Service Helper Class

On this page:

- The Ice::Service C++ Class
- Ice::Service Member Functions
- Unix Daemons
- Windows Services
- Ice::Service Logging Considerations

The `Ice::Service` C++ Class

Ice provides `Ice::Service`, a singleton class that is comparable to `Ice::Application` but also encapsulates the low-level, platform-specific initialization and shutdown procedures common to Unix daemons and Windows services. `Ice::Service` is currently available only in C++.

The `Ice::Service` class is declared as follows:

C++11

```
namespace Ice
{
    class Service
    {
    public:

        Service();
        virtual ~Service();

        virtual bool shutdown();
        virtual void interrupt();

        int main(int argc, const char* const argv[], const InitializationData& initData =
InitializationData(), int version = ICE_INT_VERSION);
#ifdef _WIN32
        int main(int argv, const wchar_t* const argv[], const InitializationData& initData =
InitializationData(), int version = ICE_INT_VERSION);
#endif
        int main(const StringSeq& args, const InitializationData& initData = InitializationData(), int
version = ICE_INT_VERSION);

        std::shared_ptr<Communicator> communicator() const;
        static Service* instance();

        bool service() const;
        std::string name() const;
        bool checkSystem() const;

        int run(int argc, const char* const argv[], const InitializationData& initData =
InitializationData(), int version= ICE_INT_VERSION);
#ifdef _WIN32
        int run(int argc, const wchar_t* const argv[], const InitializationData& initData =
InitializationData(), int version = ICE_INT_VERSION);
#endif

#ifdef _WIN32
        void configureService(const std::string& name);
#else
        void configureDaemon(bool chdir, bool close, const std::string& pidFile);
#endif

        virtual void handleInterrupt(int);

    protected:

        virtual bool start(int argc, char* argv[], int& status) = 0;
        virtual void waitForShutdown();
        virtual bool stop();

        virtual std::shared_ptr<Communicator> initializeCommunicator(int& argc, char* argv[], const
InitializationData& initData, int version);

        virtual void syserror(const std::string& msg);
        virtual void error(const std::string& msg);
        virtual void warning(const std::string& msg);
        virtual void trace(const std::string& msg);
        virtual void print(const std::string& msg);

        void enableInterrupt();
        void disableInterrupt();

        ...
    };
}
```

C++98

```
namespace Ice
{
    class Service
    {
    public:

        Service();
        virtual ~Service();

        virtual bool shutdown();
        virtual void interrupt();

        int main(int argc, const char* const argv[], const InitializationData& initData =
InitializationData(), int version = ICE_INT_VERSION);
#ifdef _WIN32
        int main(int argv, const wchar_t* const argv[], const InitializationData& initData =
InitializationData(), int version = ICE_INT_VERSION);
#endif
        int main(const StringSeq& args, const InitializationData& initData = InitializationData(), int
version = ICE_INT_VERSION);

        CommunicatorPtr communicator() const;
        static Service* instance();

        bool service() const;
        std::string name() const;
        bool checkSystem() const;

        int run(int argc, const char* const argv[], const InitializationData& initData =
InitializationData(), int version= ICE_INT_VERSION);
#ifdef _WIN32
        int run(int argc, const wchar_t* const argv[], const InitializationData& initData =
InitializationData(), int version = ICE_INT_VERSION);
#endif

#ifdef _WIN32
        void configureService(const std::string& name);
#else
        void configureDaemon(bool chdir, bool close, const std::string& pidFile);
#endif

        virtual void handleInterrupt(int);

    protected:

        virtual bool start(int argc, char* argv[], int& status) = 0;
        virtual void waitForShutdown();
        virtual bool stop();

        virtual CommunicatorPtr initializeCommunicator(int& argc, char* argv[], const InitializationData&
initData, int version);

        virtual void syserror(const std::string& msg);
        virtual void error(const std::string& msg);
        virtual void warning(const std::string& msg);
        virtual void trace(const std::string& msg);
        virtual void print(const std::string& msg);

        void enableInterrupt();
        void disableInterrupt();
        ...
    };
}
```

At a minimum, an Ice application that uses the `Ice::Service` class must define a subclass and override the `start` member function, which is where the service must perform its startup activities, such as processing command-line arguments, creating an object adapter, and registering servants. The application's `main` function must instantiate the subclass and typically invokes its `main` member function, passing the program's argument vector as parameters. The example below illustrates a minimal `Ice::Service` subclass:

C++11

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

class MyService : public Ice::Service
{
protected:
    virtual bool start(int, char*[], int&) override;
private:
    std::shared_ptr<Ice::ObjectAdapter> _adapter;
};

bool
MyService::start(int argc, char* argv[], int& status)
{
    _adapter = communicator()->createObjectAdapter("MyAdapter");
    _adapter->addWithUUID(std::make_shared<MyServantI>());
    _adapter->activate();
    status = EXIT_SUCCESS;
    return true;
}

int
main(int argc, char* argv[])
{
    MyService svc;
    return svc.main(argc, argv);
}
```

C++98

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

class MyService : public Ice::Service
{
protected:
    virtual bool start(int, char*[], int&);
private:
    Ice::ObjectAdapterPtr _adapter;
};

bool
MyService::start(int argc, char* argv[], int& status)
{
    _adapter = communicator()->createObjectAdapter("MyAdapter");
    _adapter->addWithUUID(new MyServantI);
    _adapter->activate();
    status = EXIT_SUCCESS;
    return true;
}

int
main(int argc, char* argv[])
{
    MyService svc;
    return svc.main(argc, argv);
}
```

The `Service::main` member function performs the following sequence of tasks:

1. Scans a copy of the argument vector for reserved options that indicate whether the program should run as a system service and removes these options from this copy of the argument vector. Additional reserved options are supported for administrative tasks.
2. Configures the program for running as a system service (if necessary) by invoking `configureService` or `configureDaemon`, as appropriate for the platform.
3. Invokes the `run` member function with the filtered argument vector and returns its result.

Note that, as for `Application::main`, `Service::main` is overloaded to accept a string sequence instead of an `argc`/`argv` pair. This is useful if you need to parse application-specific property settings on the command line.

For maximum portability, we strongly recommend that all initialization tasks be performed in the `start` member function and not in the global `main` function. For example, allocating resources in `main` can cause program instability for Unix daemons.

The `Service::run` member function executes the service in the steps shown below:

1. Makes a copy of the provided argument vector.
2. Installs a signal handler.
3. Invokes the `initializeCommunicator` member function to obtain a communicator. The communicator instance can be accessed using the `communicator` member function.
4. Invokes the `start` member function. If `start` returns `false` to indicate failure, `run` destroys the communicator and returns immediately using the exit status provided in `status`.
5. Invokes the `waitForShutdown` member function, which should block until `shutdown` is invoked.
6. Invokes the `stop` member function. If `stop` returns `true`, `run` considers the application to have terminated successfully.
7. Destroys the communicator.
8. Gracefully terminates the system service (if necessary).

If an unhandled exception is caught by `Service::run`, a descriptive message is logged, the communicator is destroyed and the service is terminated.

`Ice::Service` Member Functions

The virtual member functions in `Ice::Service` represent the points at which a subclass can intercept the service activities. All of the virtual member functions (except `start`) have default implementations.

- `void handleInterrupt(int sig)`
  Invoked by the signal handler when it catches a signal. The default implementation ignores the signal if it represents a logoff event and the `Ice.Nohup` property is set to a value larger than zero, otherwise it invokes the `interrupt` member function.

- `std::shared_ptr<Ice::Communicator> initializeCommunicator(int& argc, char* argv[], const InitializationData& initData, int version)` (C++11)
  `Ice::CommunicatorPtr initializeCommunicator(int& argc, char* argv[], const InitializationData& initData, int version)` (C++98)
  Initializes a communicator. The default implementation invokes `Ice::initialize` and passes the given arguments.

- `void interrupt()`
  Invoked by the signal handler to indicate a signal was received. The default implementation invokes the `shutdown` member function.

- `bool shutdown()`
  Causes the service to begin the shutdown process. The default implementation invokes `shutdown` on the communicator. The subclass must return `true` if shutdown was started successfully, and `false` otherwise.

- `bool start(int argc, char* argv[], int& status)`
  Allows the subclass to perform its startup activities, such as scanning the provided argument vector for recognized command-line options, creating an object adapter, and registering servants. The subclass must return `true` if startup was successful, and `false` otherwise. The subclass can set an exit status via the `status` parameter. This status is returned by `main`.

- `bool stop()`
  Allows the subclass to clean up prior to termination. The default implementation does nothing but return `true`. The subclass must return `true` if the service has stopped successfully, and `false` otherwise.

- `void syserror(const std::string& msg)`
  `void error(const std::string& msg)`
  `void warning(const std::string& msg)`
  `void trace(const std::string& msg)`
  `void print(const std::string& msg)`
  Convenience functions for logging messages to the communicator's logger. The `syserror` member function includes a description of the system's current error code. You can also log messages to these functions using utility classes similar to the C++ Logger Utility Classes: these classes are `ServiceSysError`, `ServiceError`, `ServiceWarning`, `ServiceTrace` and `ServicePrint`, all nested in the Service class.

- `void waitForShutdown()`
  Waits indefinitely for the service to shut down. The default implementation invokes `waitForShutdown` on the communicator.

The non-virtual member functions shown in the class definition are described below:

- `bool checkSystem() const`
  Returns true if the operating system supports Windows services or Unix daemons.

- `std::shared_ptr<Ice::Communicator> communicator() const` (C++11)
  `Ice::CommunicatorPtr communicator() const` (C++98)
  Returns the communicator used by the service, as created by `initializeCommunicator`.

- `void configureDaemon(bool chdir, bool close, const std::string& pidFile)`
  Configures the program to run as a Unix daemon. The `chdir` parameter determines whether the daemon changes its working directory to the root directory. The `close` parameter determines whether the daemon closes unnecessary file descriptors (i.e., stdin, stdout, etc.). If a non-empty string is provided in the `pidFile` parameter, the daemon writes its process ID to the given file.

- `void configureService(const std::string& name)`
  Configures the program to run as a Windows service with the given name.

- `void disableInterrupt()`
  Disables the signal handling behavior in `Ice::Service`. When disabled, signals are ignored.

- `void enableInterrupt()`
  Enables the signal handling behavior in `Ice::Service`. When enabled, the occurrence of a signal causes the `handleInterrupt` member function to be invoked.

- `static Service* instance()`
  Returns the singleton `Ice::Service` instance.

- `int main(int argv, const char* const argv[], const InitializationData& initData = InitializationData(), int version = ICE_INT_VERSION)`
  `int main(int argc, const wchar_t* const argv[], const InitializationData& initData = InitializationData(), int version = ICE_INT_VERSION)`
  `int main(const Ice::StringSeq& args, const InitializationData& initData = InitializationData(), int version = ICE_INT_VERSION);`
  The primary entry point of the `Ice::Service` class. The tasks performed by this function are described earlier in this section. The function returns `EXIT_SUCCESS` for success, `EXIT_FAILURE` for failure. For Windows, this function is overloaded to allow you to pass a `wchar_t` ar gument vector.

- `std::string name() const`
  Returns the name of the service. If the program is running as a Windows service, the return value is the Windows service name, otherwise it returns the value of `argv[0]`.

- `int run(int argc, const char* const argv[], const InitializationData& initData = InitializationData(), int version = ICE_INT_VERSION)`
  Alternative entry point for applications that prefer a different style of service configuration. The program must invoke `configureService` (Windows) or `configureDaemon` (Unix) in order to run as a service. The tasks performed by this function were described earlier. The function normally returns `EXIT_SUCCESS` or `EXIT_FAILURE`, but the `start` method can also supply a different value via its `status` argum ent.

- `bool service() const`
  Returns true if the program is running as a Windows service or Unix daemon, or false otherwise.

Unix Daemons

On Linux and macOS, passing `--daemon` causes your program to run as a daemon. When this option is present, `Ice::Service` performs the following additional actions:

- Creates a background child process in which `Service::main` performs its tasks. The foreground process does not terminate until the child process has successfully invoked the `start` member function. This behavior avoids the uncertainty often associated with starting a daemon from a shell script by ensuring that the command invocation does not complete until the daemon is ready to receive requests.
- Changes the current working directory of the child process to the root directory, unless `--nochdir` is specified.
- Closes all file descriptors, unless `--noclose` is specified. The standard input (stdin) channel is closed and reopened to `/dev/null`. Likewise, the standard output (stdout) and standard error (stderr) channels are also closed and reopened to `/dev/null` unless `Ice.StdOut` or `Ice.StdErr` are defined, respectively, in which case those channels use the designated log files.

> The file descriptors are not closed until after the communicator is initialized, meaning standard input, standard output, and standard error are available for use during this time. For example, the IceSSL plug-in may need to prompt for a passphrase on standard input, or Ice may print the child's process id on standard output if the property `Ice.PrintProcessId` is set.

The following additional command-line options can be specified in conjunction with the `--daemon` option:

- `--pidfile` *`FILE`*
  This option writes the process ID of the service into the specified *`FILE`*.

- `--noclose`
  Prevents `Ice::Service` from closing unnecessary file descriptors. This can be useful during debugging and diagnosis because it provides access to the output from the daemon's standard output and standard error.

- `--nochdir`
  Prevents `Ice::Service` from changing the current working directory.

All of these options are removed from the argument vector that is passed to the `start` member function.

We strongly recommend that you perform all initialization tasks in your service's `start` member function, and not in the global `main` function. This is especially important for process-specific resources such as file descriptors, threads, and mutexes, which can be affected by the use of the `fork` system call in `Ice::Service`. For example, any files opened in `main` are automatically closed by `Ice::Service` and therefore unusable in your service, unless the daemon is started with the `--noclose` option.

Windows Services

On Windows, `Ice::Service` recognizes the following command-line options:

- `--service` *`NAME`*
  Run as a Windows service named *`NAME`*, which must already be installed. This option is removed from the argument vector that is passed to the `start` member function.

Installing and configuring a Windows service is outside the scope of the `Ice::Service` class. Ice includes a utility for installing its services which you can use as a model for your own applications.

The `Ice::Service` class supports the Windows service control codes `SERVICE_CONTROL_INTERROGATE` and `SERVICE_CONTROL_STOP`. Upon receipt of `SERVICE_CONTROL_STOP`, `Ice::Service` invokes the `shutdown` member function.

`Ice::Service` Logging Considerations

A service that uses a custom logger has several ways of configuring it:

- as a process-wide logger,
- in the `InitializationData` argument that is passed to `main`,
- by overriding the `initializeCommunicator` member function.

On Windows, `Ice::Service` installs its own logger that uses the Windows `Application` event log if no custom logger is defined. The source name for the event log is the service's name unless a different value is specified using the property `Ice.EventLog.Source`.

On Unix, the default Ice logger (which logs to the standard error output) is used when no other logger is configured. For daemons, this is not appropriate because the output will be lost. To change this, you can either implement a custom logger or set the `Ice.UseSyslog` property, which selects a logger implementation that logs to the `syslog` facility. Alternatively, you can set the `Ice.LogFile` property to write log messages to a file.

Note that `Ice::Service` may encounter errors before the communicator is initialized. In this situation, `Ice::Service` uses its default logger unless a process-wide logger is configured. Therefore, even if a failing service is configured to use a different logger implementation, you may find useful diagnostic information in the `Application` event log (on Windows) or sent to standard error (on Linux and macOS).

See Also

- Hello World Application
- Properties and Configuration
- Communicator Initialization
- Logger Facility
- Windows Services