Program Listing for File type_traits.hpp

Return to documentation for file (include/holoscan/core/type_traits.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_TYPES_HPP #define HOLOSCAN_CORE_TYPES_HPP #include <array> #include <cstdint> #include <memory> #include <type_traits> #include <vector> #include "./forward_def.hpp" namespace holoscan { struct scalar_type : std::integral_constant<int, 0> {}; struct vector_type : std::integral_constant<int, 1> {}; struct array_type : std::integral_constant<int, 2> {}; *// base_type/base_type_t* template <typename T, typename Enable = void> struct base_type { using type = std::decay_t<T>; }; template <typename T> struct base_type<T, typename std::enable_if_t<std::is_base_of_v<Resource, std::decay_t<T>>>> { using type = Resource; }; template <typename T> struct base_type<T, typename std::enable_if_t<std::is_base_of_v<Condition, std::decay_t<T>>>> { using type = Condition; }; template <typename T> using base_type_t = typename base_type<T>::type; *// type_info* template <typename T> struct _type_info { using container_type = scalar_type; using element_type = base_type_t<T>; using derived_type = std::decay_t<T>; static constexpr int32_t dimension = 0; }; template <typename T> struct _type_info<std::shared_ptr<T>> { using container_type = scalar_type; using element_type = std::shared_ptr<base_type_t<T>>; using derived_type = std::decay_t<T>; static constexpr int32_t dimension = 0; }; template <typename T> struct _type_info<std::vector<T>> { using container_type = vector_type; using element_type = base_type_t<T>; using derived_type = std::decay_t<T>; static constexpr int32_t dimension = 1; }; template <typename T> struct _type_info<std::vector<std::shared_ptr<T>>> { using container_type = vector_type; using element_type = std::shared_ptr<base_type_t<T>>; using derived_type = std::decay_t<T>; static constexpr int32_t dimension = 1; }; template <typename T> struct _type_info<std::vector<std::vector<T>>> { using container_type

```
= vector_type; using element_type = base_type_t<T>; using derived_type =
std::decay_t<T>; static constexpr int32_t dimension = 2; }; template <typename T>
struct _type_info<std::vector<std::vector<std::shared_ptr<T>>>> { using
container_type = vector_type; using element_type =
std::shared_ptr<base_type_t<T>>; using derived_type = std::decay_t<T>; static
constexpr int32_t dimension = 2; }; template <typename T, std::size_t N> struct
_type_info<std::array<T, N>> { using container_type = array_type; using
element_type = base_type_t<T>; using derived_type = std::decay_t<T>; static
constexpr int32_t dimension = 1; }; template <typename T> struct
_type_info<Parameter<std::shared_ptr<T>>> { using container_type = scalar_type;
using element_type = std::shared_ptr<base_type_t<T>>; using derived_type =
std::decay_t<T>; static constexpr int32_t dimension = 0; }; template <typename T>
struct _type_info<Parameter<std::vector<T>>> { using container_type = vector_type;
using element_type = base_type_t<T>; using derived_type = std::decay_t<T>; static
constexpr int32_t dimension = 1; }; template <typename T> struct
_type_info<Parameter<std::vector<std::vector<T>>>> { using container_type =
vector_type; using element_type = base_type_t<T>; using derived_type =
std::decay_t<T>; static constexpr int32_t dimension = 2; }; template <typename T,
std::size_t N> struct _type_info<Parameter<std::array<T, N>>> { using
container_type = array_type; using element_type = base_type_t<T>; using
derived_type = std::decay_t<T>; static constexpr int32_t dimension = 1; }; template
<typename T> using type_info = _type_info<std::decay_t<T>>; // remove_pointer/remove_pointer_t // (This implementation removes both std::shared_ptr and raw pointers.) template <typename T> struct remove_pointer { using type =
std::remove_pointer_t<T>; }; template <typename T> struct
remove_pointer<std::shared_ptr<T>> { using type = T; }; template <typename T>
using remove_pointer_t = typename remove_pointer<T>::type; //
is_scalar/is_scalar_t/is_scalar_v template <typename T> struct is_scalar :
std::integral_constant< bool, std::is_same_v<typename
type_info<std::decay_t<T>>::container_type, scalar_type>> { }; template <typename
T> using is_scalar_t = typename is_scalar<T>::type; template <typename T> inline
constexpr bool is_scalar_v = is_scalar<std::decay_t<T>>::value; // is_vector/is_vector_v
template <typename T> struct is_vector : public std::integral_constant< bool,
std::is_same_v<typename type_info<T>::container_type, vector_type>> {}; template
<typename T> using is_vector_t = typename is_vector<T>::type; template <typename
T> inline constexpr bool is_vector_v = is_vector<std::decay_t<T>>::value; //
```

is_array/is_array_t/is_array_v template <typename T> struct is_array : public std::integral_constant< bool, std::is_same_v<typename type_info<T>::container_type, array_type>> {}; template <typename T> using is_array_t = typename is_array<T>::type; template <typename T> inline constexpr bool is_array_v = is_array<std::decay_t<T>>::value; *// is_shared_ptr/is_shared_ptr_t/is_shared_ptr_v* template <typename T> struct is_shared_ptr : public std::integral_constant<bool, false> {}; template <typename T> struct is_shared_ptr<std::shared_ptr<T>> : public std::integral_constant<bool, true> {}; template <typename T> using is_shared_ptr_t = typename is_shared_ptr<T>::type; template <typename T> inline constexpr bool is_shared_ptr_v = is_shared_ptr<std::decay_t<T>>::value; *// is_yaml_convertable/is_yaml_convertable_t/is_yaml_convertable_v* template <typename T> struct is_yaml_convertable : public std::integral_constant<bool, ! (std::is_same_v<IOSpec*, base_type_t<T>> || std::is_same_v<std::shared_ptr<Resource>, base_type_t<T>> || std::is_same_v<std::shared_ptr<Condition>, base_type_t<T>>)> { }; template <typename T> using is_yaml_convertable_t = typename is_yaml_convertable<T>::type; template <typename T> inline constexpr bool is_yaml_convertable_v = is_yaml_convertable<std::decay_t<T>>::value; *// is_one_of_v* template <typename T, typename... ArgsT> inline constexpr bool is_one_of_v = ((std::is_same_v<T, ArgsT> || ...)); *// is_one_of_derived_v* template <typename T, typename... ArgsT> inline constexpr bool is_one_of_derived_v = ((std::is_base_of_v<ArgsT, T> || ...)); *// dimension_of_v* template <typename T> inline constexpr int32_t dimension_of_v = type_info<T>::dimension; } *// namespace holoscan* #endif/* HOLOSCAN_CORE_TYPES_HPP */

© Copyright 2022-2024, NVIDIA.. PDF Generated on 06/06/2024