dhorn service\_container

# Introduction

The dhorn::service\_container class is meant to provide an easy way of publishing and looking up needed services. The service container stores one instance per type.

# Classes/Structures/Types/Enumerations/etc.

The service container is split between the main dhorn::service\_container class, and different traits structs.

## dhorn::service\_conatiner Class

The service container class essentially consists of three classes of functions: inserting, retrieving, and removing. Inserting can be done either by passing in a pointer to the object, or by emplacing a new object. The type is deduced in the former case, but must be explicitly specified in the latter. Finding and removing are both performed by explicitly specifying the type as well. The general interface is defined below:

class service\_container final

{

public:

/\*

\* Constructor(s)/Destructor

\*/

service\_container(void);

service\_container(\_Inout\_ service\_container &&other);

// Cannot copy

service\_container(\_In\_ const service\_container &) = delete;

/\*unspecified\*/ operator=(\_In\_ const service\_container &) = delete;

/\*

\* Operators

\*/

service\_container &operator=(\_Inout\_ service\_container &&other);

/\*

\* Insertion

\*/

template <typename Ty>

std::shared\_ptr<Ty> insert(\_In\_ Ty \*ptr);

template <typename Ty, typename Deleter>

std::shared\_ptr<Ty> insert(\_In\_ Ty \*ptr, \_In\_ const Deleter &del);

template <typename Ty, typename Deleter, typename Alloc>

std::shared\_ptr<Ty> insert(\_In\_ Ty \*ptr, \_In\_ const Deleter &del, \_In\_ const Alloc &alloc);

template <typename Ty>

std::shared\_ptr<Ty> insert(\_In\_ const std::shared\_ptr<Ty> &ptr);

template <typename Ty, typename... Args>

std::shared\_ptr<Ty> emplace(\_Inout\_ Args&&... args);

template <typename Ty, typename Alloc, typename... Args>

std::shared\_ptr<Ty> allocate(\_In\_ const Alloc &alloc, \_Inout\_ Args&&... args);

/\*

\* Retrieval

\*/

template <typename Ty>

const std::shared\_ptr<Ty> &find(void) const;

/\*

\* Removal

\*/

template <typename Ty>

std::shared\_ptr<Ty> remove(void);

/\*

\* Other public functions

\*/

size\_t size(void) const;

bool empty(void) const;

void swap(\_Inout\_ service\_container &other);

};

The functionality of all functions should be obvious. Restrictions and other important implementation details are described in other sections.

## Exception Types

There are three exception types introduced:

class service\_exception :

public std::exception

{

public:

service\_exception(\_In\_ const std::string &exceptionType, \_In\_ const std::string &typeName);

virtual const char \*what(void) const;

};

class service\_published :

public service\_exception

{

public:

service\_published(\_In\_ const std::string &serviceName);

};

class service\_not\_published :

public service\_exception

{

public:

service\_not\_published(\_In\_ const std::string &serviceName);

};

The dhorn::service\_exception is never explicitly thrown; it just serves at the base class for the other two exception types. It does, however, supply the useful override of the what virtual function which returns a string containing the exception type and the type associated with the exception in the format “<<exception type>> : <<service type>>.” The insertion family of functions will only throw dhorn::service\_published. This occurs when a service with the same identifier has already been published. Note that this does NOT imply that a service with the same **type** has been published. Whether or not dhorn::service\_published gets thrown in situations where two different types have the same identifier is implementation defined. Both the find and remove family of functions will only ever throw dhorn::service\_not\_published. This occurs when attempting to find or remove a type that **was not successfully** published to the service container. Note that even in the case where you attempt to find or remove a type with the same identifier as another, different published type, this exception is still required to get thrown. Behavior is only implementation defined when publishing two types with the same identifier.

## dhorn::service\_type\_traits Structure

Type identifiers are retrieved using that type’s overload of the dhorn::service\_type\_traits structure. An example overload for a hypothetical type Foo is shown below:

namespace dhorn

{

template <>

struct service\_type\_traits<Foo>

{

/\*constexpr\*/ inline static const dhorn::uuid &id(void)

{

static const dhorn::uuid id{ 0x7c026cbf, 0x3df9, 0x4a70,

{ 0xae, 0x26, 0xef, 0x30, 0x11, 0xf1, 0xfe, 0xe2 } };

return id;

}

};

}

Note that the inclusion of constexpr is optional. I.e. one could implement the id function to return dhorn::random\_uuid. Note that if that is the case, you obviously must save the uuid returned by the dhorn::random\_uuid function and ensure that the same value gets returned for all subsequent calls to the id function if you wish for correct behavior.