Program Listing for File arg.hpp

Return to documentation for file (`include/holoscan/core/arg.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_ARG_HPP #define HOLOSCAN_CORE_ARG_HPP #include <yaml-
cpp/yaml.h> #include <any> #include <complex> #include <iostream> #include
<memory> #include <sstream> #include <string> #include <type_traits> #include
<typeindex> #include <typeinfo> #include <unordered_map> #include <utility>
#include <vector> #include "./type_traits.hpp" #include
"holoscan/logger/logger.hpp" // #include "gxf/std/complex.hpp" //
nvidia::gxf::complex64, complex128 namespace holoscan { enum class
ArgElementType { kCustom, kBoolean, kInt8, kUnsigned8, kInt16, kUnsigned16,
kInt32, kUnsigned32, kInt64, kUnsigned64, kFloat32, kFloat64, kComplex64,
kComplex128, kString, kHandle, kYAMLNode, kIOSpec, kCondition, kResource, };
enum class ArgContainerType : uint8_t { kNative, kVector, kArray, }; class ArgType {
public: ArgType() = default; ArgType(ArgElementType element_type,
ArgContainerType container_type, int32_t dimension = 0) :
element_type_(element_type), container_type_(container_type),
dimension_(dimension) {} static ArgElementType get_element_type(std::type_index
index) { auto& elem_type_map = element_type_map_; if (elem_type_map.find(index)
== elem_type_map.end()) { return ArgElementType::kCustom; } const auto&
elem_type = elem_type_map[index]; return elem_type; } template <typename
typeT> static ArgType create() { if constexpr
(holoscan::is_scalar_v<std::decay_t<typeT>>) { auto index =
std::type_index(typeid(typename
holoscan::type_info<std::decay_t<typeT>>::element_type)); return
ArgType(get_element_type(index), ArgContainerType::kNative); } else if constexpr
(holoscan::is_vector_v<std::decay_t<typeT>>) { auto elem_index =
std::type_index(typeid(typename holoscan::type_info<typeT>::element_type)); return
ArgType( get_element_type(elem_index), ArgContainerType::kVector,
holoscan::dimension_of_v<typeT>); } else if constexpr
(holoscan::is_array_v<std::decay_t<typeT>>) { auto elem_index =
std::type_index(typeid(typename holoscan::type_info<typeT>::element_type)); return
ArgType( get_element_type(elem_index), ArgContainerType::kArray,
holoscan::dimension_of_v<typeT>); } else { HOLOSCAN_LOG_ERROR("No element
type for '{}' exists", typeid(std::decay_t<typeT>).name()); return
ArgType(ArgElementType::kCustom, ArgContainerType::kNative); } } ArgElementType
element_type() const { return element_type_; } ArgContainerType container_type()
const { return container_type_; } int32_t dimension() const { return dimension_; }
std::string to_string() const; private: template <class typeT> inline static
std::pair<const std::type_index, ArgElementType> to_element_type_pair(
ArgElementType element_type) { return
{std::type_index(typeid(std::decay_t<typeT>)), element_type}; } inline static
std::unordered_map<std::type_index, ArgElementType> element_type_map_{
to_element_type_pair<bool>(ArgElementType::kBoolean),
to_element_type_pair<int8_t>(ArgElementType::kInt8),
to_element_type_pair<uint8_t>(ArgElementType::kUnsigned8),
to_element_type_pair<int16_t>(ArgElementType::kInt16),
to_element_type_pair<uint16_t>(ArgElementType::kUnsigned16),
to_element_type_pair<int32_t>(ArgElementType::kInt32),
to_element_type_pair<uint32_t>(ArgElementType::kUnsigned32),
to_element_type_pair<int64_t>(ArgElementType::kInt64),
to_element_type_pair<uint64_t>(ArgElementType::kUnsigned64),
to_element_type_pair<float>(ArgElementType::kFloat32),
to_element_type_pair<double>(ArgElementType::kFloat64), //
to_element_type_pair<nvidia::gxf::complex64>(ArgElementType::kComplex64), //
to_element_type_pair<nvidia::gxf::complex128>(ArgElementType::kComplex128),
to_element_type_pair<std::complex<float>>(ArgElementType::kComplex64),
to_element_type_pair<std::complex<double>>(ArgElementType::kComplex128),
to_element_type_pair<std::string>(ArgElementType::kString),
to_element_type_pair<std::any>(ArgElementType::kHandle),
to_element_type_pair<YAML::Node>(ArgElementType::kYAMLNode),
to_element_type_pair<holoscan::IOSpec*>(ArgElementType::kIOSpec),
to_element_type_pair<std::shared_ptr<Condition>>(ArgElementType::kCondition),
to_element_type_pair<std::shared_ptr<Resource>>(ArgElementType::kResource), };
inline static const std::unordered_map<ArgElementType, const char*>
element_type_name_map_{ {ArgElementType::kCustom, "CustomType"},
{ArgElementType::kBoolean, "bool"}, {ArgElementType::kInt8, "int8_t"},
{ArgElementType::kUnsigned8, "uint8_t"}, {ArgElementType::kInt16, "int16_t"},
{ArgElementType::kUnsigned16, "uint16_t"}, {ArgElementType::kInt32, "int32_t"},
{ArgElementType::kUnsigned32, "uint32_t"}, {ArgElementType::kInt64, "int64_t"},
{ArgElementType::kUnsigned64, "uint64_t"}, {ArgElementType::kFloat32, "float"},
{ArgElementType::kFloat64, "double"}, {ArgElementType::kComplex64,
"std::complex<float>"}, {ArgElementType::kComplex128, "std::complex<double>"}, //
{ArgElementType::kComplex64, "nvidia::gxf::complex64"}, //
{ArgElementType::kComplex128, "nvidia::gxf::complex128"}, {ArgElementType::kString,
"std::string"}, {ArgElementType::kHandle, "std::any"}, {ArgElementType::kYAMLNode,
"YAML::Node"}, {ArgElementType::kIOSpec, "holoscan::IOSpec*"},
{ArgElementType::kCondition, "std::shared_ptr<Condition>"},
{ArgElementType::kResource, "std::shared_ptr<Resource>"}, }; ArgElementType
element_type_ = ArgElementType::kCustom; ArgContainerType container_type_ =
ArgContainerType::kNative; int32_t dimension_ = 0; }; class Arg { public: explicit
Arg(const std::string& name) : name_(name) {} ~Arg() = default; template <typename
ArgT> Arg(const std::string& name, const ArgT& value) { name_ = name;
set_value_<ArgT>(value); } template <typename ArgT> Arg(const std::string& name,
ArgT&& value) { name_ = name; set_value_<ArgT>(std::forward<ArgT>(value)); }
template <typename ArgT, typename = std::enable_if_t<!std::is_same_v<Arg,
std::decay_t<ArgT>>>> Arg& operator=(const ArgT& value) { set_value_<ArgT>
(value); return *this; } template <typename ArgT, typename =
std::enable_if_t<!std::is_same_v<Arg, std::decay_t<ArgT>>>> Arg&& operator=
(ArgT&& value) { set_value_<ArgT>(std::forward<ArgT>(value)); return
std::move(*this); } const std::string& name() const { return name_; } const ArgType&
arg_type() const { return arg_type_; } bool has_value() const { return
value_.has_value(); } std::any& value() { return value_; } YAML::Node to_yaml_node()
const; YAML::Node value_to_yaml_node() const; std::string description() const;
private: std::string name_; ArgType arg_type_; std::any value_; template <typename
ArgT> void set_value_(const ArgT& value) { arg_type_ = ArgType::create<ArgT>();
HOLOSCAN_LOG_TRACE( "Arg::set_value(const ArgT& value)({}) parameter: {},
element_type: {}, container_type: {}", typeid(ArgT).name(), name_, static_cast<int>
(arg_type_.element_type()), static_cast<int>(arg_type_.container_type())); if constexpr
```

```
(is_one_of_v<typename holoscan::type_info<ArgT>::element_type,
std::shared_ptr<Resource>, std::shared_ptr<Condition>>) { if constexpr
(is_scalar_v<ArgT>) { value_ = std::dynamic_pointer_cast< base_type_t<typename
holoscan::type_info<ArgT>::derived_type>>(value); } else if constexpr
(is_vector_v<ArgT> && holoscan::type_info<ArgT>::dimension == 1) {
std::vector<typename holoscan::type_info<ArgT>::element_type> components;
components.reserve(value.size()); for (auto& value_item : value) { auto component =
std::dynamic_pointer_cast< base_type_t<typename
holoscan::type_info<ArgT>::derived_type>>(value_item);
components.push_back(component); } value_ = components; } } else { value_ =
value; } } template <typename ArgT> void set_value_(ArgT&& value) { arg_type_ =
ArgType::create<ArgT>(); HOLOSCAN_LOG_TRACE( "Arg::set_value(ArgT&& value)({})
parameter: {}, element_type: {}, container_type: {}, " "ArgT: {}", typeid(ArgT).name(),
name_, static_cast<int>(arg_type_.element_type()), static_cast<int>
(arg_type_.container_type()), typeid(ArgT).name()); if constexpr
(is_one_of_v<typename holoscan::type_info<ArgT>::element_type,
std::shared_ptr<Resource>, std::shared_ptr<Condition>>) { if constexpr
(is_scalar_v<ArgT>) { value_ = std::move(std::dynamic_pointer_cast<
base_type_t<typename holoscan::type_info<ArgT>::derived_type>>(value)); } else if
constexpr (is_vector_v<ArgT> && holoscan::type_info<ArgT>::dimension == 1) {
std::vector<typename holoscan::type_info<ArgT>::element_type> components;
components.reserve(value.size()); for (auto& value_item : value) { auto component =
std::dynamic_pointer_cast< base_type_t<typename
holoscan::type_info<ArgT>::derived_type>>(value_item);
components.push_back(std::move(component)); } value_ = std::move(components);
} } else { value_ = std::forward<ArgT>(value); } } }; class ArgList { public: ArgList() =
default; explicit ArgList(std::initializer_list<Arg> args) { for (auto& arg : args) {
args_.push_back(arg); } } ~ArgList() = default; size_t size() const { return args_.size(); }
std::vector<Arg>::iterator begin() { return args_.begin(); } std::vector<Arg>::iterator
end() { return args_.end(); } std::vector<Arg>::const_iterator begin() const { return
args_.begin(); } std::vector<Arg>::const_iterator end() const { return args_.end(); }
void clear() { args_.clear(); } std::vector<Arg>& args() { return args_; } template
<typename typeT> typeT as() { if (args_.empty()) { HOLOSCAN_LOG_ERROR("No item
available in the argument list."); return typeT(); } // Only take the first item in the
argument list. auto& argument = args_[0]; if (argument.arg_type().element_type() !=
ArgElementType::kYAMLNode) { HOLOSCAN_LOG_ERROR("The type of the argument
'{}' should be kYAMLNode."); return typeT(); } auto node =
std::any_cast<YAML::Node>(argument.value()); try { return node.as<typeT>(); } catch
(...) { std::stringstream ss; ss << node; HOLOSCAN_LOG_ERROR("Unable to parse
YAML node: '{}'", ss.str()); return typeT(); } } void add(const Arg& arg) {
args_.emplace_back(arg); } void add(Arg&& arg) {
args_.emplace_back(std::move(arg)); } void add(const ArgList& arg) {
args_.reserve(args_.size() + arg.size()); args_.insert(args_.end(), arg.begin(), arg.end());
} void add(ArgList&& arg) { args_.reserve(args_.size() + arg.size()); args_.insert(
args_.end(), std::make_move_iterator(arg.begin()),
std::make_move_iterator(arg.end())); arg.clear(); } const std::string& name() const {
return name_; } YAML::Node to_yaml_node() const; std::string description() const;
private: std::string name_{"arglist"}; std::vector<Arg> args_; }; } // namespace
holoscan #endif/* HOLOSCAN_CORE_ARG_HPP */
```