Program Listing for File argument_setter.hpp

Return to documentation for file (include/holoscan/core/argument_setter.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_ARGUMENT_SETTER_HPP #define HOLOSCAN_CORE_ARGUMENT_SETTER_HPP #include <any> #include <complex> #include <functional> #include <iostream> #include <memory> #include <string> #include <type_traits> #include <typeindex> #include <typeinfo> #include <unordered_map> #include <utility> #include <vector> #include <common/logger.hpp> #include "../utils/yaml_parser.hpp" #include "./arg.hpp" #include "./common.hpp" #include "./condition.hpp" #include "./parameter.hpp" #include "./resource.hpp" #include "./type_traits.hpp" namespace holoscan { class ArgumentSetter { public: using SetterFunc = std::function<void(ParameterWrapper&, Arg&)>; inline static SetterFunc none_argument_setter = [](ParameterWrapper& param_wrap, Arg& arg) { (void)param_wrap; (void)arg; HOLOSCAN_LOG_ERROR("Unable to handle parameter: {}", arg.name()); }; static ArgumentSetter& get_instance(); static void set_param(ParameterWrapper& param_wrap, Arg& arg) { auto& instance = get_instance(); const std::type_index index = std::type_index(param_wrap.type()); const SetterFunc& func = instance.get_argument_setter(index); func(param_wrap, arg); } template <typename typeT> static void ensure_type() { auto& instance = get_instance(); instance.add_argument_setter<typeT>(); } SetterFunc& get_argument_setter(std::type_index index) { if (function_map_.find(index) == function_map_.end()) { HOLOSCAN_LOG_WARN("No argument setter for type '{}' exists", index.name()); return ArgumentSetter::none_argument_setter; } auto& handler = function_map_[index]; return handler; } template <typename typeT> void add_argument_setter(SetterFunc func) { function_map_.try_emplace(std::type_index(typeid(typeT)), func); } void
```

```
add_argument_setter(std::type_index index, SetterFunc func) { function_map_.try_emplace(index, func); } template <typename typeT> void add_argument_setter() { function_map_.try_emplace( std::type_index(typeid(typeT)), [](ParameterWrapper& param_wrap, Arg& arg) { std::any& any_param = param_wrap.value(); // Note that the type of any_param is Parameter<typeT>*, not Parameter<typeT>. auto& param = *std::any_cast<Parameter<typeT>*>(any_param); // If arg has no name and value, that indicates that we want to set the default value for // the native operator if it is not specified. if (arg.name().empty() && !arg.has_value()) { auto& param = *std::any_cast<Parameter<typeT>*>(any_param); param.set_default_value(); return; } std::any& any_arg = arg.value(); const auto& arg_type = arg.arg_type(); auto element_type = arg_type.element_type(); auto container_type = arg_type.container_type(); try { switch (container_type) { case ArgContainerType::kNative: { switch (element_type) { // Handle the argument with 'kInt64' type differently because the argument might // come from Python, and Python only has 'int' type ('int64_t' in C++). case ArgElementType::kInt64: { if constexpr (holoscan::is_one_of_v<typeT, bool, int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, uint32_t, uint64_t, float, double>) { auto& arg_value = std::any_cast<int64_t&>(any_arg); param = static_cast<typeT>(arg_value); } else { HOLOSCAN_LOG_ERROR( "Unable to convert argument type '{}' to parameter type '{}' for '{}'", any_arg.type().name(), typeid(typeT).name(), arg.name()); } break; } // Handle the argument with 'kFloat64' type differently because the argument might // come from Python, and Python only has 'float' type ('double' in C++). case ArgElementType::kFloat64: { if constexpr (holoscan::is_one_of_v<typeT, bool, int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, uint32_t, uint64_t, float, double>) { auto& arg_value = std::any_cast<double&>(any_arg); param = static_cast<typeT>(arg_value); } else { HOLOSCAN_LOG_ERROR( "Unable to convert argument type '{}' to parameter type '{}' for '{}'", any_arg.type().name(), typeid(typeT).name(), arg.name()); } break; } case ArgElementType::kBoolean: case ArgElementType::kInt8: case ArgElementType::kInt16: case ArgElementType::kInt32: case ArgElementType::kUnsigned8: case ArgElementType::kUnsigned16: case ArgElementType::kUnsigned32: case ArgElementType::kUnsigned64: case ArgElementType::kFloat32: case ArgElementType::kComplex64: case ArgElementType::kComplex128: case ArgElementType::kString: case ArgElementType::kIOSpec: { if constexpr (holoscan::is_one_of_v<typeT, bool, int8_t, int16_t, int32_t, int64_t, uint8_t, uint16_t, uint32_t, uint64_t, float, double, std::complex<float>, std::complex<double>, std::string, IOSpec*>) { auto& arg_value
```

```
= std::any_cast<typeT&>(any_arg); param = arg_value; } else { HOLOSCAN_LOG_ERROR( "Unable to convert argument type '{}' to parameter type '{}' for '{}'", any_arg.type().name(), typeid(typeT).name(), arg.name()); } break; } case ArgElementType::kCondition: { if constexpr (std::is_same_v<typename holoscan::type_info<typeT>::element_type, std::shared_ptr<Condition>> && holoscan::type_info<typeT>::dimension == 0) { auto& arg_value = std::any_cast<std::shared_ptr<Condition>&>(any_arg); auto converted_value = std::dynamic_pointer_cast< typename holoscan::type_info<typeT>::derived_type> (arg_value); // Initialize the condition in case the condition created by // Fragment::make_condition<T>() is added to the operator as an argument. // TODO: would like this to be assigned to the same entity as the operator if (converted_value) { converted_value->initialize(); } param = converted_value; } break; } case ArgElementType::kResource: { if constexpr (std::is_same_v<typename holoscan::type_info<typeT>::element_type, std::shared_ptr<Resource>> && holoscan::type_info<typeT>::dimension == 0) { auto& arg_value = std::any_cast<std::shared_ptr<Resource>&>(any_arg); auto converted_value = std::dynamic_pointer_cast< typename holoscan::type_info<typeT>::derived_type> (arg_value); // Initialize the resource in case the resource created by // Fragment::make_resource<T>() is added to the operator as an argument. // TODO: would like this to be assigned to the same entity as the operator if (converted_value) { converted_value->initialize(); } param = converted_value; } break; } case ArgElementType::kYAMLNode: { if constexpr (!holoscan::is_yaml_convertable_v<typeT>) { HOLOSCAN_LOG_ERROR( "YAML conversion for key '{}' is not supported for type '{}'", arg.name(), typeid(typeT).name()); } else { auto node = std::any_cast<YAML::Node>(any_arg); typeT value = YAMLNodeParser<typeT>::parse(node); param = value; } break; } case ArgElementType::kHandle: break; case ArgElementType::kCustom: { HOLOSCAN_LOG_ERROR( "Unable to convert argument type '{}' to parameter type '{}' for '{}'", any_arg.type().name(), typeid(typeT).name(), arg.name()); break; } } break; } case ArgContainerType::kVector: { switch (element_type) { case ArgElementType::kBoolean: case ArgElementType::kInt8: case ArgElementType::kInt16: case ArgElementType::kInt32: case ArgElementType::kInt64: case ArgElementType::kUnsigned8: case ArgElementType::kUnsigned16: case ArgElementType::kUnsigned32: case ArgElementType::kUnsigned64: case ArgElementType::kFloat32: case ArgElementType::kFloat64: case ArgElementType::kComplex64: case
```

```
ArgElementType::kComplex128: case ArgElementType::kString: case
ArgElementType::kIOSpec: { if constexpr (holoscan::is_one_of_v< typeT,
std::vector<bool>, std::vector<int8_t>, std::vector<int16_t>, std::vector<int32_t>,
std::vector<int64_t>, std::vector<uint8_t>, std::vector<uint16_t>,
std::vector<uint32_t>, std::vector<uint64_t>, std::vector<float>, std::vector<double>,
std::vector<std::complex<float>>, std::vector<std::complex<double>>,
std::vector<std::string>, std::vector<std::vector<bool>>,
std::vector<std::vector<int8_t>>, std::vector<std::vector<int16_t>>,
std::vector<std::vector<int32_t>>, std::vector<std::vector<int64_t>>,
std::vector<std::vector<uint8_t>>, std::vector<std::vector<uint16_t>>,
std::vector<std::vector<uint32_t>>, std::vector<std::vector<uint64_t>>,
std::vector<std::vector<float>>, std::vector<std::vector<double>>,
std::vector<std::vector<std::complex<float>>>,
std::vector<std::vector<std::complex<double>>>,
std::vector<std::vector<std::string>>, std::vector<IOSpec*>>) { auto& arg_value =
std::any_cast<typeT&>(any_arg); param = arg_value; } else {
HOLOSCAN_LOG_ERROR( "Unable to convert argument type '{}' to parameter type
'{}' for '{}'", any_arg.type().name(), typeid(typeT).name(), arg.name()); } break; } case
ArgElementType::kHandle: case ArgElementType::kYAMLNode: break; case
ArgElementType::kCondition: { if constexpr (std::is_same_v<typename
holoscan::type_info<typeT>::element_type, std::shared_ptr<Condition>> &&
holoscan::type_info<typeT>::dimension == 1) { auto& arg_value =
std::any_cast<std::vector<std::shared_ptr<Condition>>&>(any_arg); typeT
converted_value; converted_value.reserve(arg_value.size()); for (auto&
arg_value_item : arg_value) { auto&& condition = std::dynamic_pointer_cast<
typename holoscan::type_info<typeT>::derived_type>(arg_value_item); // Initialize
the condition in case the condition created by // Fragment::make_condition<T>() is
added to the operator as an argument. // TODO: would like this to be assigned to the
same entity as the operator if (condition) { condition->initialize(); }
converted_value.push_back(condition); } param = converted_value; } break; } case
ArgElementType::kResource: { if constexpr (std::is_same_v<typename
holoscan::type_info<typeT>::element_type, std::shared_ptr<Resource>> &&
holoscan::type_info<typeT>::dimension == 1) { auto& arg_value =
std::any_cast<std::vector<std::shared_ptr<Resource>>&>(any_arg); typeT
converted_value; converted_value.reserve(arg_value.size()); for (auto&
arg_value_item : arg_value) { auto&& resource = std::dynamic_pointer_cast<
```

typename holoscan::type_info<typeT>::derived_type>(arg_value_item); *// Initialize the resource in case the resource created by // Fragment::make_resource<T>() is added to the operator as an argument. // TODO: would like this to be assigned to the same entity as the operator* if (resource) { resource->initialize(); } converted_value.push_back(resource); } param = converted_value; } break; } case ArgElementType::kCustom: { HOLOSCAN_LOG_ERROR("Unable to convert argument type '{}' to parameter type '{}' for '{}'", any_arg.type().name(), typeid(typeT).name(), arg.name()); break; } } break; } case ArgContainerType::kArray: { HOLOSCAN_LOG_ERROR("Unable to handle ArgContainerType::kArray type for '{}'", arg.name()); break; } } } catch (std::bad_any_cast const& e) { HOLOSCAN_LOG_ERROR("Bad any cast exception caught for argument '{}': {}", arg.name(), e.what()); } }); } private: ArgumentSetter() { add_argument_setter<bool>(); add_argument_setter<int8_t>(); add_argument_setter<int16_t>(); add_argument_setter<int32_t>(); add_argument_setter<int64_t>(); add_argument_setter<uint8_t>(); add_argument_setter<uint16_t>(); add_argument_setter<uint32_t>(); add_argument_setter<uint64_t>(); add_argument_setter<float>(); add_argument_setter<double>(); add_argument_setter<std::complex<float>>(); add_argument_setter<std::complex<double>>(); add_argument_setter<std::string>(); add_argument_setter<std::vector<bool>>(); add_argument_setter<std::vector<int8_t>>(); add_argument_setter<std::vector<int16_t>>(); add_argument_setter<std::vector<int32_t>>(); add_argument_setter<std::vector<int64_t>>(); add_argument_setter<std::vector<uint8_t>>(); add_argument_setter<std::vector<uint16_t>>(); add_argument_setter<std::vector<uint32_t>>(); add_argument_setter<std::vector<uint64_t>>(); add_argument_setter<std::vector<float>>(); add_argument_setter<std::vector<double>>(); add_argument_setter<std::vector<std::complex<float>>>(); add_argument_setter<std::vector<std::complex<double>>>(); add_argument_setter<std::vector<std::string>>(); add_argument_setter<std::vector<std::vector<bool>>>(); add_argument_setter<std::vector<std::vector<int8_t>>>(); add_argument_setter<std::vector<std::vector<int16_t>>>();

```
add_argument_setter<std::vector<std::vector<int32_t>>>();
add_argument_setter<std::vector<std::vector<int64_t>>>();
add_argument_setter<std::vector<std::vector<uint8_t>>>();
add_argument_setter<std::vector<std::vector<uint16_t>>>();
add_argument_setter<std::vector<std::vector<uint32_t>>>();
add_argument_setter<std::vector<std::vector<uint64_t>>>();
add_argument_setter<std::vector<std::vector<float>>>();
add_argument_setter<std::vector<std::vector<double>>>();
add_argument_setter<std::vector<std::vector<std::complex<float>>>>();
add_argument_setter<std::vector<std::vector<std::complex<double>>>>();
add_argument_setter<std::vector<std::vector<std::string>>>();
add_argument_setter<YAML::Node>(); add_argument_setter<holoscan::IOSpec*>();
add_argument_setter<std::vector<holoscan::IOSpec*>>();
add_argument_setter<std::shared_ptr<Resource>>();
add_argument_setter<std::vector<std::shared_ptr<Resource>>>();
add_argument_setter<std::shared_ptr<Condition>>();
add_argument_setter<std::vector<std::shared_ptr<Condition>>>(); }
std::unordered_map<std::type_index, SetterFunc> function_map_; }; } // namespace
holoscan #endif/* HOLOSCAN_CORE_ARGUMENT_SETTER_HPP */
```