Initialization and CommunicatorHolder in C++11

Previous Next

On this page:

Initializing the Ice Run Time with `Ice::initialize`

Every Ice-based application needs to initialize the Ice run time, and this initialization returns an `Ice::Communicator` object.

A `Communicator` is a local C++ object that represents an instance of the Ice run time. Most Ice-based applications create and use a single `Communicator` object, although it is possible and occasionally desirable to have multiple `Communicator` objects in the same application or program.

You initialize the Ice run time by calling the C++ function `Ice::initialize`, for example:

C++

```
int
main(int argc, char* argv[])
{
    std::shared_ptr<Ice::Communicator> communicator = Ice::initialize(argc, argv);
    // ...
}
```

`initialize` accepts a C++ reference to `argc` and an argument vector `argv`. The function scans the argument vector for any command-line options that are relevant to the Ice run time; any such options are removed from the argument vector so, when `initialize` returns, the only options and arguments remaining are those that concern your application. If anything goes wrong during initialization, `initialize` throws an exception.

> `Ice::initialize` has additional overloads to permit other information to be passed to the Ice run time.

`initialize` is a low-level function, as you need to explicitly call `destroy` on the returned `Communicator` object when you're done with Ice, typically just before returning from `main`. The `destroy` member function is responsible for finalizing the Ice run time. In particular, in a server, `destroy` waits for any operation implementations that are still executing to complete. In addition, `destroy` ensures that any outstanding threads are joined with and reclaims a number of operating system resources, such as file descriptors and memory.

The general shape of the `main` function of an Ice-based application is therefore:

C++

```
#include <Ice/Ice.h>

int
main(int argc, char* argv[])
{
    int status = 0;
    try
    {
        // communicator is std::shared_ptr<Ice::Communicator>
        auto communicator = Ice::initialize(argc, argv);

        try
        {
            ... application code ...

            communicator->destroy(); // destroy is noexcept
        }
        catch(const std::exception&)
        {
            ...
            // make sure communicator is destroyed if an exception is thrown
            communicator->destroy();
            throw;
        }
    }
    catch(const std::exception& e)
    {
        cerr << e.what() << endl;
        status = 1;
    }
    return status;
}
```

This code is a little bit clunky, as we need to make sure the communicator gets destroyed in all paths, including when an exception is thrown. As a result, most of the time, you should not call `initialize` directly: you should use instead a helper class that calls `initialize` and ensures the resulting communicator gets eventually destroyed.

Back to Top ^

`Ice::CommunicatorHolder` RAII Helper Class

A `CommunicatorHolder` is a small RAII helper class that creates a `Communicator` in its constructor (by calling `initialize`) and destroys this communicator in its destructor.

With a `CommunicatorHolder`, our typical `main` function becomes much simpler:

C++

```
#include <Ice/Ice.h>

int
main(int argc, char* argv[])
{
    int status = 0;
    try
    {
        Ice::CommunicatorHolder ich(argc, argv); // Calls Ice::initialize

        ... application code ...

        // CommunicatorHolder's destructor calls destroy on the communicator
        // whether or not an exception is thrown
    }
    catch(const std::exception& e)
    {
        cerr << e.what() << endl;
        status = 1;
    }
    return status;
}
```

`Ice::CommunicatorHolder` is defined as follows:

C++

```
namespace Ice
{
    class CommunicatorHolder
    {
    public:

        CommunicatorHolder();
        template<class... T> explicit CommunicatorHolder(T&&... params);

        explicit CommunicatorHolder(std::shared_ptr<Communicator>);
        CommunictorHolder& operator=(std::shared_ptr<Communicator>);

        CommunicatorHolder(const CommunicatorHolder&) = delete;
        CommunicatorHolder(CommunicatorHolder&&) = default;
        CommunicatorHolder& operator=(CommunicatorHolder&&);

        ~CommunicatorHolder();

        explicit operator bool() const;
        const std::shared_ptr<Communicator>& communicator() const;
        const std::shared_ptr<Communicator>& operator->() const;
        std::shared_ptr<Communicator> release();
        ...
    };
}
```

Let's examine each of these functions:

- `CommunicatorHolder()`
  This default constructor creates an empty holder (holds no `Communicator`).
- `template<class... T> explicit CommunicatorHolder(T&&... params)`
  These constructors call `initialize` with the provided parameters; the new `CommunicatorHolder` then holds the resulting `Communicator` in a private data member (not shown).
- `explicit CommunicatorHolder(std::shared_ptr<Communicator>)`
  This constructor adopts the given communicator: the `CommunicatorHolder` becomes responsible for calling `destroy` on it.

- `CommunicatorHolder& operator=(std::shared_ptr<Communicator>)`
  This assignment operator destroys the communicator held by this `CommunicatorHolder`, then adopts the provided communicator.

- `CommunicatorHolder& operator=(CommunicatorHolder&&)`
  This assignment operator destroys the communicator held by this `CommunicatorHolder`, then adopts the other `CommunicatorHolder's` communicator.

- `~CommunicatorHolder()`
  The destructor calls `destroy` on the communicator held by this `CommunicatorHolder`.

- `explicit operator bool() const`
  Returns `true` when this `CommunicatorHolder` holds a `Communicator`, and `false` otherwise.

- `const std::shared_ptr<Communicator>& communicator() const`

  This function gives read-only access to the communicator held by this `CommunicatorHolder`.

- `const std::shared_ptr<Communicator>& operator->() const`
  This arrow operator allows you to use a `CommunicatorHolder` just like a `Communicator` object. For example:

  C++

  ```
  Ice::CommunicatorHolder ich(argc, argv);
  auto base = ich->stringToProxy("SimplePrinter:default -p 10000");
  ```

  is equivalent to:

  C++

  ```
  Ice::CommunicatorHolder ich(argc, argv);
  auto base = ich->communicator()->stringToProxy("SimplePrinter:default -p 10000");
  ```

- `std::shared_ptr<Communicator> release()`
  This function returns the communicator held by `CommunicatorHolder` to the caller, and the caller becomes responsible for destroying this communicator. `CommunicatorHolder` no longer holds a communicator after this call.

The default constructor of `CommunicatorHolder` does nothing:

C++

```
Ice::CommunicatorHolder ich; // does not create a Communicator, ich.communicator() returns a null
shared_ptr
```

If you want to create a `CommunicatorHolder` that holds a `Communicator` created by `initialize` with no args, you can write:

C++

```
Ice::CommunicatorHolder ich = Ice::initialize();
```

Back to Top ^

See Also

- Communicators
- Communicator Initialization
- Application Helper Class