Program Listing for File logger.hpp

Return to documentation for file (include/holoscan/logger/logger.hpp)

/* * SPDX-FileCopyrightText: Copyright (c) 2022-2024 NVIDIA CORPORATION & AFFILIATES. All rights reserved. * SPDX-License-Identifier: Apache-2.0 * * Licensed under the Apache License, Version 2.0 (the "License"); * you may not use this file except in compliance with the License. * You may obtain a copy of the License at * * http://www.apache.org/licenses/LICENSE-2.0 * * Unless required by applicable law or agreed to in writing, software * distributed under the License is distributed on an "AS IS" BASIS, * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. * See the License for the specific language governing permissions and * limitations under the License. */ #ifndef HOLOSCAN_LOGGER_LOGGER_HPP #define HOLOSCAN_LOGGER_LOGGER_HPP #include <fmt/format.h> #include <fmt/ranges.h> *// allows fmt to format std::array, std::vector, etc.* #include <chrono> #include <memory> #include <string> #include <string_view> #include <utility> #define HOLOSCAN_LOG_LEVEL_TRACE 0 #define HOLOSCAN_LOG_LEVEL_DEBUG 1 #define HOLOSCAN_LOG_LEVEL_INFO 2 #define HOLOSCAN_LOG_LEVEL_WARN 3 #define HOLOSCAN_LOG_LEVEL_ERROR 4 #define HOLOSCAN_LOG_LEVEL_CRITICAL 5 #define HOLOSCAN_LOG_LEVEL_OFF 6 *// Please define (or call CMake's `target_compile_definitions` with) HOLOSCAN_LOG_ACTIVE_LEVEL // before including <holoscan/holoscan.h> to one of the above levels if you want to skip logging at // a certain level at compile time. // // E.g., // #define HOLOSCAN_LOG_ACTIVE_LEVEL 3 // #include <holoscan/holoscan.h> // ... // // Then, it will only log at the WARN(3)/ERROR(4)/CRITICAL(5) levels. // // You can define GXF_LOG_ACTIVE_LEVEL in your build system. For instance, in CMake, use: // // target_compile_definitions(my_target PRIVATE HOLOSCAN_LOG_ACTIVE_LEVEL=3) // // This sets the active logging level to WARN(3) for the target `my_target`. // // Alternatively, define HOLOSCAN_LOG_ACTIVE_LEVEL at compile time by passing // `-DHOLOSCAN_LOG_ACTIVE_LEVEL=3` directly to the compiler. // Workaround for zero-arguments // (https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0306r2.html) // If __VA_OPT__ is supported (since C++20), we could use it to use compile-time format string check // : FMT_STRING(format) // (https://fmt.dev/latest/api.html#compile-time-format-string-checks)* #define HOLOSCAN_LOG_CALL(level, ...) \ ::holoscan::Logger::log(\ __FILE__, __LINE__, static_cast<const char*>(__FUNCTION__), level, __VA_ARGS__) *// clang-format off* #if HOLOSCAN_LOG_ACTIVE_LEVEL <= HOLOSCAN_LOG_LEVEL_TRACE #

define HOLOSCAN_LOG_TRACE(...) HOLOSCAN_LOG_CALL(::holoscan::LogLevel::TRACE, __VA_ARGS__) #else # define HOLOSCAN_LOG_TRACE(...) (void)0 #endif #if HOLOSCAN_LOG_ACTIVE_LEVEL <= HOLOSCAN_LOG_LEVEL_DEBUG # define HOLOSCAN_LOG_DEBUG(...) HOLOSCAN_LOG_CALL(::holoscan::LogLevel::DEBUG, __VA_ARGS__) #else # define HOLOSCAN_LOG_DEBUG(...) (void)0 #endif #if HOLOSCAN_LOG_ACTIVE_LEVEL <= HOLOSCAN_LOG_LEVEL_INFO # define HOLOSCAN_LOG_INFO(...) HOLOSCAN_LOG_CALL(::holoscan::LogLevel::INFO, __VA_ARGS__) #else # define HOLOSCAN_LOG_INFO(...) (void)0 #endif #if HOLOSCAN_LOG_ACTIVE_LEVEL <= HOLOSCAN_LOG_LEVEL_WARN # define HOLOSCAN_LOG_WARN(...) HOLOSCAN_LOG_CALL(::holoscan::LogLevel::WARN, __VA_ARGS__) #else # define HOLOSCAN_LOG_WARN(...) (void)0 #endif #if HOLOSCAN_LOG_ACTIVE_LEVEL <= HOLOSCAN_LOG_LEVEL_ERROR # define HOLOSCAN_LOG_ERROR(...) HOLOSCAN_LOG_CALL(::holoscan::LogLevel::ERROR, __VA_ARGS__) #else # define HOLOSCAN_LOG_ERROR(...) (void)0 #endif #if HOLOSCAN_LOG_ACTIVE_LEVEL <= HOLOSCAN_LOG_LEVEL_CRITICAL # define HOLOSCAN_LOG_CRITICAL(...) \ HOLOSCAN_LOG_CALL(::holoscan::LogLevel::CRITICAL, __VA_ARGS__) #else # define HOLOSCAN_LOG_CRITICAL(...) (void)0 #endif *// clang-format on* namespace holoscan { enum class LogLevel { TRACE = 0, DEBUG = 1, INFO = 2, WARN = 3, ERROR = 4, CRITICAL = 5, OFF = 6, }; class Logger { public: static void set_level(LogLevel level, bool* is_overridden_by_env = nullptr); static LogLevel level(); static void set_pattern(std::string pattern = "", bool* is_overridden_by_env = nullptr); static std::string& pattern(); template <typename FormatT, typename... ArgsT> static void log(const char* file, int line, const char* function_name, LogLevel level, const FormatT& format, ArgsT&&... args) { log_message(file, line, function_name, level, format, fmt::make_args_checked<ArgsT...>(format, std::forward<ArgsT>(args)...)); } template <typename FormatT, typename... ArgsT> static void log(LogLevel level, const FormatT& format, ArgsT&&... args) { log_message(level, format, fmt::make_args_checked<ArgsT...>(format, std::forward<ArgsT>(args)...)); } static bool log_pattern_set_by_user; static bool log_level_set_by_user; private: static void log_message(const char* file, int line, const char* function_name, LogLevel level, fmt::string_view format, fmt::format_args args); static void log_message(LogLevel level, fmt::string_view format, fmt::format_args args); }; void set_log_level(LogLevel level); inline LogLevel log_level() { return Logger::level(); } void set_log_pattern(std::string pattern = ""); template <typename FormatT, typename... ArgsT> inline void log_trace(const FormatT& format, ArgsT&&... args) {

```
Logger::log(LogLevel::TRACE, format, std::forward<ArgsT>(args)...); } template <typename FormatT, typename... ArgsT> inline void log_debug(const FormatT& format, ArgsT&&... args) { Logger::log(LogLevel::DEBUG, format, std::forward<ArgsT>(args)...); } template <typename FormatT, typename... ArgsT> inline void log_info(const FormatT& format, ArgsT&&... args) { Logger::log(LogLevel::INFO, format, std::forward<ArgsT>(args)...); } template <typename FormatT, typename... ArgsT> inline void log_warn(const FormatT& format, ArgsT&&... args) { Logger::log(LogLevel::WARN, format, std::forward<ArgsT>(args)...); } template <typename FormatT, typename... ArgsT> inline void log_error(const FormatT& format, ArgsT&&... args) { Logger::log(LogLevel::ERROR, format, std::forward<ArgsT>(args)...); } template <typename FormatT, typename... ArgsT> inline void log_critical(const FormatT& format, ArgsT&&... args) { Logger::log(LogLevel::CRITICAL, format, std::forward<ArgsT>(args)...); } template <typename FormatT, typename... ArgsT> inline void log_message(const char* file, int line, const char* function_name, LogLevel level, const FormatT& format, ArgsT&&... args) { Logger::log(file, line, function_name, level, format, std::forward<ArgsT>(args)...); } } // namespace holoscan #endif/* HOLOSCAN_LOGGER_LOGGER_HPP */
```