Program Listing for File parameter.hpp

Return to documentation for file (`include/holoscan/core/parameter.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_CORE_PARAMETER_HPP #define HOLOSCAN_CORE_PARAMETER_HPP //
Include fmt library for specialized formatting #include <fmt/format.h> #include
<fmt/ranges.h> // allows fmt to format std::array, std::vector, etc. #include <any>
#include <functional> #include <iostream> #include <optional> #include <string>
#include <typeinfo> #include <utility> #include "./type_traits.hpp" namespace
holoscan { enum class ParameterFlag { kNone = 0, kOptional = 1, kDynamic = 2, };
template <typename ValueT> class MetaParameter { public: MetaParameter() =
default; explicit MetaParameter(const ValueT& value) : value_(value) {} explicit
MetaParameter(ValueT&& value) : value_(std::move(value)) {} MetaParameter(const
ValueT& value, const char* key, const char* headline, const char* description,
ParameterFlag flag) : key_(key), headline_(headline), description_(description),
flag_(flag), value_(value) {} MetaParameter& operator=(const ValueT& value) { value_
= value; return *this; } MetaParameter&& operator=(ValueT&& value) { value_ =
std::move(value); return std::move(*this); } const std::string& key() const { return
key_; } const std::string& headline() const { return headline_; } const std::string&
description() const { return description_; } const ParameterFlag& flag() const { return
flag_; } bool has_value() const { return value_.has_value(); } ValueT& get() { if
(value_.has_value()) { return value_.value(); } else { throw
std::runtime_error(fmt::format("MetaParameter: value for '{}' is not set", key_)); } }
std::optional<ValueT>& try_get() { return value_; } template <typename PointerT =
ValueT, typename = std::enable_if_t<holoscan::is_shared_ptr_v<PointerT> ||
std::is_pointer_v<PointerT>>> holoscan::remove_pointer_t<PointerT>* operator->() {
if constexpr (holoscan::is_shared_ptr_v<PointerT>) { return get().get(); } else { return
get(); } } template <typename PointerT = ValueT, typename =
std::enable_if_t<holoscan::is_shared_ptr_v<PointerT> ||
std::is_pointer_v<PointerT>>> holoscan::remove_pointer_t<PointerT> operator*() {
return *get(); } void set_default_value() { if (!value_.has_value()) { value_ =
default_value_; } } ValueT& default_value() { if (default_value_.has_value()) { return
default_value_.value(); } else { throw std::runtime_error(
fmt::format("MetaParameter: default value for '{}' is not set", key_)); } } bool
has_default_value() const { return default_value_.has_value(); } operator ValueT&() {
return get(); } private: friend class ComponentSpec; friend class OperatorSpec;
std::string key_; std::string headline_; std::string description_; ParameterFlag flag_ =
ParameterFlag::kNone; std::optional<ValueT> value_; std::optional<ValueT>
default_value_; }; } // namespace holoscan // ------------------------------------------------------------
----------------------------------- // holoscan::Parameter<T> format support for fmt::format // //
After defining the holoscan::Parameter<T> class, we need to specialize the fmt::formatter
// struct for the holoscan::Parameter<T> type to use it with fmt::format. Here, we
specialize the // fmt::formatter struct for the holoscan::Parameter<T> type before
including the // holoscan/logger/logger.hpp file. // -----------------------------------------------------
----------------------------------------- namespace fmt { template <typename typeT> struct
formatter<holoscan::Parameter<typeT>> : formatter<typeT> { template <typename
FormatContext> auto format(const holoscan::Parameter<typeT>& v,
FormatContext& ctx) const { return
formatter<typeT>::format(const_cast<holoscan::Parameter<typeT>&>(v).get(), ctx); }
}; } // namespace fmt // Include the logger.hpp after the fmt::formatter specialization
#include "holoscan/logger/logger.hpp" // -------------------------------------------------------------
-------------------------------- // Define ParameterWrapper class #include "./arg.hpp"
namespace holoscan { class ParameterWrapper { public: ParameterWrapper() =
default; template <typename typeT> explicit
ParameterWrapper(Parameter<typeT>& param) : type_(&typeid(typeT)),
arg_type_(ArgType::create<typeT>()), value_(&param),
storage_ptr_(static_cast<void*>(&param)) {} ParameterWrapper(std::any value,
const std::type_info* type, const ArgType& arg_type, void* storage_ptr = nullptr) :
type_(type), arg_type_(arg_type), value_(std::move(value)), storage_ptr_(storage_ptr)
{} const std::type_info& type() const { if (type_) { return *type_; } return typeid(void); }
const ArgType& arg_type() const { return arg_type_; } std::any& value() { return
value_; } void* storage_ptr() const { return storage_ptr_; } private: const
std::type_info* type_ = nullptr; ArgType arg_type_; std::any value_; void*
```

```
storage_ptr_ = nullptr; }; } // namespace holoscan #endif/*
HOLOSCAN_CORE_PARAMETER_HPP */
```