief In-memory implementation of output stream
00020  *
00021  * Implements the IOutputStream interface for writing data to memory.
00022  * All data is stored in std::vector.
00023  *
00024  * Note: Supports move-friendly Write(T&&) so callers can avoid copies when
00025  * they have rvalue elements or use std::move().
00026  *
00027  * @tparam T Type of elements in the stream
00028  */
00029 template <typename T>
00030 class InMemoryOutputStream : public IOutputStream<T> {
00031    private:
00032     StorageId id_;                                ///< Stream identifier
00033     std::shared_ptr<std::vector<T> data_ptr_;    ///< Pointer to data
00034     std::shared_ptr<uint64_t> actual_size_ptr_;  ///< Pointer to size ("header")
00035     uint64_t elements_written_ = 0;              ///< Number of elements written
00036     bool finalized_ = false;                     ///< Finalization flag
00037
00038    public:
00039     /**
00040      * @brief Constructor for in-memory output stream
00041      * @param id Stream identifier
00042      * @param data_vec_ptr Pointer to data vector
00043      * @param size_ptr Pointer to data size
00044      * @param buffer_capacity Buffer capacity (not used in in-memory)
00045      */
00046     InMemoryOutputStream(
00047         StorageId id, std::shared_ptr<std::vector<T> data_vec_ptr,
00048         std::shared_ptr<uint64_t> size_ptr, [[maybe_unused]] uint64_t buffer_capacity);
00049
00050     /**
00051      * @brief Destructor that finalizes the stream
00052      */
00053     ~InMemoryOutputStream() override;
00054
00055     /**
00056      * @brief Writes an element to the stream
00057      * @param value Element to write
00058      * @throws std::logic_error if the stream is finalized
00059      */
00060     void Write(const T& value) override;
00061
00062     /**
00063      * @brief Writes an rvalue element to the stream
00064      * @param value Rvalue element to write
00065      * @throws std::logic_error if the stream is finalized
00066      */
00067     void Write(T&& value) override;
00068
00069     /**
00070      * @brief Finalizes the stream
00071      */
00072     void Finalize() override;
00073
00074     /**
00075      * @brief Returns the total number of elements written
00076      * @return Number of elements written to the stream
00077      */
00078     uint64_t GetTotalElementsWritten() const override;
00079
00080     /**
00081      * @brief Returns the total number of bytes that would be written if serialized
00082      * @return Estimated total bytes for in-memory data
```

```
00083      */
00084     uint64_t GetTotalBytesWritten() const override;
00085
00086     /**
00087      * @brief Returns the stream identifier
00088      * @return Stream StorageId
00089      */
00090     StorageId GetId() const override;
00091 };
00092
00093 /**
00094  * @brief In-memory implementation of input stream
00095  *
00096  * Implements the IInputStream interface for reading data from memory.
00097  * Reads data from std::vector.
00098  *
00099  * @tparam T Type of elements in the stream
00100  */
00101 template <typename T>
00102 class InMemoryInputStream : public IInputStream<T> {
00103    private:
00104     StorageId id_;                                      ///< Stream identifier
00105     std::shared_ptr<std::vector<T» data_ptr_;  ///< Pointer to data (modifiable to allow move-out of
    elements)
00106     uint64_t total_elements_in_storage_;               ///< Total number of elements in storage
00107     uint64_t read_cursor_ = 0;                         ///< Read cursor
00108     T current_value_{};                                ///< Current element
00109     bool has_valid_value_ = false;                     ///< Current element validity flag
00110     bool is_exhausted_ = false;                        ///< Stream exhaustion flag
00111
00112    public:
00113     /**
00114      * @brief Constructor for in-memory input stream
00115      * @param id Stream identifier
00116      * @param data_vec_ptr Pointer to data vector
00117      * @param actual_storage_size Actual data size
00118      * @param buffer_capacity Buffer capacity (not used in in-memory)
00119      */
00120     InMemoryInputStream(
00121         StorageId id, std::shared_ptr<std::vector<T» data_vec_ptr,
00122         uint64_t actual_storage_size, [[maybe_unused]] uint64_t buffer_capacity);
00123
00124     /**
00125      * @brief Destructor
00126      */
00127     ~InMemoryInputStream() override = default;
00128
00129     /**
00130      * @brief Advances the stream to the next element
00131      */
00132     void Advance() override;
00133
00134     /**
00135      * @brief Returns the current element of the stream
00136      * @return Constant reference to the current element
00137      * @throws std::logic_error if the stream is exhausted
00138      */
00139     const T& Value() const override;
00140
00141     /**
00142      * @brief Takes the current element of the stream (removing it from the stream)
00143      * @return The current element
00144      * @throws std::logic_error if the stream is exhausted
00145      */
00146     T TakeValue() override;
00147
00148     /**
00149      * @brief Checks if the stream is exhausted
00150      * @return true if there are no more elements to read
00151      */
00152     bool IsExhausted() const override;
00153
00154     /**
00155      * @brief Checks if the original storage was empty
00156      * @return true if the storage contained no elements
00157      */
00158     bool IsEmptyOriginalStorage() const override;
00159 };
00160
00161 /**
00162  * @brief Factory for in-memory streams
00163  *
00164  * Implements the IStreamFactory interface for creating streams
00165  * that work with data in memory.
00166  *
00167  * @tparam T Type of elements in the streams
00168  */
```

```
00169 template <typename T>
00170 class InMemoryStreamFactory : public IStreamFactory<T> {
00171    private:
00172     std::map<StorageId, std::shared_ptr<std::vector<T>> storages_;  ///< Data storages
00173     std::map<StorageId, std::shared_ptr<uint64_t>
00174         storage_declared_sizes_;                              ///< Declared storage sizes
00175     uint64_t temp_id_counter_ = 0;                            ///< Temporary ID counter
00176     const std::string temp_prefix_ = "in_memory_temp_run_";  ///< Temporary ID prefix
00177
00178    public:
00179     /**
00180      * @brief Constructor for in-memory stream factory
00181      */
00182     InMemoryStreamFactory() = default;
00183
00184     /**
00185      * @brief Creates an input stream for reading from memory
00186      * @param id Storage identifier
00187      * @param buffer_capacity_elements Buffer capacity (not used)
00188      * @return Unique pointer to InMemoryInputStream
00189      * @throws std::runtime_error if storage is not found
00190      */
00191     std::unique_ptr<IInputStream<T» CreateInputStream(
00192         const StorageId& id, uint64_t buffer_capacity_elements) override;
00193
00194     /**
00195      * @brief Creates an output stream for writing to memory
00196      * @param id Storage identifier
00197      * @param buffer_capacity_elements Buffer capacity (not used)
00198      * @return Unique pointer to InMemoryOutputStream
00199      */
00200     std::unique_ptr<IOutputStream<T» CreateOutputStream(
00201         const StorageId& id, uint64_t buffer_capacity_elements) override;
00202
00203     /**
00204      * @brief Creates an output stream for writing to temporary storage
00205      * @param out_temp_id Output parameter for temporary storage ID
00206      * @param buffer_capacity_elements Buffer capacity (not used)
00207      * @return Unique pointer to InMemoryOutputStream
00208      */
00209     std::unique_ptr<IOutputStream<T» CreateTempOutputStream(
00210         StorageId& out_temp_id, uint64_t buffer_capacity_elements) override;
00211
00212     /**
00213      * @brief Deletes a storage from memory
00214      * @param id Storage identifier to delete
00215      */
00216     void DeleteStorage(const StorageId& id) override;
00217
00218     /**
00219      * @brief Makes a temporary storage permanent
00220      * @param temp_id Temporary storage identifier
00221      * @param final_id Final storage identifier
00222      * @throws std::runtime_error if temporary storage is not found
00223      */
00224     void MakeStoragePermanent(const StorageId& temp_id, const StorageId& final_id) override;
00225
00226     /**
00227      * @brief Checks if a storage exists
00228      * @param id Storage identifier
00229      * @return true if the storage exists
00230      */
00231     bool StorageExists(const StorageId& id) const override;
00232
00233     /**
00234      * @brief Returns the temporary storage context
00235      * @return Temporary ID prefix
00236      */
00237     StorageId GetTempStorageContextId() const override;
00238
00239     /**
00240      * @brief Returns storage data (for tests)
00241      * @param id Storage identifier
00242      * @return Pointer to data or nullptr if not found
00243      */
00244     std::shared_ptr<const std::vector<T> GetStorageData(const StorageId& id) const;
00245
00246     /**
00247      * @brief Returns declared storage size (for tests)
00248      * @param id Storage identifier
00249      * @return Declared size or 0 if not found
00250      */
00251     uint64_t GetStorageDeclaredSize(const StorageId& id) const;
00252 };
00253
00254 template <typename T>
00255 InMemoryOutputStream<T>::InMemoryOutputStream(
```

```
00256     StorageId id, std::shared_ptr<std::vector<T>> data_vec_ptr, std::shared_ptr<uint64_t> size_ptr,
00257     [[maybe_unused]] uint64_t buffer_capacity)
00258     : id_(std::move(id))
00259     , data_ptr_(std::move(data_vec_ptr))
00260     , actual_size_ptr_(std::move(size_ptr)) {
00261     data_ptr_->clear();
00262     *actual_size_ptr_ = 0;
00263 }
00264
00265 template <typename T>
00266 InMemoryOutputStream<T>::~InMemoryOutputStream() {
00267     Finalize();
00268 }
00269
00270 template <typename T>
00271 void InMemoryOutputStream<T>::Write(const T& value) {
00272     if (finalized_) {
00273         throw std::logic_error("Write to finalized InMemoryOutputStream: " + id_);
00274     }
00275     data_ptr_->push_back(value);
00276     elements_written_++;
00277 }
00278
00279 template <typename T>
00280 void InMemoryOutputStream<T>::Write(T&& value) {
00281     if (finalized_) {
00282         throw std::logic_error("Write to finalized InMemoryOutputStream: " + id_);
00283     }
00284     data_ptr_->push_back(std::move(value));
00285     elements_written_++;
00286 }
00287
00288 template <typename T>
00289 void InMemoryOutputStream<T>::Finalize() {
00290     if (finalized_) {
00291         return;
00292     }
00293     *actual_size_ptr_ = elements_written_;
00294     finalized_ = true;
00295     detail::LogInfo("InMemoryOutputStream: Finalized " + id_ +
00296                     ". Elements: " + std::to_string(*actual_size_ptr_));
00297 }
00298
00299 template <typename T>
00300 uint64_t InMemoryOutputStream<T>::GetTotalElementsWritten() const {
00301     return elements_written_;
00302 }
00303
00304 template <typename T>
00305 uint64_t InMemoryOutputStream<T>::GetTotalBytesWritten() const {
00306     // For in-memory streams, calculate the serialized size of all elements
00307     // This includes the "header" (element count) that would be written to a file
00308     uint64_t total_bytes = sizeof(uint64_t);  // Header with element count
00309
00310     auto serializer = serialization::CreateSerializer<T>();
00311     for (size_t i = 0; i < elements_written_; ++i) {
00312         total_bytes += serializer->GetSerializedSize((*data_ptr_)[i]);
00313     }
00314
00315     return total_bytes;
00316 }
00317
00318 template <typename T>
00319 StorageId InMemoryOutputStream<T>::GetId() const {
00320     return id_;
00321 }
00322
00323 template <typename T>
00324 InMemoryInputStream<T>::InMemoryInputStream(
00325     StorageId id, std::shared_ptr<std::vector<T>> data_vec_ptr, uint64_t actual_storage_size,
00326     [[maybe_unused]] uint64_t buffer_capacity)
00327     : id_(std::move(id))
00328     , data_ptr_(std::move(data_vec_ptr))
00329     , total_elements_in_storage_(actual_storage_size) {
00330     if (total_elements_in_storage_ > data_ptr_->size()) {
00331         detail::LogWarning("Warning: InMemoryInputStream " + id_ +
00332                            " declared size (" + std::to_string(total_elements_in_storage_) +
00333                            ") > actual vector size (" + std::to_string(data_ptr_->size()) +
00334                            "). Clamping to actual size.");
00335         total_elements_in_storage_ = data_ptr_->size();
00336     }
00337
00338     if (total_elements_in_storage_ == 0) {
00339         is_exhausted_ = true;
00340         has_valid_value_ = false;
00341     } else {
00342         Advance();
```

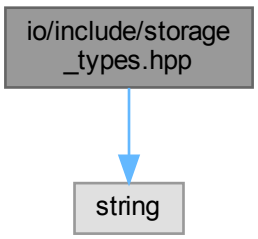
```
00343     }
00344 }
00345
00346 template <typename T>
00347 void InMemoryInputStream<T>::Advance() {
00348     if (is_exhausted_ || read_cursor_ >= total_elements_in_storage_) {
00349         has_valid_value_ = false;
00350         is_exhausted_ = true;
00351         return;
00352     }
00353     // Move the element out of the underlying storage to avoid copies.
00354     current_value_ = std::move((*data_ptr_)[read_cursor_]);
00355     read_cursor_++;
00356     has_valid_value_ = true;
00357     if (read_cursor_ >= total_elements_in_storage_) {
00358         is_exhausted_ = true;
00359     }
00360 }
00361
00362 template <typename T>
00363 const T& InMemoryInputStream<T>::Value() const {
00364     if (!has_valid_value_) {
00365         throw std::logic_error("Value from exhausted InMemoryInputStream: " + id_);
00366     }
00367     return current_value_;
00368 }
00369
00370 template <typename T>
00371 T InMemoryInputStream<T>::TakeValue() {
00372     if (!has_valid_value_) {
00373         throw std::logic_error("TakeValue from exhausted InMemoryInputStream: " + id_);
00374     }
00375     T tmp = std::move(current_value_);
00376     return tmp;
00377 }
00378
00379 template <typename T>
00380 bool InMemoryInputStream<T>::IsExhausted() const {
00381     return is_exhausted_ && !has_valid_value_;
00382 }
00383
00384 template <typename T>
00385 bool InMemoryInputStream<T>::IsEmptyOriginalStorage() const {
00386     return total_elements_in_storage_ == 0;
00387 }
00388
00389 template <typename T>
00390 std::unique_ptr<IInputStream<T» InMemoryStreamFactory<T>::CreateInputStream(
00391     const StorageId& id, uint64_t buffer_capacity_elements) {
00392     auto it_data = storages_.find(id);
00393     auto it_size = storage_declared_sizes_.find(id);
00394     if (it_data == storages_.end() || it_size == storage_declared_sizes_.end()) {
00395         throw std::runtime_error("InMemoryStreamFactory: Storage ID not found for input: " + id);
00396     }
00397     return std::make_unique<InMemoryInputStream<T»(
00398         id, it_data->second, *(it_size->second), buffer_capacity_elements);
00399 }
00400
00401 template <typename T>
00402 std::unique_ptr<IOutputStream<T» InMemoryStreamFactory<T>::CreateOutputStream(
00403     const StorageId& id, uint64_t buffer_capacity_elements) {
00404     auto data_ptr = std::make_shared<std::vector<T»();
00405     auto size_ptr = std::make_shared<uint64_t>(0);
00406     storages_[id] = data_ptr;
00407     storage_declared_sizes_[id] = size_ptr;
00408     return std::make_unique<InMemoryOutputStream<T»(
00409         id, data_ptr, size_ptr, buffer_capacity_elements);
00410 }
00411
00412 template <typename T>
00413 std::unique_ptr<IOutputStream<T» InMemoryStreamFactory<T>::CreateTempOutputStream(
00414     StorageId& out_temp_id, uint64_t buffer_capacity_elements) {
00415     out_temp_id = temp_prefix_ + std::to_string(temp_id_counter_++);
00416     auto data_ptr = std::make_shared<std::vector<T»();
00417     auto size_ptr = std::make_shared<uint64_t>(0);
00418     storages_[out_temp_id] = data_ptr;
00419     storage_declared_sizes_[out_temp_id] = size_ptr;
00420     return std::make_unique<InMemoryOutputStream<T»(
00421         out_temp_id, data_ptr, size_ptr, buffer_capacity_elements);
00422 }
00423
00424 template <typename T>
00425 void InMemoryStreamFactory<T>::DeleteStorage(const StorageId& id) {
00426     storages_.erase(id);
00427     storage_declared_sizes_.erase(id);
00428     detail::LogInfo("InMemoryStreamFactory: Deleted storage " + id);
00429 }
```

```
00430
00431 template <typename T>
00432 void InMemoryStreamFactory<T>::MakeStoragePermanent(
00433     const StorageId& temp_id, const StorageId& final_id) {
00434     if (temp_id == final_id) {
00435         return;
00436     }
00437
00438     auto it_temp_data = storages_.find(temp_id);
00439     auto it_temp_size = storage_declared_sizes_.find(temp_id);
00440     if (it_temp_data == storages_.end() || it_temp_size == storage_declared_sizes_.end()) {
00441         throw std::runtime_error("InMemoryStreamFactory: Temp ID not found: " + temp_id);
00442     }
00443     storages_[final_id] = it_temp_data->second;
00444     storage_declared_sizes_[final_id] = it_temp_size->second;
00445
00446     storages_.erase(it_temp_data);
00447     storage_declared_sizes_.erase(it_temp_size);
00448     detail::LogInfo("InMemoryStreamFactory: Made " + temp_id + " permanent as " + final_id);
00449 }
00450
00451 template <typename T>
00452 bool InMemoryStreamFactory<T>::StorageExists(const StorageId& id) const {
00453     return storages_.count(id);
00454 }
00455
00456 template <typename T>
00457 StorageId InMemoryStreamFactory<T>::GetTempStorageContextId() const {
00458     return temp_prefix_;
00459 }
00460
00461 template <typename T>
00462 std::shared_ptr<const std::vector<T» InMemoryStreamFactory<T>::GetStorageData(
00463     const StorageId& id) const {
00464     auto it = storages_.find(id);
00465     return (it != storages_.end()) ? it->second : nullptr;
00466 }
00467
00468 template <typename T>
00469 uint64_t InMemoryStreamFactory<T>::GetStorageDeclaredSize(const StorageId& id) const {
00470     auto it = storage_declared_sizes_.find(id);
00471     return (it != storage_declared_sizes_.end()) ? *(it->second) : 0;
00472 }
00473
00474
00475 }  // namespace io
```

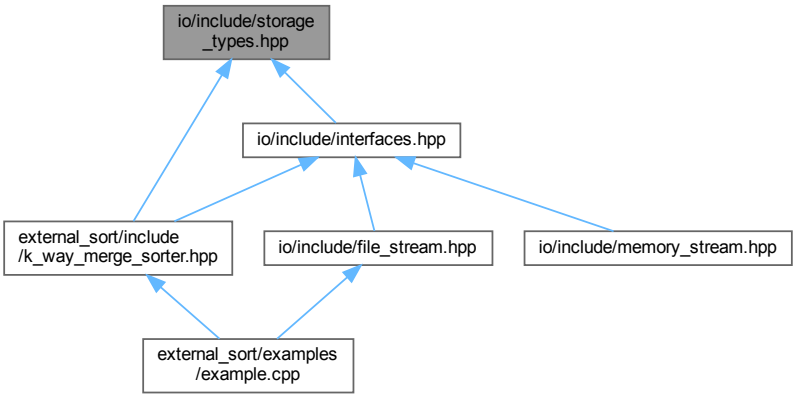
9.15 io/include/storage_types.hpp File Reference

Definitions of different identificators for files.

`#include <string>`
Include dependency graph for storage_types.hpp:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace io

Typedefs

- using io::StorageId = std::string

  Alias for std::string to identify files by their name.

9.15.1 Detailed Description

Definitions of different identificators for files.

Definition in file storage_types.hpp.

9.16 storage_types.hpp

Go to the documentation of this file.

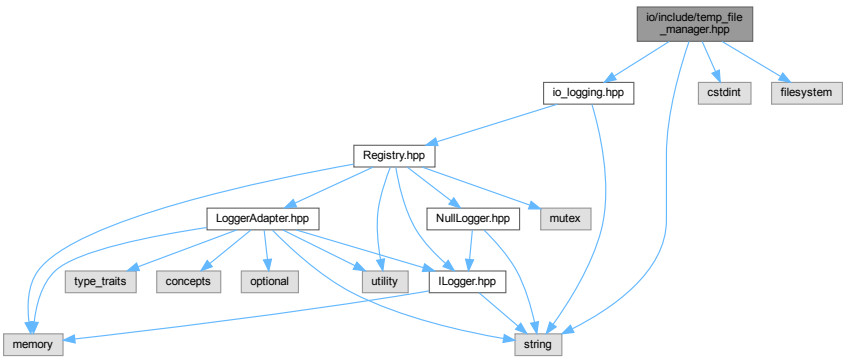
```
00001 /**
00002  * @file storage_types.hpp
00003  * @brief Definitions of different identificators for files
00004  */
00005
00006 #pragma once
00007
00008 #include <string>
00009
00010 namespace io {
00011
00012 /**
00013  * @brief Alias for std::string to identify files by their name
00014  */
00015 using StorageId = std::string;
00016
00017 }  // namespace external_sort
```

9.17 io/include/temp_file_manager.hpp File Reference

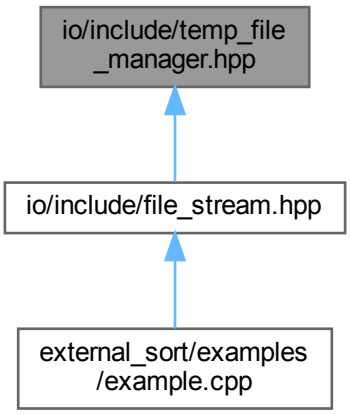
Temporary file manager.

```
#include "io_logging.hpp"
#include <cstdint>
#include <filesystem>
#include <string>
```

Include dependency graph for temp_file_manager.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class io::TempFileManager

  A manager for temporary files.

Namespaces

- namespace io

9.17.1 Detailed Description

Temporary file manager.

Definition in file temp_file_manager.hpp.

9.18 temp_file_manager.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file temp_file_manager.hpp
00003  * @brief Temporary file manager
00004  */
00005
00006 #pragma once
00007
00008 #include "io_logging.hpp"
00009
00010 #include <cstdint>
00011 #include <filesystem>
00012 #include <string>
00013
00014 namespace io {
00015
00016 /**
00017  * @brief A manager for temporary files
00018  *
00019  * The class is responsible for creating, tracking, and deleting temporary files
00020  * used during the external sorting process.
00021  */
00022 class TempFileManager {
00023 private:
00024     std::filesystem::path base_temp_dir_path_;  ///< Base directory for temporary files
00025     uint64_t temp_file_counter_ = 0;            ///< Counter for generating unique names
00026     bool owns_directory_{};  ///< Flag to indicate if this instance owns the directory
00027
00028 public:
00029     /**
00030      * @brief Constructs a new TempFileManager object
00031      * @param base_dir_name The name of the base directory for temporary files
00032      */
00033     explicit TempFileManager(const std::string& base_dir_name = "ts");
00034
00035     /**
00036      * @brief Destructor that cleans up all temporary files
00037      */
00038     ~TempFileManager();
00039
00040     TempFileManager(const TempFileManager&) = delete;
00041     TempFileManager& operator=(const TempFileManager&) = delete;
00042
00043     TempFileManager(TempFileManager&&) = default;
00044     TempFileManager& operator=(TempFileManager&&) = default;
00045
00046     /**
00047      * @brief Generates a unique temporary filename
00048      * @param prefix The prefix for the filename
00049      * @param extension The file extension
00050      * @return The full path to the temporary file
00051      */
00052     std::string GenerateTempFilename(const std::string& prefix = "tmp",
00053                                      const std::string& extension = ".b");
00054
00055     /**
00056      * @brief Deletes the specified file
00057      * @param filename_str Path to the file to be deleted
00058      */
00059     void CleanupFile(const std::string& filename_str);
00060
00061     /**
```

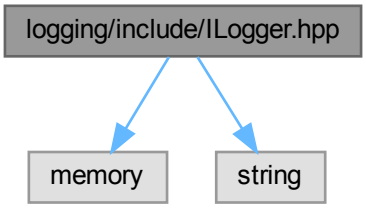
```
00062      * @brief Returns the path to the base directory
00063      * @return A const reference to the base directory path
00064      */
00065     const std::filesystem::path& GetBaseDirPath() const;
00066 };
00067
00068 TempFileManager::TempFileManager(const std::string& base_dir_name) {
00069     base_temp_dir_path_ = std::filesystem::current_path() / base_dir_name;
00070     if (!std::filesystem::exists(base_temp_dir_path_)) {
00071         std::error_code ec;
00072         std::filesystem::create_directories(base_temp_dir_path_, ec);
00073         if (ec) {
00074             throw std::runtime_error("TempFileManager failed to create temp directory: " +
00075                                      base_temp_dir_path_.string() + " Error: " + ec.message());
00076         }
00077         owns_directory_ = true;
00078         detail::LogInfo("TempFileManager created temporary directory: " +
      base_temp_dir_path_.string());
00079     } else {
00080         owns_directory_ = false;
00081         detail::LogInfo("TempFileManager using existing temporary directory: "
00082                         + base_temp_dir_path_.string());
00083     }
00084 }
00085
00086 TempFileManager::~TempFileManager() {
00087     detail::LogInfo("TempFileManager destructor: Attempting to clean up temporary directory...");
00088     if (owns_directory_ && std::filesystem::exists(base_temp_dir_path_)) {
00089         std::error_code ec;
00090         std::filesystem::remove_all(base_temp_dir_path_, ec);
00091         if (ec) {
00092             detail::LogWarning("Warning: TempFileManager failed to remove_all temp directory "
00093                                + base_temp_dir_path_.string() + ": " + ec.message());
00094         } else {
00095             detail::LogInfo("TempFileManager removed temp directory and all its contents: "
00096                             + base_temp_dir_path_.string());
00097         }
00098     } else if (std::filesystem::exists(base_temp_dir_path_)) {
00099         detail::LogInfo("TempFileManager: Temporary directory "
00100                         + base_temp_dir_path_.string()
00101                         + " exists but was not created by this instance, not removing.");
00102     }
00103 }
00104
00105 std::string TempFileManager::GenerateTempFilename(const std::string& prefix,
00106                                                   const std::string& extension) {
00107     std::filesystem::path file_path =
00108         base_temp_dir_path_ / (prefix + std::to_string(temp_file_counter_++) + extension);
00109     detail::LogInfo("TempFileManager generated temp filename: " + file_path.string());
00110     return file_path.string();
00111 }
00112
00113 void TempFileManager::CleanupFile(const std::string& filename_str) {
00114     std::filesystem::path file_path(filename_str);
00115     if (std::filesystem::exists(file_path)) {
00116         std::error_code ec;
00117         std::filesystem::remove(file_path, ec);
00118         if (ec) {
00119             detail::LogWarning("CleanupFile failed to remove " + filename_str + ": "
00120                                + ec.message());
00121         }
00122     }
00123 }
00124
00125 const std::filesystem::path& TempFileManager::GetBaseDirPath() const {
00126     return base_temp_dir_path_;
00127 }
00128
00129 }  // namespace io
```

9.19 logging/include/ILogger.hpp File Reference

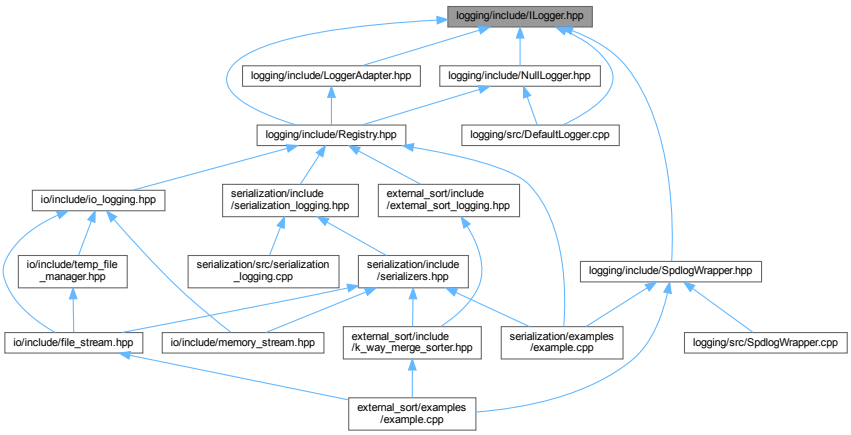
Interface definition for all logger implementations.

```
#include <memory>
#include <string>
```

Include dependency graph for ILogger.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class logging::ILogger

  Abstract base class defining the logger interface.

Namespaces

- namespace logging

Functions

- std::shared_ptr< ILogger > logging::CreateDefaultLogger ()

  Factory function to create a default logger instance.

9.19.1 Detailed Description

Interface definition for all logger implementations.

Definition in file ILogger.hpp.

9.20 ILogger.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file ILogger.hpp
00003  * @brief Interface definition for all logger implementations
00004  */
00005
00006 #pragma once
00007
00008 #include <memory>
00009 #include <string>
00010
00011 namespace logging {
00012
00013 /**
00014  * @brief Abstract base class defining the logger interface
00015  *
00016  * All logger implementations must inherit from this interface and implement
00017  * the three logging methods for different severity levels: info, warning, and error.
00018  * This allows the library to work with any logging backend through polymorphism.
00019  */
00020 class ILogger {
00021 public:
00022     virtual ~ILogger() = default;
00023
00024     /**
00025      * @brief Log an informational message
00026      * @param message The message to log
00027      */
00028     virtual void LogInfo(const std::string& message) = 0;
00029
00030     /**
00031      * @brief Log a warning message
00032      * @param message The message to log
00033      */
00034     virtual void LogWarning(const std::string& message) = 0;
00035
00036     /**
00037      * @brief Log an error message
00038      * @param message The message to log
00039      */
00040     virtual void LogError(const std::string& message) = 0;
00041 };
00042
00043 /**
00044  * @brief Factory function to create a default logger instance
00045  * @return Shared pointer to a NullLogger (default no-op logger)
00046  */
00047 std::shared_ptr<ILogger> CreateDefaultLogger();
00048
00049 }  // namespace logging
```

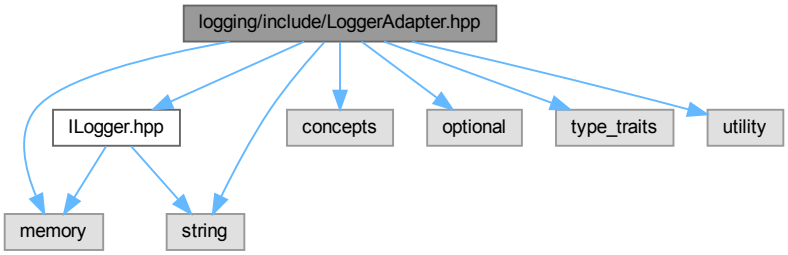
9.21 logging/include/LoggerAdapter.hpp File Reference

Wrapper for any user-defined logger to be used with the logging library.

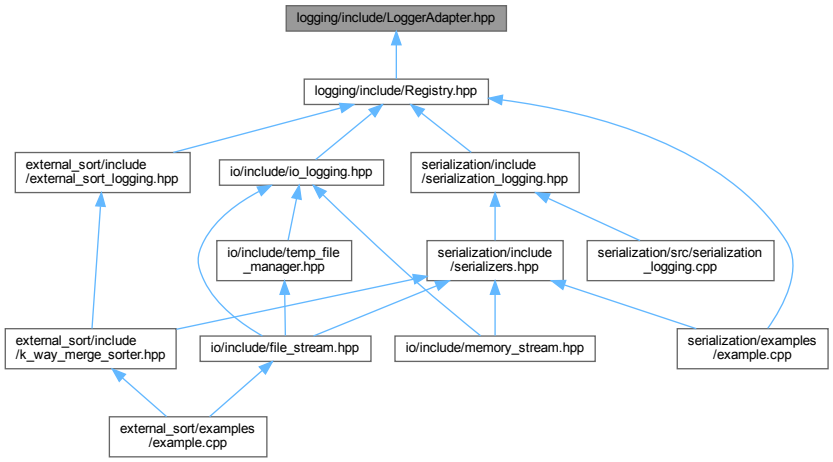
```
#include "ILogger.hpp"
#include <concepts>
#include <memory>
#include <optional>
#include <string>
#include <type_traits>
```

```
#include <utility>
```

Include dependency graph for LoggerAdapter.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class logging::LoggerAdapter< UserLogger >

  Adapter class that wraps a user-defined logger to conform to the ILogger interface.

Namespaces

- namespace logging

9.21.1 Detailed Description

Wrapper for any user-defined logger to be used with the logging library.

Definition in file LoggerAdapter.hpp.

9.22 LoggerAdapter.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file LoggerAdapter.hpp
00003  * @brief Wrapper for any user-defined logger to be used with the logging library
00004  */
00005
00006 #pragma once
00007
00008 #include "ILogger.hpp"
00009 #include <concepts>
00010 #include <memory>
00011 #include <optional>
00012 #include <string>
00013 #include <type_traits>
00014 #include <utility>
00015
00016 namespace logging {
00017
00018 /**
00019  * @brief Adapter class that wraps a user-defined logger to conform to the ILogger interface.
00020  *
00021  * This template class allows any custom logger implementation to be used with current library by
00022  * adapting its interface to match the ILogger contract. The user logger must provide info(),
00023  * warn(), and error() methods that accept string messages.
00024  *
00025  * The adapter automatically handles both copyable and non-copyable loggers:
00026  * - Copyable loggers are stored by value
00027  * - Non-copyable loggers are stored via shared_ptr
00028  *
00029  * @tparam UserLogger The type of the user-defined logger to be adapted
00030  *
00031  * Example usage with copyable logger:
00032  * @code
00033  * struct MyLogger {
00034  *     void info(const std::string& msg) { std::cout « "[INFO] " « msg « '\n'; }
00035  *     void warn(const std::string& msg) { std::cout « "[WARN] " « msg « '\n'; }
00036  *     void error(const std::string& msg) { std::cerr « "[ERROR] " « msg « '\n'; }
00037  * };
00038  *
00039  * auto adapter = std::make_shared<LoggerAdapter<MyLogger»(MyLogger{});
00040  * @endcode
00041  *
00042  * Example usage with non-copyable logger stored via shared_ptr:
00043  * @code
00044  * struct HeavyLogger {
00045  *     HeavyLogger() = default;
00046  *     HeavyLogger(const HeavyLogger&) = delete;
00047  *     void info(const std::string& msg) { }
00048  *     void warn(const std::string& msg) { }
00049  *     void error(const std::string& msg) { }
00050  * };
00051  *
00052  * auto adapter = std::make_shared<LoggerAdapter<HeavyLogger»(
00053  *     std::make_shared<HeavyLogger>());
00054  * @endcode
00055  */
00056 template <typename UserLogger>
00057 class LoggerAdapter : public ILogger {
00058 public:
00059     /**
00060      * @brief Construct adapter for copyable logger by forwarding the user logger
00061      * @tparam T Deduced type for perfect forwarding
00062      * @param logger The user-defined logger instance to wrap (by value)
00063      */
00064     template <typename T>
00065         requires std::copy_constructible<std::decay_t<T» &&
00066                  (!std::same_as<std::decay_t<T>, std::shared_ptr<UserLogger»)
00067     explicit LoggerAdapter(T&& logger) : user_logger_(std::forward<T>(logger)) {
00068     }
00069
00070     /**
00071      * @brief Construct adapter for non-copyable logger via shared_ptr
00072      * @param logger Shared pointer to the user-defined logger instance
00073      */
00074     explicit LoggerAdapter(std::shared_ptr<UserLogger> logger)
00075         : user_logger_ptr_(std::move(logger)) {
00076     }
00077
00078     /**
00079      * @brief Log an informational message using the wrapped logger
00080      * @param message The message to log
00081      */
00082     void LogInfo(const std::string& message) override {
```

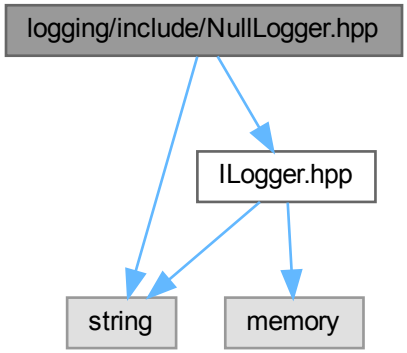
```
00083         if (user_logger_ptr_) {
00084             user_logger_ptr_->info(message);
00085         } else {
00086             user_logger_->info(message);
00087         }
00088     }
00089
00090     /**
00091      * @brief Log a warning message using the wrapped logger
00092      * @param message The message to log
00093      */
00094     void LogWarning(const std::string& message) override {
00095         if (user_logger_ptr_) {
00096             user_logger_ptr_->warn(message);
00097         } else {
00098             user_logger_->warn(message);
00099         }
00100     }
00101
00102     /**
00103      * @brief Log an error message using the wrapped logger
00104      * @param message The message to log
00105      */
00106     void LogError(const std::string& message) override {
00107         if (user_logger_ptr_) {
00108             user_logger_ptr_->error(message);
00109         } else {
00110             user_logger_->error(message);
00111         }
00112     }
00113
00114 private:
00115     std::optional<UserLogger> user_logger_;        ///< Logger stored by value (for copyable types)
00116     std::shared_ptr<UserLogger> user_logger_ptr_;  ///< Logger stored by pointer (for non-copyable)
00117 };
00118
00119 }  // namespace logging
```

9.23 logging/include/NullLogger.hpp File Reference

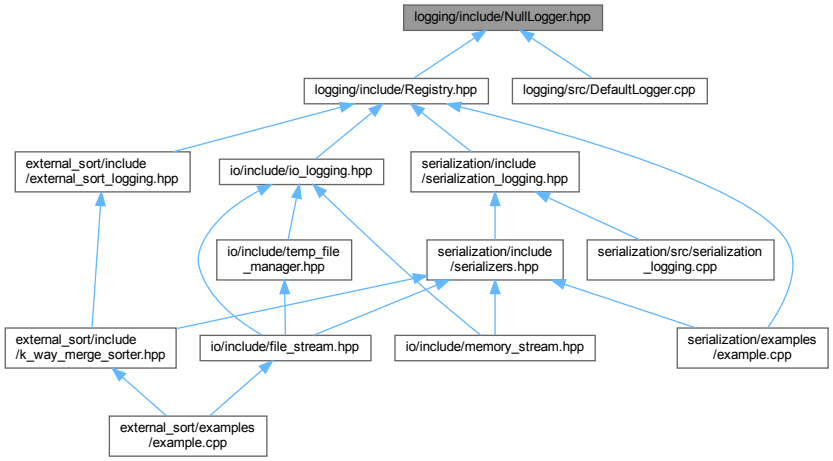
Empty no-op logger implementation for when logging is disabled.

```
#include <string>
#include "ILogger.hpp"
```

Include dependency graph for NullLogger.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class logging::NullLogger

  A logger implementation that discards all log messages.

Namespaces

- namespace logging

9.23.1 Detailed Description

Empty no-op logger implementation for when logging is disabled.

Definition in file NullLogger.hpp.

9.24 NullLogger.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file NullLogger.hpp
00003  * @brief Empty no-op logger implementation for when logging is disabled
00004  */
00005
00006 #pragma once
00007
00008 #include <string>
00009 #include "ILogger.hpp"
00010
00011 namespace logging {
00012
00013 /**
00014  * @brief A logger implementation that discards all log messages
00015  *
00016  * This is the default logger used when no custom logger is set.
00017  * All logging methods are no-ops, making it suitable for production
00018  * code where logging overhead should be minimal or when logging
```

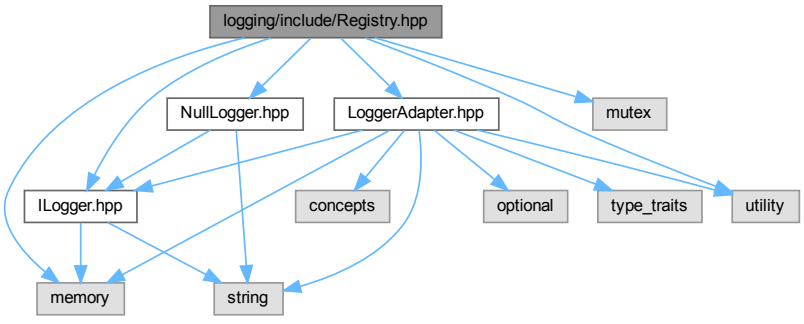
```
00019  * is not desired.
00020  */
00021 class NullLogger : public ILogger {
00022 public:
00023     /**
00024      * @brief Log an informational message (no-op)
00025      * @param message The message to discard (unused)
00026      */
00027     void LogInfo(const std::string& message) override {
00028         (void)message;  // Suppress unused parameter warning
00029     }
00030
00031     /**
00032      * @brief Log a warning message (no-op)
00033      * @param message The message to discard (unused)
00034      */
00035     void LogWarning(const std::string& message) override {
00036         (void)message;  // Suppress unused parameter warning
00037     }
00038
00039     /**
00040      * @brief Log an error message (no-op)
00041      * @param message The message to discard (unused)
00042      */
00043     void LogError(const std::string& message) override {
00044         (void)message;  // Suppress unused parameter warning
00045     }
00046 };
00047
00048 }  // namespace logging
```

9.25 logging/include/Registry.hpp File Reference

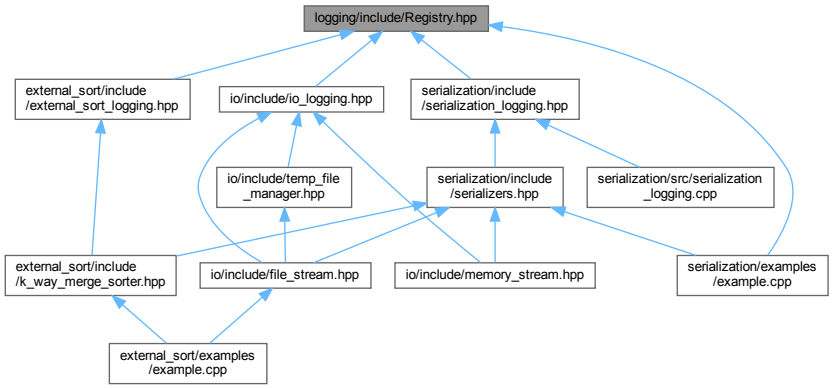
Global logger registry for setting and managing the active logger instance.

```
#include "ILogger.hpp"
#include "NullLogger.hpp"
#include "LoggerAdapter.hpp"
#include <memory>
#include <mutex>
#include <utility>
```

Include dependency graph for Registry.hpp:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace logging
- namespace logging::detail

Functions

- std::shared_ptr< ILogger > & logging::detail::GetLoggerInstance ()

  Get the global logger instance.
- std::mutex & logging::detail::GetLoggerMutex ()

  Get the mutex protecting the logger instance.
- template<typename UserLogger >
  void logging::SetLogger (UserLogger &&user_logger)

  Set a custom user-defined logger as the global logger.
- template<typename UserLogger >
  requires (!std::derived_from<UserLogger, ILogger>)
  void logging::SetLogger (std::shared_ptr< UserLogger > user_logger)

  Set a custom user-defined logger passed in std::shared_ptr as the global logger.
- template<typename DerivedLogger >
  requires std::derived_from<DerivedLogger, ILogger>
  void logging::SetLogger (std::shared_ptr< DerivedLogger > logger)

  Set a logger instance derived from ILogger as the global logger.
- void logging::SetLogger (std::shared_ptr< ILogger > logger)

  Set a logger instance that already conforms to the ILogger interface.
- void logging::SetDefaultLogger ()

  Reset the logger to the default NullLogger.

9.25.1 Detailed Description

Global logger registry for setting and managing the active logger instance.

Definition in file Registry.hpp.

9.26 Registry.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file Registry.hpp
00003  * @brief Global logger registry for setting and managing the active logger instance
00004  */
00005
00006 #pragma once
00007
00008 #include "ILogger.hpp"
00009 #include "NullLogger.hpp"
00010 #include "LoggerAdapter.hpp"
00011 #include <memory>
00012 #include <mutex>
00013 #include <utility>
00014
00015 namespace logging {
00016
00017 namespace detail {
00018 /**
00019  * @brief Get the global logger instance
00020  * @return Reference to the shared pointer holding the current logger
00021  */
00022 inline std::shared_ptr<ILogger>& GetLoggerInstance() {
00023     static std::shared_ptr<ILogger> logger = std::make_shared<NullLogger>();
00024     return logger;
00025 }
00026
00027 /**
00028  * @brief Get the mutex protecting the logger instance
00029  * @return Reference to the global logger mutex
00030  */
00031 inline std::mutex& GetLoggerMutex() {
00032     static std::mutex m;
00033     return m;
00034 }
00035 }  // namespace detail
00036
00037 /**
00038  * @brief Set a custom user-defined logger as the global logger
00039  *
00040  * Wraps the user logger in a LoggerAdapter and sets it as the active logger.
00041  * The user logger must provide info(), warn(), and error() methods.
00042  * Thread-safe: uses a mutex to protect the global logger instance.
00043  *
00044  * @tparam UserLogger The type of the user-defined logger
00045  * @param user_logger The logger instance to set (will be moved or copied)
00046  *
00047  * Example:
00048  * @code
00049  * struct MyLogger {
00050  *     void info(const std::string& msg) { }
00051  *     void warn(const std::string& msg) { }
00052  *     void error(const std::string& msg) { }
00053  * };
00054  * SetLogger(MyLogger{});
00055  * @endcode
00056  */
00057 template <typename UserLogger>
00058 void SetLogger(UserLogger&& user_logger) {
00059     auto adapter = std::make_shared<LoggerAdapter<std::decay_t<UserLogger>>>(
00060         std::forward<UserLogger>(user_logger));
00061     std::lock_guard<std::mutex> lock(detail::GetLoggerMutex());
00062     detail::GetLoggerInstance() = std::move(adapter);
00063 }
00064
00065 /**
00066  * @brief Set a custom user-defined logger passed in std::shared_ptr as the global logger
00067  *
00068  * Wraps the user logger (held by shared_ptr) in a LoggerAdapter and sets it as the active logger.
00069  * This overload can be used for non-copyable user loggers.
00070  * Thread-safe: uses a mutex to protect the global logger instance.
00071  *
00072  * @tparam UserLogger The type of the user-defined logger (not derived from ILogger)
00073  * @param user_logger Shared pointer to the user logger instance
00074  *
00075  * Example:
00076  * @code
00077  * struct HeavyLogger {
00078  *     HeavyLogger() = default;
00079  *     HeavyLogger(const HeavyLogger&) = delete;
00080  *     void info(const std::string& msg) { }
00081  *     void warn(const std::string& msg) { }
00082  *     void error(const std::string& msg) { }
```

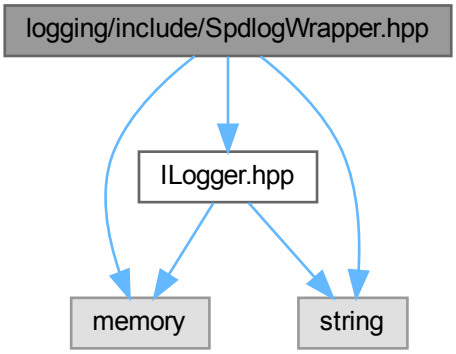
```
00083  * };
00084  * auto logger = std::make_shared<HeavyLogger>();
00085  * SetLogger(logger);
00086  * @endcode
00087  */
00088 template <typename UserLogger>
00089     requires (!std::derived_from<UserLogger, ILogger>)
00090 void SetLogger(std::shared_ptr<UserLogger> user_logger) {
00091     auto adapter = std::make_shared<LoggerAdapter<UserLogger»(std::move(user_logger));
00092     std::lock_guard<std::mutex> lock(detail::GetLoggerMutex());
00093     detail::GetLoggerInstance() = std::move(adapter);
00094 }
00095
00096 /**
00097  * @brief Set a logger instance derived from ILogger as the global logger
00098  *
00099  * Thread-safe: uses a mutex to protect the global logger instance.
00100  * This overload is used when passing a shared pointer to a type that derives from ILogger.
00101  *
00102  * @tparam DerivedLogger A type that derives from ILogger
00103  * @param logger Shared pointer to the derived logger instance to set
00104  */
00105 template <typename DerivedLogger>
00106     requires std::derived_from<DerivedLogger, ILogger>
00107 void SetLogger(std::shared_ptr<DerivedLogger> logger) {
00108     std::lock_guard<std::mutex> lock(detail::GetLoggerMutex());
00109     detail::GetLoggerInstance() = std::move(logger);
00110 }
00111
00112 /**
00113  * @brief Set a logger instance that already conforms to the ILogger interface
00114  *
00115  * Thread-safe: uses a mutex to protect the global logger instance.
00116  * If nullptr is passed, sets a NullLogger instead.
00117  *
00118  * @param logger Shared pointer to the logger to set, or nullptr for NullLogger
00119  */
00120 inline void SetLogger(std::shared_ptr<ILogger> logger) {
00121     std::lock_guard<std::mutex> lock(detail::GetLoggerMutex());
00122     detail::GetLoggerInstance() = logger ? std::move(logger) : std::make_shared<NullLogger>();
00123 }
00124
00125 /**
00126  * @brief Reset the logger to the default NullLogger
00127  *
00128  * Convenience function that sets a NullLogger instance.
00129  * Thread-safe: uses a mutex to protect the global logger instance.
00130  */
00131 inline void SetDefaultLogger() {
00132     std::lock_guard<std::mutex> lock(detail::GetLoggerMutex());
00133     detail::GetLoggerInstance() = std::make_shared<NullLogger>();
00134 }
00135
00136
00137
00138 }  // namespace logging
```

9.27 logging/include/SpdlogWrapper.hpp File Reference

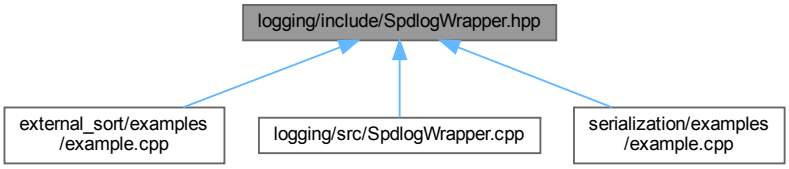
Wrapper for spdlog library using PIMPL pattern.

```
#include "ILogger.hpp"
#include <memory>
#include <string>
```

Include dependency graph for SpdlogWrapper.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class logging::SpdlogWrapper

  Concrete logger implementation wrapping the spdlog library.

Namespaces

- namespace logging

Enumerations

- enum class logging::SpdlogSinkType { logging::Console , logging::File , logging::Both }

  Types of logging sinks available for SpdlogWrapper.

9.27.1 Detailed Description

Wrapper for spdlog library using PIMPL pattern.

Definition in file SpdlogWrapper.hpp.

9.28 SpdlogWrapper.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file SpdlogWrapper.hpp
00003  * @brief Wrapper for spdlog library using PIMPL pattern
00004  */
00005
00006 #pragma once
00007
00008 #include "ILogger.hpp"
00009 #include <memory>
00010 #include <string>
00011
00012 namespace logging {
00013
00014 /**
00015  * @brief Types of logging sinks available for SpdlogWrapper
00016  */
00017 enum class SpdlogSinkType {
00018     Console,  ///< Log to console (stdout) with colors
00019     File,     ///< Log to file only
00020     Both      ///< Log to both console and file
00021 };
00022
00023 /**
00024  * @brief Concrete logger implementation wrapping the spdlog library
00025  *
00026  * Uses the PIMPL idiom to hide spdlog dependencies
00027  * from the header, reducing compilation dependencies and allowing the logging
00028  * library to be used without requiring spdlog to be installed.
00029  *
00030  * Supports multiple output targets:
00031  * - Console only (default)
00032  * - File only
00033  * - Both console and file simultaneously
00034  */
00035 class SpdlogWrapper : public ILogger {
00036 public:
00037     /**
00038      * @brief Construct with specified sink type
00039      * @param sink_type Type of sink to use (Console, File, or Both)
00040      * @param filename Filename for file sink (required if sink_type is File or Both)
00041      *
00042      * Example:
00043      * @code
00044      * // Console only
00045      * auto console_logger = std::make_shared<SpdlogWrapper>(SpdlogSinkType::Console);
00046      *
00047      * // File only
00048      * auto file_logger = std::make_shared<SpdlogWrapper>(SpdlogSinkType::File, "app.log");
00049      *
00050      * // Both console and file
00051      * auto both_logger = std::make_shared<SpdlogWrapper>(SpdlogSinkType::Both, "app.log");
00052      * @endcode
00053      */
00054     explicit SpdlogWrapper(const std::string& name = "spdlog",
00055                            SpdlogSinkType sink_type = SpdlogSinkType::Console,
00056                            const std::string& filename = "logs.log");
00057
00058     ~SpdlogWrapper();
00059     SpdlogWrapper(SpdlogWrapper&&) noexcept;
00060     SpdlogWrapper& operator=(SpdlogWrapper&&) noexcept;
00061     SpdlogWrapper(const SpdlogWrapper&) = delete;
00062     SpdlogWrapper& operator=(const SpdlogWrapper&) = delete;
00063
00064     /**
00065      * @brief Log an informational message
00066      * @param message The message to log
00067      */
00068     void LogInfo(const std::string& message) override;
00069
00070     /**
00071      * @brief Log a warning message
00072      * @param message The message to log
00073      */
00074     void LogWarning(const std::string& message) override;
00075
00076     /**
00077      * @brief Log an error message
00078      * @param message The message to log
00079      */
00080     void LogError(const std::string& message) override;
00081
00082 private:
```

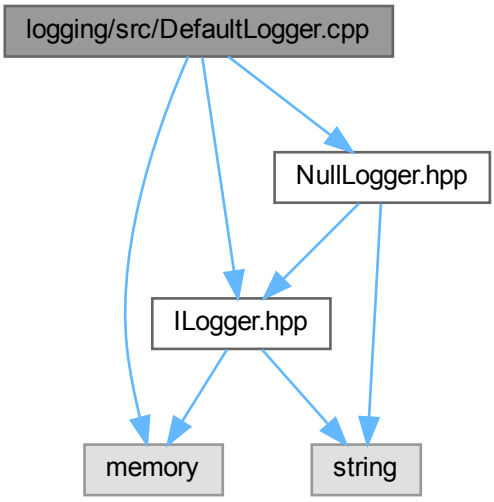
```
00083     class Impl;                    ///< Forward declaration of implementation class
00084     std::unique_ptr<Impl> pimpl_;  ///< Pointer to implementation
00085 };
00086
00087 }  // namespace logging
```

9.29 logging/src/DefaultLogger.cpp File Reference

Implementation of default logger factory function.

```
#include "ILogger.hpp"
#include "NullLogger.hpp"
#include <memory>
```

Include dependency graph for DefaultLogger.cpp:



Namespaces

- namespace logging

Functions

- std::shared_ptr< ILogger > logging::CreateDefaultLogger ()

  Factory function to create a default logger instance.

9.29.1 Detailed Description

Implementation of default logger factory function.

Definition in file DefaultLogger.cpp.

9.30 DefaultLogger.cpp

Go to the documentation of this file.

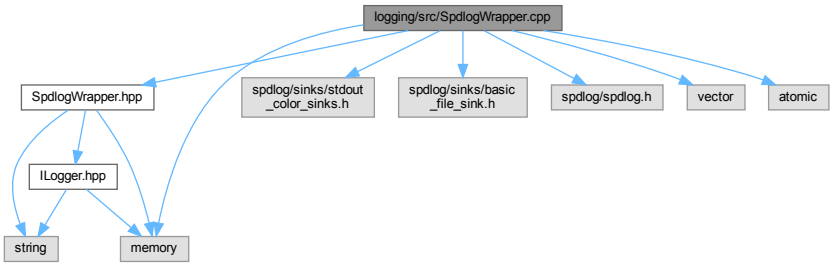
```
00001 /**
00002  * @file DefaultLogger.cpp
00003  * @brief Implementation of default logger factory function
00004  */
00005
00006 #include "ILogger.hpp"
00007 #include "NullLogger.hpp"
00008 #include <memory>
00009
00010 namespace logging {
00011
00012 std::shared_ptr<ILogger> CreateDefaultLogger() {
00013     return std::make_shared<NullLogger>();
00014 }
00015
00016 }  // namespace logging
```

9.31 logging/src/SpdlogWrapper.cpp File Reference

Implementation of the spdlog wrapper.

```
#include "SpdlogWrapper.hpp"
#include <spdlog/sinks/stdout_color_sinks.h>
#include <spdlog/sinks/basic_file_sink.h>
#include <spdlog/spdlog.h>
#include <vector>
#include <memory>
#include <atomic>
```

Include dependency graph for SpdlogWrapper.cpp:



Classes

- class logging::SpdlogWrapper::Impl

  Implementation class hiding spdlog dependencies.

Namespaces

- namespace logging

9.31.1 Detailed Description

Implementation of the spdlog wrapper.

Definition in file SpdlogWrapper.cpp.

9.32 SpdlogWrapper.cpp

Go to the documentation of this file.

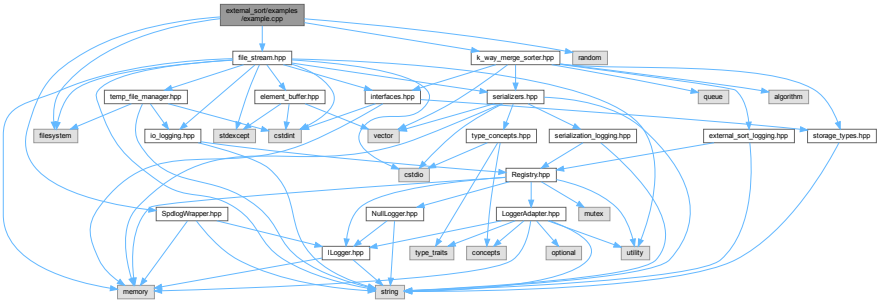
```
00001 /**
00002  * @file SpdlogWrapper.cpp
00003  * @brief Implementation of the spdlog wrapper
00004  */
00005
00006 #include "SpdlogWrapper.hpp"
00007 #include <spdlog/sinks/stdout_color_sinks.h>
00008 #include <spdlog/sinks/basic_file_sink.h>
00009 #include <spdlog/spdlog.h>
00010 #include <vector>
00011 #include <memory>
00012 #include <atomic>
00013
00014 namespace logging {
00015
00016 namespace {
00017 // Counter for unique logger names
00018 std::atomic<int> logger_counter{0};
00019 }  // namespace
00020
00021 /**
00022  * @brief Implementation class hiding spdlog dependencies
00023  *
00024  * This nested class contains the actual spdlog logger instance,
00025  * keeping spdlog headers out of the public interface.
00026  */
00027 class SpdlogWrapper::Impl {
00028 public:
00029     Impl() = delete;
00030
00031     /**
00032      * @brief Construct with specified sink configuration
00033      * @param sink_type Type of sink to create
00034      * @param filename Filename for file sink (if applicable)
00035      */
00036     Impl(const std::string& name, SpdlogSinkType sink_type, const std::string& filename) {
00037         std::vector<spdlog::sink_ptr> sinks;
00038
00039         switch (sink_type) {
00040             case SpdlogSinkType::Console: {
00041                 auto console_sink = std::make_shared<spdlog::sinks::stdout_color_sink_mt>();
00042                 sinks.push_back(console_sink);
00043                 break;
00044             }
00045             case SpdlogSinkType::File: {
00046                 if (filename.empty()) {
00047                     throw std::invalid_argument("Filename required for File sink type");
00048                 }
00049                 auto file_sink =
00050                     std::make_shared<spdlog::sinks::basic_file_sink_mt>(filename, true);
00051                 sinks.push_back(file_sink);
00052                 break;
00053             }
00054             case SpdlogSinkType::Both: {
00055                 if (filename.empty()) {
00056                     throw std::invalid_argument("Filename required for Both sink type");
00057                 }
00058                 auto console_sink = std::make_shared<spdlog::sinks::stdout_color_sink_mt>();
00059                 auto file_sink =
00060                     std::make_shared<spdlog::sinks::basic_file_sink_mt>(filename, true);
00061                 sinks.push_back(console_sink);
00062                 sinks.push_back(file_sink);
00063                 break;
00064             }
00065         }
00066
00067         logger = std::make_shared<spdlog::logger>(name + "_" + std::to_string(logger_counter++),
00068                                                   sinks.begin(), sinks.end());
00069         logger->set_level(spdlog::level::trace);
```

```
00070     }
00071
00072     std::shared_ptr<spdlog::logger> logger;  ///< The underlying spdlog logger
00073 };
00074
00075 SpdlogWrapper::SpdlogWrapper(const std::string& name, SpdlogSinkType sink_type,
00076                              const std::string& filename)
00077     : pimpl_(std::make_unique<Impl>(name, sink_type, filename)) {
00078 }
00079
00080 SpdlogWrapper::~SpdlogWrapper() = default;
00081
00082 SpdlogWrapper::SpdlogWrapper(SpdlogWrapper&&) noexcept = default;
00083
00084 SpdlogWrapper& SpdlogWrapper::operator=(SpdlogWrapper&&) noexcept = default;
00085
00086 void SpdlogWrapper::LogInfo(const std::string& message) {
00087     pimpl_->logger->info(message);
00088 }
00089
00090 void SpdlogWrapper::LogWarning(const std::string& message) {
00091     pimpl_->logger->warn(message);
00092 }
00093
00094 void SpdlogWrapper::LogError(const std::string& message) {
00095     pimpl_->logger->error(message);
00096 }
00097
00098 }  // namespace logging
```

9.33 external_sort/examples/example.cpp File Reference

```
#include "SpdlogWrapper.hpp"
#include "k_way_merge_sorter.hpp"
#include "file_stream.hpp"
#include <random>
#include <filesystem>
```

Include dependency graph for example.cpp:



Functions

- int main ()

9.33.1 Function Documentation

9.33.1.1 main()

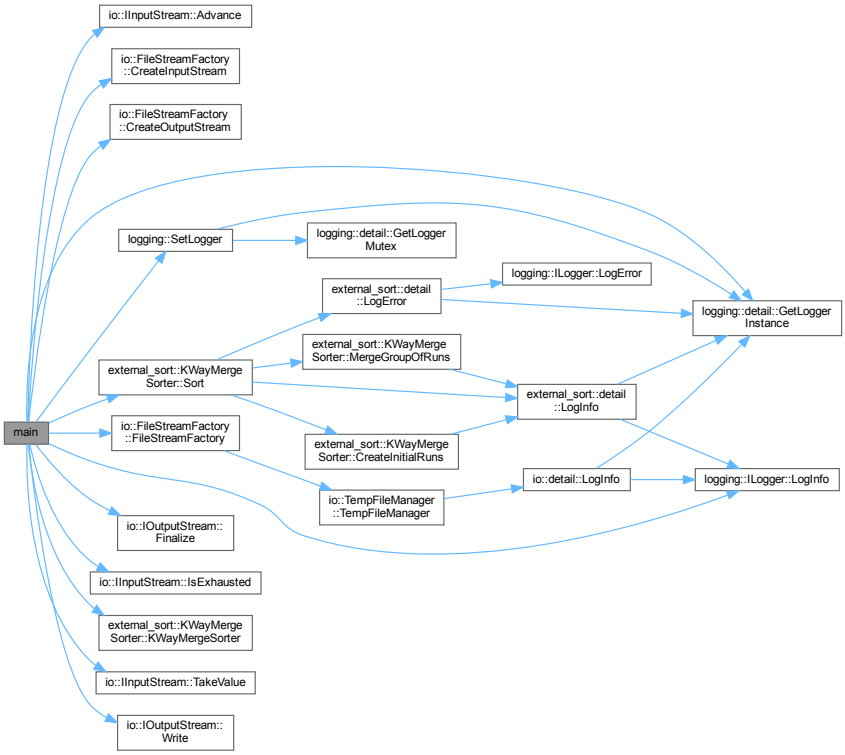
```
int main ( )
```

Definition at line 19 of file example.cpp.

```
00019           {
00020
00021     auto spdlog_logger = std::make_shared<logging::SpdlogWrapper>(
00022         "spdlog", logging::SpdlogSinkType::Both, "logs.log");
00023     logging::SetLogger(spdlog_logger);
00024     auto& logger = logging::detail::GetLoggerInstance();
00025
00026     // Prepare test directory and file names
00027     std::string test_dir = "external_sort_example_dir";
00028     std::filesystem::remove_all(test_dir);
00029     std::filesystem::create_directory(test_dir);
00030     FileStreamFactory<int> factory(test_dir);
00031     std::string input_id = "input_file";
00032     std::string output_id = "output_file";
00033
00034     // Generate and write random data to input file
00035     {
00036         auto output = factory.CreateOutputStream(input_id, 100);
00037         std::mt19937 gen(42);
00038         std::uniform_int_distribution<int> dist(0, 10000);
00039         for (int i = 0; i < 1000; ++i) {
00040             output->Write(dist(gen));
00041         }
00042         output->Finalize();
00043     }
00044
00045     // Sort the file using external sort
00046     KWayMergeSorter<int> sorter(factory, input_id, output_id, /*memory_bytes=*/sizeof(int) * 100,
00047                                 /*k=*/4, /*buffer_elems=*/50, /*ascending=*/true);
00048     sorter.Sort();
00049
00050     // Read and log sorted data
00051     {
00052         auto input = factory.CreateInputStream(output_id, 100);
00053         std::string log_msg = "Sorted data (first 20 elements): ";
00054         int count = 0;
00055         while (!input->IsExhausted() && count < 20) {
00056             log_msg += std::to_string(input->TakeValue()) + " ";
00057             input->Advance();
00058             ++count;
00059         }
00060         logger->LogInfo(log_msg);
00061     }
00062
00063     // Cleanup
00064     std::filesystem::remove_all(test_dir);
00065     return 0;
00066 }
```

References io::IInputStream< T >::Advance(), logging::Both, io::FileStreamFactory< T >::CreateInputStream(), io::FileStreamFactory< T >::CreateOutputStream(), io::FileStreamFactory< T >::FileStreamFactory(), io::IOutputStream< T >::Fina logging::detail::GetLoggerInstance(), io::IInputStream< T >::IsExhausted(), external_sort::KWayMergeSorter< T >::KWayMergeSor logging::ILogger::LogInfo(), logging::SetLogger(), external_sort::KWayMergeSorter< T >::Sort(), io::IInputStream< T >::TakeValue() and io::IOutputStream< T >::Write().

Here is the call graph for this function:



9.34 example.cpp

Go to the documentation of this file.

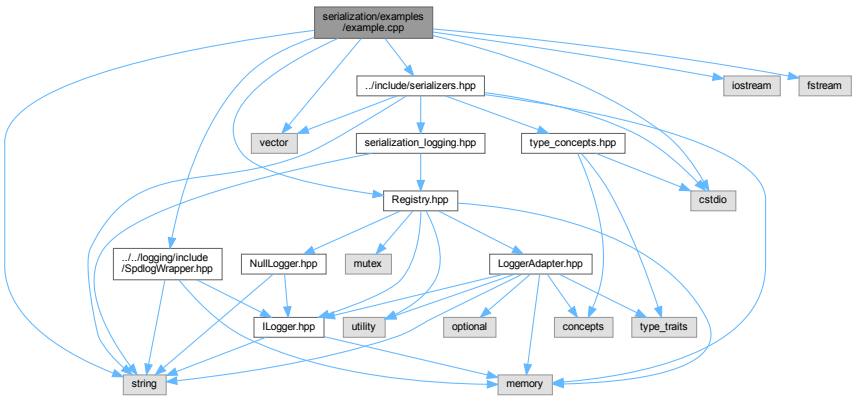
```
00001 /**
00002  * @file external_sort_example.cpp
00003  * @brief Example usage of the External Sort library
00004  *
00005  * This example demonstrates how to use the KWayMergeSorter with file-based storage
00006  * to sort a large file of integers. The same approach can be used for custom types
00007  * that support serialization.
00008  */
00009
00010 #include "SpdlogWrapper.hpp"
00011 #include "k_way_merge_sorter.hpp"
00012 #include "file_stream.hpp"
00013 #include <random>
00014 #include <filesystem>
00015
00016 using external_sort::KWayMergeSorter;
00017 using io::FileStreamFactory;
00018
00019 int main() {
00020
00021     auto spdlog_logger = std::make_shared<logging::SpdlogWrapper>(
00022         "spdlog", logging::SpdlogSinkType::Both, "logs.log");
00023     logging::SetLogger(spdlog_logger);
00024     auto& logger = logging::detail::GetLoggerInstance();
00025
00026     // Prepare test directory and file names
00027     std::string test_dir = "external_sort_example_dir";
00028     std::filesystem::remove_all(test_dir);
00029     std::filesystem::create_directory(test_dir);
00030     FileStreamFactory<int> factory(test_dir);
00031     std::string input_id = "input_file";
00032     std::string output_id = "output_file";
00033
00034     // Generate and write random data to input file
```

```
00035     {
00036         auto output = factory.CreateOutputStream(input_id, 100);
00037         std::mt19937 gen(42);
00038         std::uniform_int_distribution<int> dist(0, 10000);
00039         for (int i = 0; i < 1000; ++i) {
00040             output->Write(dist(gen));
00041         }
00042         output->Finalize();
00043     }
00044 
00045     // Sort the file using external sort
00046     KWayMergeSorter<int> sorter(factory, input_id, output_id, /*memory_bytes=*/sizeof(int) * 100,
00047                                 /*k=*/4, /*buffer_elems=*/50, /*ascending=*/true);
00048     sorter.Sort();
00049 
00050     // Read and log sorted data
00051     {
00052         auto input = factory.CreateInputStream(output_id, 100);
00053         std::string log_msg = "Sorted data (first 20 elements): ";
00054         int count = 0;
00055         while (!input->IsExhausted() && count < 20) {
00056             log_msg += std::to_string(input->TakeValue()) + " ";
00057             input->Advance();
00058             ++count;
00059         }
00060         logger->LogInfo(log_msg);
00061     }
00062 
00063     // Cleanup
00064     std::filesystem::remove_all(test_dir);
00065     return 0;
00066 }
```

9.35 serialization/examples/example.cpp File Reference

```
#include "../include/serializers.hpp"
#include "../../logging/include/Registry.hpp"
#include "../../logging/include/SpdlogWrapper.hpp"
#include <iostream>
#include <vector>
#include <string>
#include <cstdio>
#include <fstream>
```

Include dependency graph for example.cpp:



Classes

- class ConsoleLogger

Custom console logger with colored output and message prefixes.

- class FileLogger

  Non-copyable file logger that writes to a log file.

Functions

- int main ()

9.35.1 Function Documentation

9.35.1.1 main()

```
int main ( )
```

Definition at line 92 of file example.cpp.

```
00092           {
00093     // ================================================================================
00094     // LOGGER SETUP - Change this section to use a different logger
00095     // ================================================================================
00096
00097     // spdlog
00098     auto spdlog_logger = std::make_shared<logging::SpdlogWrapper>(
00099         "spdlog", logging::SpdlogSinkType::Both, "logs.log");
00100     logging::SetLogger(spdlog_logger);
00101
00102     // NullLogger
00103     // logging::SetDefaultLogger();
00104
00105     // Custom logger
00106     // logging::SetLogger(ConsoleLogger{});
00107
00108     // Custom non-copyable logger
00109     // auto file_logger = std::make_shared<FileLogger>(log_filename);
00110     // logging::SetLogger(file_logger);
00111     // ================================================================================
00112
00113     // Get logger instance for application logging
00114     auto& logger = logging::detail::GetLoggerInstance();
00115
00116     logger->LogInfo("=== Serialization Library Example ===");
00117     logger->LogInfo("Demonstrating various serialization scenarios");
00118
00119     const char* filename = "serialization_example.bin";
00120
00121     // Example 1: Basic string serialization
00122     logger->LogInfo("");
00123     logger->LogInfo("Example 1: std::string serialization");
00124     {
00125         std::string original = "Hello, Serialization!";
00126         logger->LogInfo("Original string: \"" + original + "\"");
00127
00128         auto serializer = CreateSerializer<std::string>();
00129
00130         // Serialize
00131         FILE* file = fopen(filename, "wb");
00132         if (file) {
00133             logger->LogInfo("Serializing...");
00134             bool success = serializer->Serialize(original, file);
00135             fclose(file);
00136             if (success) {
00137                 logger->LogInfo("Serialization: SUCCESS");
00138             } else {
00139                 logger->LogError("Serialization: FAILED");
00140             }
00141         }
00142
00143         // Deserialize
00144         std::string loaded;
00145         file = fopen(filename, "rb");
00146         if (file) {
00147             logger->LogInfo("Deserializing...");
00148             bool success = serializer->Deserialize(loaded, file);
00149             fclose(file);
00150             if (success) {
```

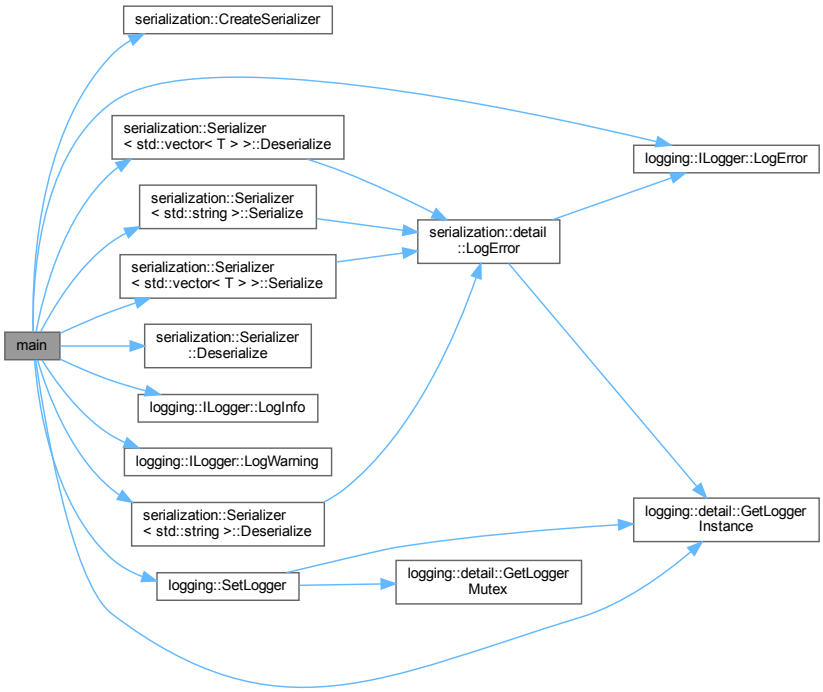
```
00151                 logger->LogInfo("Deserialization: SUCCESS");
00152                 logger->LogInfo("Loaded string: \"" + loaded + "\"");
00153
00154                 // Verification
00155                 if (original == loaded) {
00156                     logger->LogInfo("Verification passed: strings match");
00157                 } else {
00158                     logger->LogError("Verification failed: strings do NOT match");
00159                 }
00160             } else {
00161                 logger->LogError("Deserialization: FAILED");
00162             }
00163         }
00164     }
00165
00166     // Example 2: Vector of strings
00167     logger->LogInfo("");
00168     logger->LogInfo("Example 2: std::vector<std::string> serialization");
00169     {
00170         std::vector<std::string> original = {"First item", "Second item",
00171                                              "Third item with Unicode: ",
00172                                              "Fourth item with emoji "};
00173
00174         logger->LogInfo("Original vector (" + std::to_string(original.size()) + " elements):");
00175         for (size_t i = 0; i < original.size(); ++i) {
00176             logger->LogInfo("  [" + std::to_string(i) + "] \"" + original[i] + "\"");
00177         }
00178
00179         auto serializer = CreateSerializer<std::vector<std::string>>();
00180
00181         // Serialize
00182         FILE* file = fopen(filename, "wb");
00183         if (file) {
00184             logger->LogInfo("Serializing...");
00185             bool success = serializer->Serialize(original, file);
00186             fclose(file);
00187             if (success) {
00188                 logger->LogInfo("Serialization: SUCCESS");
00189             } else {
00190                 logger->LogError("Serialization: FAILED");
00191             }
00192         }
00193
00194         // Deserialize
00195         std::vector<std::string> loaded;
00196         file = fopen(filename, "rb");
00197         if (file) {
00198             logger->LogInfo("Deserializing...");
00199             bool success = serializer->Deserialize(loaded, file);
00200             fclose(file);
00201             if (success) {
00202                 logger->LogInfo("Deserialization: SUCCESS");
00203                 logger->LogInfo("Loaded vector (" + std::to_string(loaded.size()) + " elements):");
00204                 for (size_t i = 0; i < loaded.size(); ++i) {
00205                     logger->LogInfo("  [" + std::to_string(i) + "] \"" + loaded[i] + "\"");
00206                 }
00207
00208                 // Verification
00209                 if (original == loaded) {
00210                     logger->LogInfo("Verification passed: vectors match");
00211                 } else {
00212                     logger->LogError("Verification failed: vectors do NOT match");
00213                 }
00214             } else {
00215                 logger->LogError("Deserialization: FAILED");
00216             }
00217         }
00218     }
00219
00220     // Example 3: Nested vectors
00221     logger->LogInfo("");
00222     logger->LogInfo("Example 3: std::vector<std::vector<int>> serialization");
00223     {
00224         std::vector<std::vector<int>> original = {{1, 2, 3}, {4, 5, 6, 7}, {8, 9}};
00225
00226         logger->LogInfo("Original nested vector:");
00227         for (size_t i = 0; i < original.size(); ++i) {
00228             std::string row = "  Row " + std::to_string(i) + ": [";
00229             for (size_t j = 0; j < original[i].size(); ++j) {
00230                 row += std::to_string(original[i][j]);
00231                 if (j < original[i].size() - 1) {
00232                     row += ", ";
00233                 }
00234             }
00235             row += "]";
00236             logger->LogInfo(row);
00237         }
```

```
00238
00239         auto serializer = CreateSerializer<std::vector<std::vector<int>>>();
00240
00241         // Serialize
00242         FILE* file = fopen(filename, "wb");
00243         if (file) {
00244             logger->LogInfo("Serializing...");
00245             bool success = serializer->Serialize(original, file);
00246             fclose(file);
00247             if (success) {
00248                 logger->LogInfo("Serialization: SUCCESS");
00249             } else {
00250                 logger->LogError("Serialization: FAILED");
00251             }
00252         }
00253
00254         // Deserialize
00255         std::vector<std::vector<int>> loaded;
00256         file = fopen(filename, "rb");
00257         if (file) {
00258             logger->LogInfo("Deserializing...");
00259             bool success = serializer->Deserialize(loaded, file);
00260             fclose(file);
00261             if (success) {
00262                 logger->LogInfo("Deserialization: SUCCESS");
00263                 logger->LogInfo("Loaded nested vector:");
00264                 for (size_t i = 0; i < loaded.size(); ++i) {
00265                     std::string row = "  Row " + std::to_string(i) + ": [";
00266                     for (size_t j = 0; j < loaded[i].size(); ++j) {
00267                         row += std::to_string(loaded[i][j]);
00268                         if (j < loaded[i].size() - 1) {
00269                             row += ", ";
00270                         }
00271                     }
00272                     row += "]";
00273                     logger->LogInfo(row);
00274                 }
00275
00276                 // Verification
00277                 if (original == loaded) {
00278                     logger->LogInfo("Verification passed: nested vectors match");
00279                 } else {
00280                     logger->LogError("Verification failed: nested vectors do NOT match");
00281                 }
00282             } else {
00283                 logger->LogError("Deserialization: FAILED");
00284             }
00285         }
00286     }
00287
00288     // Example 4: Error handling demonstration
00289     logger->LogInfo("");
00290     logger->LogInfo("Example 4: Error handling");
00291     {
00292         struct TestData {
00293             int x, y, z;
00294         };
00295
00296         auto serializer = CreateSerializer<TestData>();
00297
00298         logger->LogInfo("Creating empty file to trigger error...");
00299         FILE* file = fopen(filename, "wb");
00300         if (file) {
00301             fclose(file);
00302         }
00303
00304         TestData data{0, 0, 0};
00305         file = fopen(filename, "rb");
00306         if (file) {
00307             logger->LogInfo("Attempting to deserialize from empty file (will fail)...");
00308             bool success = serializer->Deserialize(data, file);
00309             fclose(file);
00310             if (!success) {
00311                 logger->LogWarning("Deserialization failed as expected");
00312                 logger->LogWarning("See error message above from serialization library");
00313             }
00314         }
00315     }
00316
00317     // Cleanup and summary
00318     logger->LogInfo("");
00319     logger->LogInfo("=== Example completed ===");
00320     logger->LogInfo("All serialization operations were logged");
00321     logger->LogInfo("Summary: 3 successful examples + 1 error handling demo");
00322
00323     std::remove(filename);
00324     return 0;
```

```
00325 }
```

References logging::Both, serialization::CreateSerializer(), serialization::Serializer< std::string >::Deserialize(), serialization::Serializer< std::vector< T > >::Deserialize(), serialization::Serializer< T >::Deserialize(), logging::detail::GetLoggerIns logging::ILogger::LogError(), logging::ILogger::LogInfo(), logging::ILogger::LogWarning(), serialization::Serializer< std::string >::Seria serialization::Serializer< std::vector< T > >::Serialize(), and logging::SetLogger().

Here is the call graph for this function:



9.36 example.cpp

Go to the documentation of this file.

```
00001 /**
00002  * @file example_spdlog.cpp
00003  * @brief Example demonstrating serialization library with different logger backends
00004  *
00005  * This example shows the serialization library in action with various data types.
00006  * The logger can be easily swapped by changing the initialization code at the top of main().
00007  *
00008  * Features demonstrated:
00009  * - Simple string serialization
00010  * - Vector of strings with Unicode/emoji support
00011  * - Nested vectors (vector of vectors)
00012  * - Error handling and logging
00013  */
00014
00015 #include "../include/serializers.hpp"
00016 #include "../../logging/include/Registry.hpp"
00017 #include "../../logging/include/SpdlogWrapper.hpp"
00018
00019 #include <iostream>
00020 #include <vector>
00021 #include <string>
00022 #include <cstdio>
00023 #include <fstream>
00024
00025 using serialization::CreateSerializer;
```

```
00026
00027 /**
00028  * @brief Custom console logger with colored output and message prefixes
00029  */
00030 class ConsoleLogger {
00031 public:
00032     void info(const std::string& msg) {  // NOLINT(readability-identifier-naming)
00033         std::cout « "\033[32m[INFO]\033[0m " « msg « std::endl;
00034     }
00035
00036     void warn(const std::string& msg) {  // NOLINT(readability-identifier-naming)
00037         std::cout « "\033[33m[WARN]\033[0m " « msg « std::endl;
00038     }
00039
00040     void error(const std::string& msg) {  // NOLINT(readability-identifier-naming)
00041         std::cerr « "\033[31m[ERROR]\033[0m " « msg « std::endl;
00042     }
00043 };
00044
00045 /**
00046  * @brief Non-copyable file logger that writes to a log file
00047  *
00048  * This logger uses RAII to manage a file handle and is explicitly non-copyable
00049  * to demonstrate that LoggerAdapter can handle such types via shared_ptr.
00050  */
00051 class FileLogger {
00052 public:
00053     explicit FileLogger(const std::string& filename) : log_file_(filename, std::ios::app) {
00054         if (log_file_.is_open()) {
00055             log_file_ « "\n=== New logging session started ===\n";
00056         }
00057     }
00058
00059     ~FileLogger() {
00060         if (log_file_.is_open()) {
00061             log_file_ « "=== Logging session ended ===\n\n";
00062         }
00063     }
00064
00065     FileLogger(const FileLogger&) = delete;
00066     FileLogger& operator=(const FileLogger&) = delete;
00067     FileLogger(FileLogger&&) = default;
00068     FileLogger& operator=(FileLogger&&) = default;
00069
00070     void info(const std::string& msg) {  // NOLINT(readability-identifier-naming)
00071         if (log_file_.is_open()) {
00072             log_file_ « "[INFO] " « msg « std::endl;
00073         }
00074     }
00075
00076     void warn(const std::string& msg) {  // NOLINT(readability-identifier-naming)
00077         if (log_file_.is_open()) {
00078             log_file_ « "[WARN] " « msg « std::endl;
00079         }
00080     }
00081
00082     void error(const std::string& msg) {  // NOLINT(readability-identifier-naming)
00083         if (log_file_.is_open()) {
00084             log_file_ « "[ERROR] " « msg « std::endl;
00085         }
00086     }
00087
00088 private:
00089     std::ofstream log_file_;
00090 };
00091
00092 int main() {
00093     // ================================================================================
00094     // LOGGER SETUP - Change this section to use a different logger
00095     // ================================================================================
00096
00097     // spdlog
00098     auto spdlog_logger = std::make_shared<logging::SpdlogWrapper>(
00099         "spdlog", logging::SpdlogSinkType::Both, "logs.log");
00100     logging::SetLogger(spdlog_logger);
00101
00102     // NullLogger
00103     // logging::SetDefaultLogger();
00104
00105     // Custom logger
00106     // logging::SetLogger(ConsoleLogger{});
00107
00108     // Custom non-copyable logger
00109     // auto file_logger = std::make_shared<FileLogger>(log_filename);
00110     // logging::SetLogger(file_logger);
00111     // ================================================================================
00112
```

```
00113     // Get logger instance for application logging
00114     auto& logger = logging::detail::GetLoggerInstance();
00115
00116     logger->LogInfo("=== Serialization Library Example ===");
00117     logger->LogInfo("Demonstrating various serialization scenarios");
00118
00119     const char* filename = "serialization_example.bin";
00120
00121     // Example 1: Basic string serialization
00122     logger->LogInfo("");
00123     logger->LogInfo("Example 1: std::string serialization");
00124     {
00125         std::string original = "Hello, Serialization!";
00126         logger->LogInfo("Original string: \"" + original + "\"");
00127
00128         auto serializer = CreateSerializer<std::string>();
00129
00130         // Serialize
00131         FILE* file = fopen(filename, "wb");
00132         if (file) {
00133             logger->LogInfo("Serializing...");
00134             bool success = serializer->Serialize(original, file);
00135             fclose(file);
00136             if (success) {
00137                 logger->LogInfo("Serialization: SUCCESS");
00138             } else {
00139                 logger->LogError("Serialization: FAILED");
00140             }
00141         }
00142
00143         // Deserialize
00144         std::string loaded;
00145         file = fopen(filename, "rb");
00146         if (file) {
00147             logger->LogInfo("Deserializing...");
00148             bool success = serializer->Deserialize(loaded, file);
00149             fclose(file);
00150             if (success) {
00151                 logger->LogInfo("Deserialization: SUCCESS");
00152                 logger->LogInfo("Loaded string: \"" + loaded + "\"");
00153
00154                 // Verification
00155                 if (original == loaded) {
00156                     logger->LogInfo("Verification passed: strings match");
00157                 } else {
00158                     logger->LogError("Verification failed: strings do NOT match");
00159                 }
00160             } else {
00161                 logger->LogError("Deserialization: FAILED");
00162             }
00163         }
00164     }
00165
00166     // Example 2: Vector of strings
00167     logger->LogInfo("");
00168     logger->LogInfo("Example 2: std::vector<std::string> serialization");
00169     {
00170         std::vector<std::string> original = {"First item", "Second item",
00171                                              "Third item with Unicode: ",
00172                                              "Fourth item with emoji "};
00173
00174         logger->LogInfo("Original vector (" + std::to_string(original.size()) + " elements):");
00175         for (size_t i = 0; i < original.size(); ++i) {
00176             logger->LogInfo("  [" + std::to_string(i) + "] \"" + original[i] + "\"");
00177         }
00178
00179         auto serializer = CreateSerializer<std::vector<std::string>>();
00180
00181         // Serialize
00182         FILE* file = fopen(filename, "wb");
00183         if (file) {
00184             logger->LogInfo("Serializing...");
00185             bool success = serializer->Serialize(original, file);
00186             fclose(file);
00187             if (success) {
00188                 logger->LogInfo("Serialization: SUCCESS");
00189             } else {
00190                 logger->LogError("Serialization: FAILED");
00191             }
00192         }
00193
00194         // Deserialize
00195         std::vector<std::string> loaded;
00196         file = fopen(filename, "rb");
00197         if (file) {
00198             logger->LogInfo("Deserializing...");
00199             bool success = serializer->Deserialize(loaded, file);
```

```
00200             fclose(file);
00201             if (success) {
00202                 logger->LogInfo("Deserialization: SUCCESS");
00203                 logger->LogInfo("Loaded vector (" + std::to_string(loaded.size()) + " elements):");
00204                 for (size_t i = 0; i < loaded.size(); ++i) {
00205                     logger->LogInfo("  [" + std::to_string(i) + "] \"" + loaded[i] + "\"");
00206                 }
00207 
00208                 // Verification
00209                 if (original == loaded) {
00210                     logger->LogInfo("Verification passed: vectors match");
00211                 } else {
00212                     logger->LogError("Verification failed: vectors do NOT match");
00213                 }
00214             } else {
00215                 logger->LogError("Deserialization: FAILED");
00216             }
00217         }
00218     }
00219 
00220     // Example 3: Nested vectors
00221     logger->LogInfo("");
00222     logger->LogInfo("Example 3: std::vector<std::vector<int» serialization");
00223     {
00224         std::vector<std::vector<int» original = {{1, 2, 3}, {4, 5, 6, 7}, {8, 9}};
00225 
00226         logger->LogInfo("Original nested vector:");
00227         for (size_t i = 0; i < original.size(); ++i) {
00228             std::string row = "  Row " + std::to_string(i) + ": [";
00229             for (size_t j = 0; j < original[i].size(); ++j) {
00230                 row += std::to_string(original[i][j]);
00231                 if (j < original[i].size() - 1) {
00232                     row += ", ";
00233                 }
00234             }
00235             row += "]";
00236             logger->LogInfo(row);
00237         }
00238 
00239         auto serializer = CreateSerializer<std::vector<std::vector<int»>();
00240 
00241         // Serialize
00242         FILE* file = fopen(filename, "wb");
00243         if (file) {
00244             logger->LogInfo("Serializing...");
00245             bool success = serializer->Serialize(original, file);
00246             fclose(file);
00247             if (success) {
00248                 logger->LogInfo("Serialization: SUCCESS");
00249             } else {
00250                 logger->LogError("Serialization: FAILED");
00251             }
00252         }
00253 
00254         // Deserialize
00255         std::vector<std::vector<int» loaded;
00256         file = fopen(filename, "rb");
00257         if (file) {
00258             logger->LogInfo("Deserializing...");
00259             bool success = serializer->Deserialize(loaded, file);
00260             fclose(file);
00261             if (success) {
00262                 logger->LogInfo("Deserialization: SUCCESS");
00263                 logger->LogInfo("Loaded nested vector:");
00264                 for (size_t i = 0; i < loaded.size(); ++i) {
00265                     std::string row = "  Row " + std::to_string(i) + ": [";
00266                     for (size_t j = 0; j < loaded[i].size(); ++j) {
00267                         row += std::to_string(loaded[i][j]);
00268                         if (j < loaded[i].size() - 1) {
00269                             row += ", ";
00270                         }
00271                     }
00272                     row += "]";
00273                     logger->LogInfo(row);
00274                 }
00275 
00276                 // Verification
00277                 if (original == loaded) {
00278                     logger->LogInfo("Verification passed: nested vectors match");
00279                 } else {
00280                     logger->LogError("Verification failed: nested vectors do NOT match");
00281                 }
00282             } else {
00283                 logger->LogError("Deserialization: FAILED");
00284             }
00285         }
00286     }
```

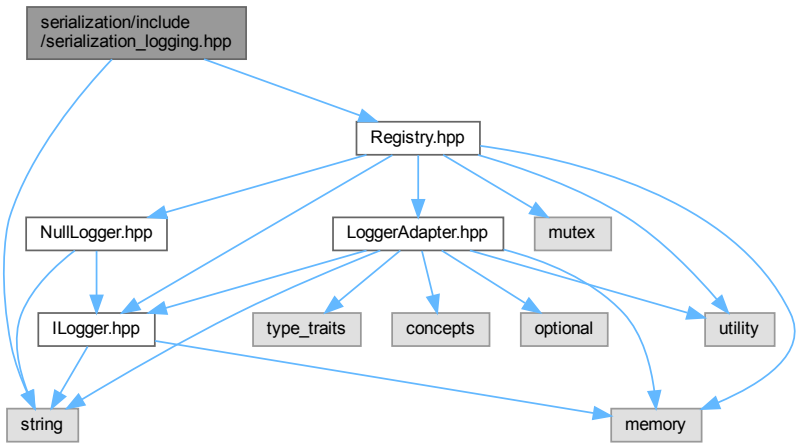
```
00287
00288     // Example 4: Error handling demonstration
00289     logger->LogInfo("");
00290     logger->LogInfo("Example 4: Error handling");
00291     {
00292         struct TestData {
00293             int x, y, z;
00294         };
00295
00296         auto serializer = CreateSerializer<TestData>();
00297
00298         logger->LogInfo("Creating empty file to trigger error...");
00299         FILE* file = fopen(filename, "wb");
00300         if (file) {
00301             fclose(file);
00302         }
00303
00304         TestData data{0, 0, 0};
00305         file = fopen(filename, "rb");
00306         if (file) {
00307             logger->LogInfo("Attempting to deserialize from empty file (will fail)...");
00308             bool success = serializer->Deserialize(data, file);
00309             fclose(file);
00310             if (!success) {
00311                 logger->LogWarning("Deserialization failed as expected");
00312                 logger->LogWarning("See error message above from serialization library");
00313             }
00314         }
00315     }
00316
00317     // Cleanup and summary
00318     logger->LogInfo("");
00319     logger->LogInfo("=== Example completed ===");
00320     logger->LogInfo("All serialization operations were logged");
00321     logger->LogInfo("Summary: 3 successful examples + 1 error handling demo");
00322
00323     std::remove(filename);
00324     return 0;
00325 }
```

9.37 serialization/include/serialization_logging.hpp File Reference

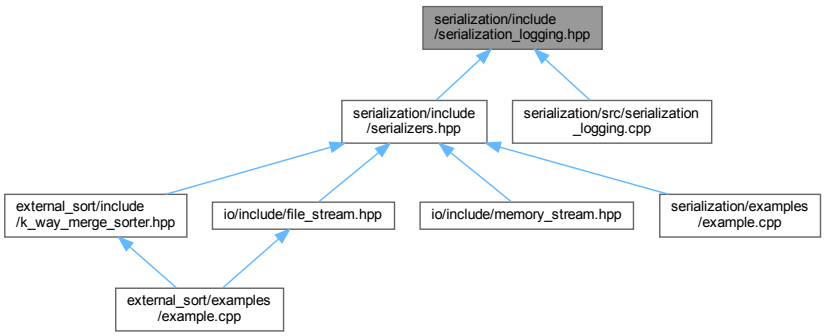
Logging integration for serialization library.

```
#include "Registry.hpp"
#include <string>
```

Include dependency graph for serialization_logging.hpp:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace serialization
- namespace serialization::detail

Functions

- void serialization::detail::LogError (const std::string &message)

  Log an error message from serialization operations.
- void serialization::detail::LogWarning (const std::string &message)

  Log a warning message from serialization operations.
- void serialization::detail::LogInfo (const std::string &message)

  Log an informational message from serialization operations.

9.37.1 Detailed Description

Logging integration for serialization library.

Definition in file serialization_logging.hpp.

9.38 serialization_logging.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file serialization_logging.hpp
00003  * @brief Logging integration for serialization library
00004  */
00005
00006 #pragma once
00007
00008 #include "Registry.hpp"
00009 #include <string>
00010
00011 namespace serialization {
00012 namespace detail {
00013
00014 /**
00015  * @brief Log an error message from serialization operations
```

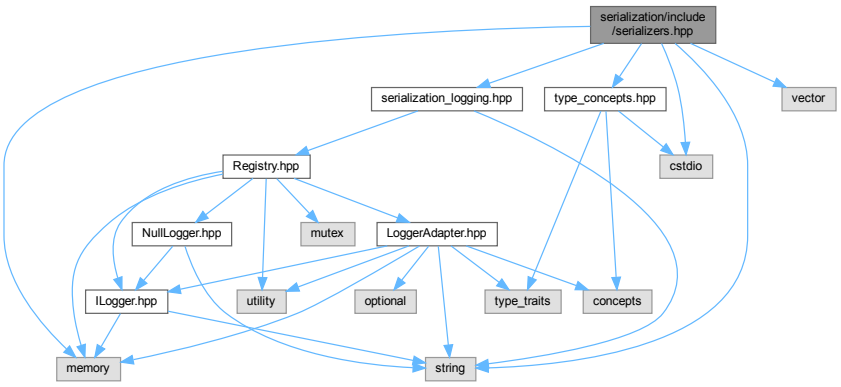
```
00016  * @param message The error message to log
00017  */
00018 inline void LogError(const std::string& message) {
00019     auto& logger = logging::detail::GetLoggerInstance();
00020     if (logger) {
00021         logger->LogError("[Serialization] " + message);
00022     }
00023 }
00024
00025 /**
00026  * @brief Log a warning message from serialization operations
00027  * @param message The warning message to log
00028  */
00029 inline void LogWarning(const std::string& message) {
00030     auto& logger = logging::detail::GetLoggerInstance();
00031     if (logger) {
00032         logger->LogWarning("[Serialization] " + message);
00033     }
00034 }
00035
00036 /**
00037  * @brief Log an informational message from serialization operations
00038  * @param message The info message to log
00039  */
00040 inline void LogInfo(const std::string& message) {
00041     auto& logger = logging::detail::GetLoggerInstance();
00042     if (logger) {
00043         logger->LogInfo("[Serialization] " + message);
00044     }
00045 }
00046
00047 }  // namespace detail
00048 }  // namespace serialization
```

9.39 serialization/include/serializers.hpp File Reference

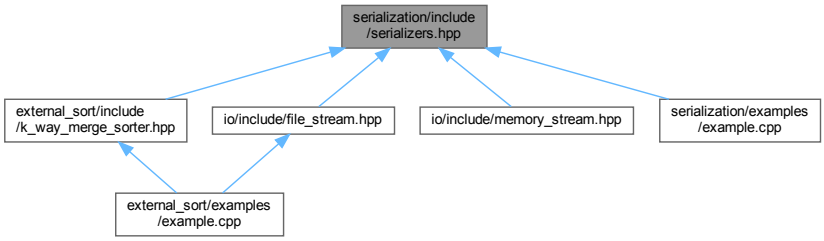
Serializers implementations for some types.

```
#include "serialization_logging.hpp"
#include "type_concepts.hpp"
#include <cstdio>
#include <memory>
#include <string>
#include <vector>
```

Include dependency graph for serializers.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class serialization::detail::ByteCountingFile

  Helper class to calculate serialized size by writing to a null device.
- class serialization::Serializer< T >

  Base interface for serializable types.
- class serialization::PodSerializer< T >

  Serializer for POD types.
- class serialization::CustomFunctionSerializer< T >

  Serializer for types with free function serialization.
- class serialization::MethodSerializer< T >

  Serializer for types with method serialization.
- class serialization::Serializer< std::string >
- class serialization::Serializer< std::vector< T > >

Namespaces

- namespace serialization
- namespace serialization::detail

Functions

- template<typename T , typename SerializeFunc >
  uint64_t serialization::detail::CalculateSizeBySerializing (const T &obj, SerializeFunc serialize_func)

  Calculate serialized size by actually serializing to /dev/null.
- template<typename T >
  std::unique_ptr< Serializer< T > > serialization::CreateSerializer ()

  Serializer factory.

9.39.1 Detailed Description

Serializers implementations for some types.

9.39.2 Adding Serialization Support

For std:: or external namespace types:

1. Add serialization::Serializer specialization in `type_concepts.hpp`:
```
template <>
class Serializer<std::string> {
public:
    using Specialized = std::true_type;

    bool Serialize(const std::string& obj, FILE* file) {
        ...
    }

    bool Deserialize(std::string& obj, FILE* file) {
        ...
    }

    uint64_t GetSerializedSize(const std::string& obj) {
        return sizeof(uint64_t) + obj.length();
    }
};
```

Alternative: Create a wrapper type with Serialize/Deserialize methods or Serialize/Deserialize free functions in its' namespace.

9.39.3 Performance-Critical Consideration: The GetSerializedSize() Method

A key feature of this library is the ability to pre-calculate the serialized size of an object via `GetSerialized↩Size()`. This is crucial for performance in algorithms like external sorting, which need to manage memory buffers efficiently.

The library uses two strategies to calculate this size:

1. **Optimized Path (Fast, O(1) or arithmetic)**: If a type provides a public const method `uint64_t Get↩SerializedSize() const`, it will be called directly. This is the **strongly recommended** approach for any custom type used in performance-sensitive code. This method should return the exact size in bytes that `Serialize()` would write.
```
struct MyData {
    uint64_t id;
    std::string name;

    bool Serialize(FILE* file) const { ... }
    bool Deserialize(FILE* file) { ... }

    // RECOMMENDED: Add this for optimal performance!
    uint64_t GetSerializedSize() const {
        return sizeof(id) + (sizeof(uint64_t) + name.length());
    }
};
```
2. **Fallback Path (Slow, involves I/O)**: If a type has `Serialize/Deserialize` methods but **lacks** `Get↩SerializedSize()`, the framework will fall back to a slower mechanism: it will perform a full serialization of the object to a null device (`/dev/null` or `NUL`) and count the bytes written. This works correctly but introduces significant I/O overhead.

9.39.3.1 Current Implementation Limitation

This library does **not** support automatic reflection to deduce the serialized size by recursively summing the sizes of a struct's members. For an aggregate struct like:

```
struct Aggregate {
    uint64_t field1;
    std::string field2;
};
```

...the library cannot automatically calculate its size as `sizeof(field1) + size_of(field2)`. The user is responsible for providing the serialization logic, and for performance, the `GetSerializedSize()` method. Implementing this automatic behavior would require advanced metaprogramming techniques that are beyond the scope of this project.

Conclusion: For any non-POD type that will be part of a large dataset, always implement the `GetSerialized↩Size()` method to avoid performance degradation. The benchmarks for this project demonstrate the significant difference in speed between the optimized and fallback paths.

See also

serialization::CreateSerializer

serialization::Serializer<std::vector<T>>

type_concepts.hpp for serialization concepts

Definition in file serializers.hpp.

9.40 serializers.hpp

Go to the documentation of this file.

```
00001 /**
00002  * @file serializers.hpp
00003  * @brief Serializers implementations for some types
00004  *
00005  * @section adding_serialization Adding Serialization Support
00006  *
00007  * **For std:: or external namespace types:**
00008  *
00009  * 1. Add serialization::Serializer specialization in `type_concepts.hpp`:
00010  * @code{.cpp}
00011  * template <>
00012  * class Serializer<std::string> {
00013  * public:
00014  *     using Specialized = std::true_type;
00015  *
00016  *     bool Serialize(const std::string& obj, FILE* file) {
00017  *         ...
00018  *     }
00019  *
00020  *     bool Deserialize(std::string& obj, FILE* file) {
00021  *         ...
00022  *     }
00023  *
00024  *     uint64_t GetSerializedSize(const std::string& obj) {
00025  *         return sizeof(uint64_t) + obj.length();
00026  *     }
00027  * };
00028  * @endcode
00029  *
00030  * **Alternative:** Create a wrapper type with Serialize/Deserialize methods
00031  * or Serialize/Deserialize free functions in its' namespace.
00032  *
00033  * @section performance_optimization Performance-Critical Consideration: The GetSerializedSize()
      Method
00034  *
00035  * A key feature of this library is the ability to pre-calculate the serialized size of an
00036  * object via `GetSerializedSize()`. This is crucial for performance in algorithms like
00037  * external sorting, which need to manage memory buffers efficiently.
00038  *
```

```
00039  * The library uses two strategies to calculate this size:
00040  *
00041  * 1.  **Optimized Path (Fast, O(1) or arithmetic)**: If a type provides a public const method
00042  *     `uint64_t GetSerializedSize() const`, it will be called directly. This is the
00043  *     **strongly recommended** approach for any custom type used in performance-sensitive
00044  *     code. This method should return the exact size in bytes that `Serialize()` would write.
00045  *
00046  *     @code{.cpp}
00047  *     struct MyData {
00048  *         uint64_t id;
00049  *         std::string name;
00050  *
00051  *         bool Serialize(FILE* file) const { ... }
00052  *         bool Deserialize(FILE* file) { ... }
00053  *
00054  *         // RECOMMENDED: Add this for optimal performance!
00055  *         uint64_t GetSerializedSize() const {
00056  *             return sizeof(id) + (sizeof(uint64_t) + name.length());
00057  *         }
00058  *     };
00059  *     @endcode
00060  *
00061  * 2.  **Fallback Path (Slow, involves I/O)**: If a type has `Serialize`/`Deserialize` methods
00062  *     but **lacks** `GetSerializedSize()`, the framework will fall back to a slower mechanism:
00063  *     it will perform a full serialization of the object to a null device (`/dev/null` or `NUL`)
00064  *     and count the bytes written. This works correctly but introduces significant I/O overhead.
00065  *
00066  * @subsection limitation Current Implementation Limitation
00067  *
00068  * This library does **not** support automatic reflection to deduce the serialized size by
00069  * recursively summing the sizes of a struct's members. For an aggregate struct like:
00070  *
00071  * @code{.cpp}
00072  * struct Aggregate {
00073  *     uint64_t field1;
00074  *     std::string field2;
00075  * };
00076  * @endcode
00077  *
00078  * ...the library cannot automatically calculate its size as `sizeof(field1) + size_of(field2)`.
00079  * The user is responsible for providing the serialization logic, and for performance, the
00080  * `GetSerializedSize()` method. Implementing this automatic behavior would require advanced
00081  * metaprogramming techniques that are beyond the scope of this project.
00082  *
00083  * **Conclusion**: For any non-POD type that will be part of a large dataset, always implement
00084  * the `GetSerializedSize()` method to avoid performance degradation. The benchmarks for this
00085  * project demonstrate the significant difference in speed between the optimized and fallback paths.
00086  *
00087  *
00088  * @see serialization::CreateSerializer
00089  * @see serialization::Serializer<std::vector<T>>
00090  * @see type_concepts.hpp for serialization concepts
00091  */
00092
00093 #pragma once
00094
00095 #include "serialization_logging.hpp"
00096 #include "type_concepts.hpp"
00097
00098 #include <cstdio>
00099 #include <memory>
00100 #include <string>
00101 #include <vector>
00102
00103
00104 namespace serialization {
00105
00106 namespace detail {
00107
00108 /**
00109  * @brief Helper class to calculate serialized size by writing to a null device
00110  *
00111  * This class provides a fallback mechanism to calculate serialized size
00112  * when the type doesn't provide GetSerializedSize() method.
00113  * It works by actually serializing the object and counting bytes written.
00114  */
00115 class ByteCountingFile {
00116 private:
00117     FILE* file_;
00118     uint64_t bytes_written_ = 0;
00119
00120 public:
00121     ByteCountingFile() {
00122 #ifdef _WIN32
00123         file_ = fopen("NUL", "wb");
00124 #else
00125         file_ = fopen("/dev/null", "wb");
```

```
00126 #endif
00127         if (!file_) {
00128             throw std::runtime_error("Failed to open null device for byte counting");
00129         }
00130     }
00131
00132     ~ByteCountingFile() {
00133         if (file_) {
00134             fclose(file_);
00135         }
00136     }
00137
00138     FILE* GetFile() { return file_; }
00139
00140     void TrackWrite(size_t bytes) {
00141         bytes_written_ += bytes;
00142     }
00143
00144     uint64_t GetBytesWritten() const { return bytes_written_; }
00145
00146     ByteCountingFile(const ByteCountingFile&) = delete;
00147     ByteCountingFile& operator=(const ByteCountingFile&) = delete;
00148 };
00149
00150 /**
00151  * @brief Calculate serialized size by actually serializing to /dev/null
00152  *
00153  * This is a fallback mechanism that works for any serializable type.
00154  * It temporarily serializes the object to count the bytes.
00155  *
00156  * **Performance Warning:** This function involves actual I/O operations to /dev/null.
00157  * While /dev/null is optimized by the OS, this is still slower than calculating
00158  * size arithmetically. Use this only when no faster alternative is available.
00159  *
00160  * @tparam T Type to serialize
00161  * @tparam SerializeFunc Function type that performs serialization
00162  * @param obj Object to calculate size for
00163  * @param serialize_func Function that performs serialization (signature: bool(const T&, FILE*))
00164  * @return Number of bytes that would be written
00165  * @throws std::runtime_error if /dev/null cannot be opened or serialization fails
00166  */
00167 template <typename T, typename SerializeFunc>
00168 uint64_t CalculateSizeBySerializing(const T& obj, SerializeFunc serialize_func) {
00169     // Save current position if we're using a real file
00170     FILE* null_file = nullptr;
00171 #ifdef _WIN32
00172     null_file = fopen("NUL", "wb");
00173 #else
00174     null_file = fopen("/dev/null", "wb");
00175 #endif
00176
00177     if (!null_file) {
00178         throw std::runtime_error("Failed to open null device for size calculation");
00179     }
00180
00181     // Get initial position
00182     int64_t start_pos = ftell(null_file);
00183
00184     // Serialize the object
00185     bool result = serialize_func(obj, null_file);
00186
00187     // Get final position
00188     int64_t end_pos = ftell(null_file);
00189     fclose(null_file);
00190
00191     if (!result) {
00192         throw std::runtime_error("Failed to serialize object for size calculation");
00193     }
00194
00195     // Return the number of bytes written
00196     return static_cast<uint64_t>(end_pos - start_pos);
00197 }
00198
00199 }  // namespace detail
00200
00201 /**
00202  * @brief Base interface for serializable types
00203  */
00204 template <typename T>
00205 class Serializer {
00206 public:
00207     using Specialized = std::false_type;
00208
00209     virtual ~Serializer() = default;
00210
00211     Serializer() = default;
00212     Serializer(const Serializer&) = default;
```

```
00213     Serializer& operator=(const Serializer&) = default;
00214     Serializer(Serializer&&) = default;
00215     Serializer& operator=(Serializer&&) = default;
00216 
00217     /**
00218      * @brief Serialize object into file
00219      *
00220      * @param obj Object ro serialize
00221      * @param file Pointer to file to write to
00222      * @return true, if serialization is successful
00223      */
00224     virtual bool Serialize(const T& obj, FILE* file) = 0;
00225 
00226     /**
00227      * @brief Deserialize object from file
00228      *
00229      * @param obj Object to deserialize to
00230      * @param file Pointer to file to read from
00231      * @return true, if deserialization is successful
00232      */
00233     virtual bool Deserialize(T& obj, FILE* file) = 0;
00234 
00235     /**
00236      * @brief Calculate the serialized size of an object in bytes
00237      *
00238      * This method computes the exact number of bytes that would be written
00239      * to a file when serializing the given object. This is useful for:
00240      * - Estimating storage requirements before serialization
00241      * - Tracking actual data size for complex types (e.g., types with dynamic memory)
00242      * - Generating correct amount of data to match target file sizes in benchmarks
00243      *
00244      * @param obj Object to calculate serialized size for
00245      * @return Number of bytes that would be written when serializing this object
00246      *
00247      * @note For POD types, this returns sizeof(T). For complex types (std::string,
00248      *       std::vector, custom types), this computes the actual serialized size
00249      *       including all dynamic data.
00250      *
00251      * @par Performance Optimization for Custom Types:
00252      * For types using MethodSerializer (types with Serialize/Deserialize methods),
00253      * there are two implementation paths:
00254      *
00255      * 1. **Optimized path (RECOMMENDED)**: If your type provides a `GetSerializedSize()`
00256      *    const method, it will be called directly. This is **much faster** as it only
00257      *    calculates the size without actual serialization.
00258      *
00259      * 2. **Fallback path**: If `GetSerializedSize()` is not provided, the serializer
00260      *    will perform actual serialization to /dev/null and count bytes. This works
00261      *    but is **significantly slower** (involves I/O operations).
00262      *
00263      * @par Example (Basic usage):
00264      * @code
00265      * auto serializer = serialization::CreateSerializer<Person>();
00266      * Person p{"Alice", 30, 170.5, 65, "123 Main St"};
00267      * uint64_t size = serializer->GetSerializedSize(p);
00268      * // size includes: sizeof(name length) + name data + age + height + weight +
00269      * //                sizeof(address length) + address data
00270      * @endcode
00271      *
00272      * @par Example (Optimized custom type):
00273      * @code
00274      * struct Person {
00275      *     std::string name;
00276      *     int32_t age;
00277      *     std::string address;
00278      *
00279      *     bool Serialize(FILE* file) const { ... }
00280      *     bool Deserialize(FILE* file) { ... }
00281      *
00282      *     // Add this method for optimal performance!
00283      *     uint64_t GetSerializedSize() const {
00284      *         return sizeof(size_t) + name.size()
00285      *              + sizeof(int32_t)
00286      *              + sizeof(size_t) + address.size();
00287      *     }
00288      * };
00289      * @endcode
00290      *
00291      * @warning If your type is used in performance-critical code (e.g., benchmarks,
00292      *          large datasets), strongly consider implementing GetSerializedSize()
00293      *          method to avoid I/O overhead.
00294      */
00295     virtual uint64_t GetSerializedSize(const T& obj) = 0;
00296 };
00297 
00298 /**
00299  * @brief Serializer for POD types
```

```
00300  *
00301  * @see serialization::PodSerializable
00302  */
00303 template <PodSerializable T>
00304 class PodSerializer : public Serializer<T> {
00305 public:
00306     bool Serialize(const T& obj, FILE* file) override {
00307         bool result = fwrite(&obj, sizeof(T), 1, file) == 1;
00308         if (!result) {
00309             detail::LogError("Failed to serialize POD type of size " + std::to_string(sizeof(T)));
00310         }
00311         return result;
00312     }
00313
00314     bool Deserialize(T& obj, FILE* file) override {
00315         bool result = fread(&obj, sizeof(T), 1, file) == 1;
00316         if (!result) {
00317             detail::LogError("Failed to deserialize POD type of size " + std::to_string(sizeof(T)));
00318         }
00319         return result;
00320     }
00321
00322     uint64_t GetSerializedSize([[maybe_unused]] const T& obj) override {
00323         return sizeof(T);
00324     }
00325 };
00326
00327 /**
00328  * @brief Serializer for types with free function serialization
00329  *
00330  * This serializer is used for types that provide Serialize() and Deserialize()
00331  * as free functions found via ADL (Argument Dependent Lookup).
00332  *
00333  * **Size Calculation:**
00334  * GetSerializedSize() performs actual serialization to /dev/null to count bytes.
00335  * This works reliably but involves I/O overhead.
00336  *
00337  * @par Performance Note:
00338  * If you need optimal performance for size calculation, consider using
00339  * MethodSerializable instead (with member functions) and adding a GetSerializedSize()
00340  * method, or create a specialized Serializer<YourType> with optimized GetSerializedSize().
00341  *
00342  * @tparam T Type with free Serialize() and Deserialize() functions
00343  * @see serialization::CustomSerializable
00344  */
00345 template <CustomSerializable T>
00346 class CustomFunctionSerializer : public Serializer<T> {
00347 public:
00348     bool Serialize(const T& obj, FILE* file) override {
00349         bool result = detail_adl::AdlSerialize(obj, file);
00350         result = result && (ferror(file) == 0);
00351         if (!result) {
00352             detail::LogError("Custom serialization failed for type");
00353         }
00354         return result;
00355     }
00356
00357     bool Deserialize(T& obj, FILE* file) override {
00358         bool result = detail_adl::AdlDeserialize(obj, file);
00359         result = result && (ferror(file) == 0) && (feof(file) == 0);
00360         if (!result) {
00361             detail::LogError("Custom deserialization failed for type");
00362         }
00363         return result;
00364     }
00365
00366     uint64_t GetSerializedSize([[maybe_unused]] const T& obj) override {
00367         // For custom serialization, try to calculate size by actually serializing
00368         // This works as a fallback when specialized GetSerializedSize is not available
00369         return detail::CalculateSizeBySerializing(obj, [this](const T& o, FILE* f) {
00370             return detail_adl::AdlSerialize(o, f);
00371         });
00372     }
00373 };
00374
00375 /**
00376  * @brief Serializer for types with method serialization
00377  *
00378  * This serializer is used for types that provide Serialize() and Deserialize()
00379  * methods as member functions. It supports two performance modes for size calculation:
00380  *
00381  * **Optimized Mode (Recommended):**
00382  * If the type provides a `GetSerializedSize() const` method returning uint64_t,
00383  * it will be called directly for fast size calculation without I/O overhead.
00384  *
00385  * **Fallback Mode:**
00386  * If GetSerializedSize() is not provided, the serializer will perform actual
```

```
00387  * serialization to /dev/null to count bytes. This works but is slower.
00388  *
00389  * @par Performance Tip:
00390  * For best performance, especially in benchmarks or when processing large datasets,
00391  * add a GetSerializedSize() method to your type:
00392  * @code
00393  * struct MyType {
00394  *     bool Serialize(FILE* file) const { ... }
00395  *     bool Deserialize(FILE* file) { ... }
00396  *
00397  *     // Add this for optimal performance:
00398  *     uint64_t GetSerializedSize() const {
00399  *         return ...; // Calculate total size
00400  *     }
00401  * };
00402  * @endcode
00403  *
00404  * @tparam T Type with Serialize() and Deserialize() member methods
00405  * @see serialization::MethodSerializable
00406  */
00407 template <MethodSerializable T>
00408 class MethodSerializer : public Serializer<T> {
00409 public:
00410     bool Serialize(const T& obj, FILE* file) override {
00411         return obj.Serialize(file);
00412     }
00413
00414     bool Deserialize(T& obj, FILE* file) override {
00415         return obj.Deserialize(file);
00416     }
00417
00418     uint64_t GetSerializedSize([[maybe_unused]] const T& obj) override {
00419         // For method-based serialization, first check if the type provides GetSerializedSize() method
00420         if constexpr (requires(const T& t) { { t.GetSerializedSize() } ->
      std::convertible_to<uint64_t>; }) {
00421             return obj.GetSerializedSize();
00422         } else {
00423             // Fallback: calculate size by actually serializing to /dev/null
00424             return detail::CalculateSizeBySerializing(obj, [](const T& o, FILE* f) {
00425                 return o.Serialize(f);
00426             });
00427         }
00428     }
00429 };
00430
00431 /**
00432  * @brief Serializer factory
00433  *
00434  * Creates appropriate serializer based on type traits
00435  */
00436 template <typename T>
00437 std::unique_ptr<Serializer<T> CreateSerializer() {
00438     static_assert(FileSerializable<T>, "Type must be serializable to be used with this library");
00439     if constexpr (PodSerializable<T>) {
00440         return std::make_unique<PodSerializer<T>();
00441     } else if constexpr (CustomSerializable<T>) {
00442         return std::make_unique<CustomFunctionSerializer<T>();
00443     } else if constexpr (MethodSerializable<T>) {
00444         return std::make_unique<MethodSerializer<T>();
00445     } else if constexpr (SpecializedSerializable<T>) {
00446         return std::make_unique<Serializer<T>();
00447     } else {
00448         throw std::logic_error("No serializer found for type " + std::string(typeid(T).name()));
00449     }
00450 }
00451
00452 template <>
00453 class Serializer<std::string> {
00454 public:
00455     using Specialized = std::true_type;
00456     bool Serialize(const std::string& obj, FILE* file) {
00457         uint64_t length = obj.length();
00458         if (fwrite(&length, sizeof(uint64_t), 1, file) != 1) {
00459             detail::LogError("Failed to write string length: " + std::to_string(length));
00460             return false;
00461         }
00462         if (fwrite(obj.data(), sizeof(char), length, file) != length) {
00463             detail::LogError("Failed to write string data of length: " + std::to_string(length));
00464             return false;
00465         }
00466         return true;
00467     }
00468
00469     bool Deserialize(std::string& obj, FILE* file) {
00470         uint64_t length;
00471         if (fread(&length, sizeof(uint64_t), 1, file) != 1) {
00472             detail::LogError("Failed to read string length");
```

```
00473             return false;
00474         }
00475         obj.resize(length);
00476         if (fread(&obj[0], sizeof(char), length, file) != length) {
00477             detail::LogError("Failed to read string data of length: " + std::to_string(length));
00478             return false;
00479         }
00480         return true;
00481     }
00482
00483     uint64_t GetSerializedSize(const std::string& obj) {
00484         // Size = sizeof(length field) + actual string data
00485         return sizeof(uint64_t) + obj.length() * sizeof(char);
00486     }
00487 };
00488
00489 template <typename T>
00490 class Serializer<std::vector<T>> {
00491 public:
00492     using Specialized = std::true_type;
00493
00494     bool Serialize(const std::vector<T>& obj, FILE* file) {
00495         uint64_t size = obj.size();
00496         if (fwrite(&size, sizeof(uint64_t), 1, file) != 1) {
00497             detail::LogError("Failed to write vector size: " + std::to_string(size));
00498             return false;
00499         }
00500
00501         auto item_serializer = CreateSerializer<T>();
00502         for (size_t i = 0; i < obj.size(); ++i) {
00503             if (!item_serializer->Serialize(obj[i], file)) {
00504                 detail::LogError("Failed to serialize vector element at index: " + std::to_string(i));
00505                 return false;
00506             }
00507         }
00508         return true;
00509     }
00510
00511     bool Deserialize(std::vector<T>& obj, FILE* file) {
00512         uint64_t size;
00513         if (fread(&size, sizeof(uint64_t), 1, file) != 1) {
00514             detail::LogError("Failed to read vector size");
00515             return false;
00516         }
00517
00518         obj.resize(size);
00519         auto item_serializer = CreateSerializer<T>();
00520         for (size_t i = 0; i < size; ++i) {
00521             if (!item_serializer->Deserialize(obj[i], file)) {
00522                 detail::LogError("Failed to deserialize vector element at index: " +
    std::to_string(i));
00523                 return false;
00524             }
00525         }
00526         return true;
00527     }
00528
00529     uint64_t GetSerializedSize(const std::vector<T>& obj) {
00530         // Size = sizeof(size field) + sum of element sizes
00531         uint64_t total_size = sizeof(uint64_t);
00532         auto item_serializer = CreateSerializer<T>();
00533         for (const auto& item : obj) {
00534             total_size += item_serializer->GetSerializedSize(item);
00535         }
00536         return total_size;
00537     }
00538 };
00539
00540 }  // namespace serialization
```

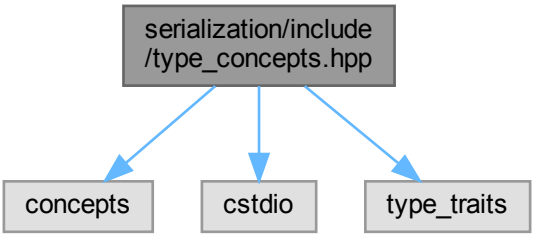
9.41 serialization/include/type_concepts.hpp File Reference

Concepts for checking types in serialization library.

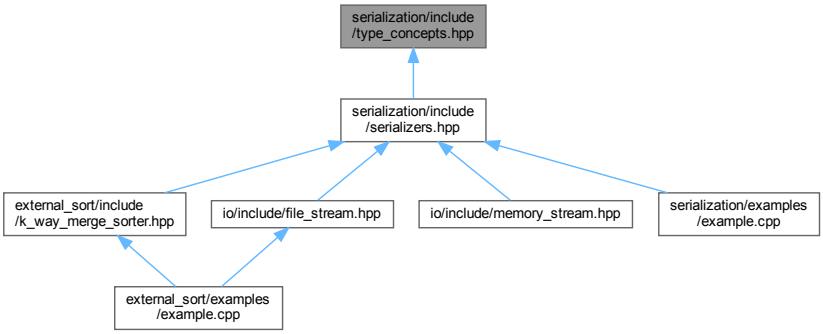
```
#include <concepts>
#include <cstdio>
```

```
#include <type_traits>
```

Include dependency graph for type_concepts.hpp:



This graph shows which files directly or indirectly include this file:



Namespaces

- namespace serialization
- namespace serialization::detail_adl

  Helpers for ADL - defined in a nested namespace.

Concepts

- concept serialization::PodSerializable

  Concept for POD types that can be serialized through fwrite/fread.
- concept serialization::CustomSerializable

  Concept for types with custom serialization through free functions.
- concept serialization::MethodSerializable

  Concept for types with serialization methods.
- concept serialization::SpecializedSerializable

  Concept for types with Serializer specialization.
- concept serialization::FileSerializable

  Concept for types that can be serialized into files.

Functions

- template<typename T >
  auto serialization::detail_adl::AdlSerialize (const T &obj, FILE *file) -> decltype(Serialize(obj, file))
- template<typename T >
  auto serialization::detail_adl::AdlDeserialize (T &obj, FILE *file) -> decltype(Deserialize(obj, file))

9.41.1 Detailed Description

Concepts for checking types in serialization library.

Definition in file type_concepts.hpp.

9.42 type_concepts.hpp

Go to the documentation of this file.

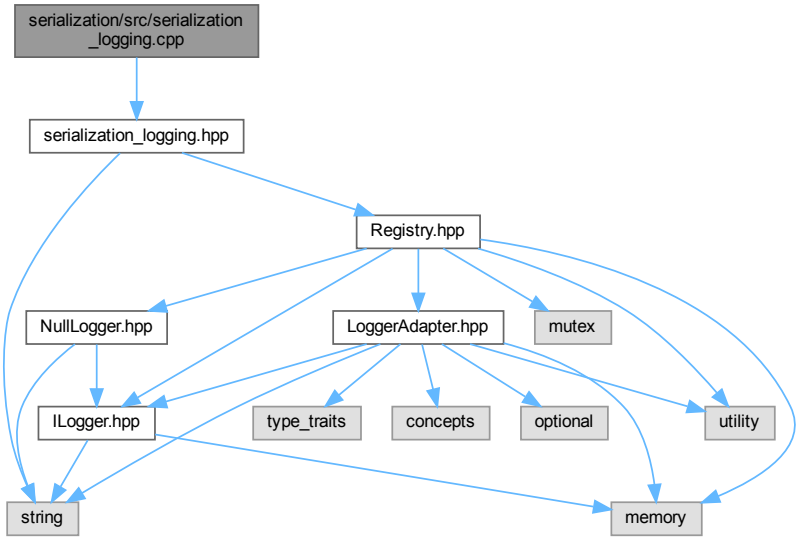
```
00001 /**
00002  * @file type_concepts.hpp
00003  * @brief Concepts for checking types in serialization library
00004  */
00005
00006 #pragma once
00007
00008 #include <concepts>
00009 #include <cstdio>
00010 #include <type_traits>
00011
00012 namespace serialization {
00013
00014 // Forward declaration of Serializer
00015 template <typename T>
00016 class Serializer;
00017
00018 /**
00019  * @brief Concept for POD types that can be serialized through fwrite/fread
00020  *
00021  * Checks that type is trivially copyable and has standard layout.
00022  * Such types can be directly written to/read from files using binary I/O.
00023  *
00024  * Examples: int, double, struct with POD members.
00025  *
00026  * @see serialization::FileSerializable
00027  * @see serialization::CustomSerializable
00028  * @see serialization::MethodSerializable
00029  */
00030 template <typename T>
00031 concept PodSerializable = std::is_trivially_copyable_v<T> && std::is_standard_layout_v<T>;
00032
00033 /**
00034  * @brief Helpers for ADL - defined in a nested namespace
00035  *
00036  * These functions use unqualified lookup which enables ADL.
00037  * They will find Serialize/Deserialize either:
00038  * - In the namespace of the type T (via ADL)
00039  * - In the serialization namespace (via normal lookup)
00040  */
00041 namespace detail_adl {
00042 template <typename T>
00043 auto AdlSerialize(const T& obj, FILE* file) -> decltype(Serialize(obj, file)) {
00044     return Serialize(obj, file);
00045 }
00046
00047 template <typename T>
00048 auto AdlDeserialize(T& obj, FILE* file) -> decltype(Deserialize(obj, file)) {
00049     return Deserialize(obj, file);
00050 }
00051 }  // namespace detail_adl
00052
00053 /**
00054  * @brief Concept for types with custom serialization through free functions
00055  *
00056  * Checks that type has free functions Serialize and Deserialize.
00057  * These functions can be defined either:
```

```
00058  * 1. In the serialization namespace (for std types like std::string, std::vector)
00059  * 2. In the same namespace as the type for ADL (Argument-Dependent Lookup)
00060  *
00061  * Functions must have the following signatures:
00062  * @code{.cpp}
00063  * bool Serialize(const T& obj, FILE* file);
00064  * bool Deserialize(T& obj, FILE* file);
00065  * @endcode
00066  *
00067  * - Serialize writes obj into file and returns true on success
00068  * - Deserialize reads one object from file into obj and returns true on success
00069  *
00070  * @see serialization::FileSerializable
00071  * @see serialization::PodSerializable
00072  * @see serialization::MethodSerializable
00073  */
00074 template <typename T>
00075 concept CustomSerializable = requires(const T& obj, T& obj_mut, FILE* file) {
00076     { detail_adl::AdlSerialize(obj, file) } -> std::same_as<bool>;
00077     { detail_adl::AdlDeserialize(obj_mut, file) } -> std::same_as<bool>;
00078 };
00079
00080 /**
00081  * @brief Concept for types with serialization methods
00082  *
00083  * Checks that type has member methods Serialize and Deserialize.
00084  *
00085  * Methods must have the following signatures:
00086  * @code{.cpp}
00087  * bool Serialize(FILE* file) const;
00088  * bool Deserialize(FILE* file);
00089  * @endcode
00090  *
00091  * - Serialize writes the object into file and returns true on success
00092  * - Deserialize reads one object from file and returns true on success
00093  *
00094  * @see serialization::FileSerializable
00095  * @see serialization::PodSerializable
00096  * @see serialization::CustomSerializable
00097  */
00098 template <typename T>
00099 concept MethodSerializable = requires(const T& obj, T& obj_mut, FILE* file) {
00100     { obj.Serialize(file) } -> std::same_as<bool>;
00101     { obj_mut.Deserialize(file) } -> std::same_as<bool>;
00102 };
00103
00104 /**
00105  * @brief Concept for types with Serializer specialization
00106  *
00107  * Checks that type has a specialization of class Serializer
00108  *
00109  * Specialization must define
00110  * @code{.cpp}
00111  * using Specialized = std::true_type;
00112  * @endcode
00113  *
00114  * @see serialization::Serializer
00115  * @see serialization::Serializer<std::string>
00116  */
00117 template <typename T>
00118 concept SpecializedSerializable = Serializer<T>::Specialized::value;
00119
00120 /**
00121  * @brief Concept for types that can be serialized into files.
00122  *
00123  * Types must satisfy one of the following concepts:
00124  * - serialization::PodSerializable - POD types with fwrite/fread
00125  * - serialization::CustomSerializable - types with free Serialize/Deserialize functions
00126  * - serialization::MethodSerializable - types with Serialize/Deserialize methods
00127  *
00128  * @see serialization::PodSerializable
00129  * @see serialization::CustomSerializable
00130  * @see serialization::MethodSerializable
00131  */
00132 template <typename T>
00133 concept FileSerializable = PodSerializable<T> || CustomSerializable<T> || MethodSerializable<T> ||
      SpecializedSerializable<T>;
00134
00135 }  // namespace serialization
```

9.43 serialization/src/serialization_logging.cpp File Reference

Implementation of logging functions for serialization library.

`#include "serialization_logging.hpp"`
Include dependency graph for serialization_logging.cpp:



9.43.1 Detailed Description

Implementation of logging functions for serialization library.

Definition in file serialization_logging.cpp.

9.44 serialization_logging.cpp

Go to the documentation of this file.

```
00001 /**
00002  * @file serialization_logging.cpp
00003  * @brief Implementation of logging functions for serialization library
00004  */
00005
00006 #include "serialization_logging.hpp"
00007
00008 // The actual implementations are in the header (inline functions)
00009 // This file exists to allow the library to be compiled separately if needed
```