The Server-Side main Function in C++

On this page:

A Basic `main` Function in C++

The main entry point to the Ice run time is represented by the local Slice interface `Ice::Communicator`. As for the client side, you must initialize the Ice run time by calling `Ice::initialize` before you can do anything else in your server. `Ice::initialize` returns a smart pointer to an instance of an `Ice::Communicator`:

C++

```
int
main(int argc, char* argv[])
{
    Ice::CommunicatorPtr ic = Ice::initialize(argc, argv);
    // ...
}
```

`Ice::initialize` accepts a C++ reference to `argc` and `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 `Ice::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.

Before leaving your `main` function, you must call `Communicator::destroy`. The `destroy` operation is responsible for finalizing the Ice run time. In particular, `destroy` waits for any operation implementations that are still executing in the server 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. Never allow your `main` function to terminate without calling `destroy` first; doing so has undefined behavior.

The general shape of our server-side `main` function is therefore as follows:

C++

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

int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);

        // Server code here...

    } catch (const Ice::Exception& e) {
        cerr << e << endl;
        status = 1;
    } catch (const std::string& msg) {
        cerr << msg << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic) {
        try {
            ic->destroy();
        } catch (const Ice::Exception& e) {
            cerr << e << endl;
            status = 1;
        }
    }
    return status;
}
```

Note that the code places the call to `Ice::initialize` in to a `try` block and takes care to return the correct exit status to the operating system. Also note that an attempt to destroy the communicator is made only if the initialization succeeded.

The `catch` handlers for `const std::string &` and `const char *` are in place as a convenience feature: if we encounter a fatal error condition anywhere in the server code, we can simply throw a string or a string literal containing an error message; this causes the stack to be unwound back to `main`, at which point the error message is printed and, after destroying the communicator, `main` terminates with non-zero exit status.

The `Ice::Application` Class

The preceding structure for the `main` function is so common that Ice offers a class, `Ice::Application`, that encapsulates all the correct initialization and finalization activities. The definition of the class is as follows (with some detail omitted for now):

C++

```
namespace Ice {
    enum SignalPolicy { HandleSignals, NoSignalHandling };

    class Application /* ... */ {
    public:
        Application(SignalPolicy = HandleSignals);
        virtual ~Application();

        int main(int argc, char*[] argv);
        int main(int argc, char*[] argv, const char* config);
        int main(int argc, char*[] argv, const Ice::InitializationData& id);
        int main(int argc, char* const [] argv);
        int main(int argc, char* const [] argv, const char* config);
        int main(int argc, char* const [] argv, const Ice::InitializationData& id);
        int main(const Ice::StringSeq&);
        int main(const Ice::StringSeq&, const char* config);
        int main(const Ice::StringSeq&, const Ice::InitializationData& id);

#ifdef _WIN32
        int main(int argc, wchar_t*[] argv);
        int main(int argc, wchar_t*[] argv, const char* config);
        int main(int argc, wchar_t*[] argv, const Ice::InitializationData& id);
#endif

        virtual int run(int, char*[]) = 0;

        static  const char* appName();
        static  CommunicatorPtr communicator();
        // ...
    };
}
```

The intent of this class is that you specialize `Ice::Application` and implement the pure virtual `run` method in your derived class. Whatever code you would normally place in `main` goes into the `run` method instead. Using `Ice::Application`, our program looks as follows:

C++

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

class MyApplication : virtual public Ice::Application {
public:
    virtual int run(int, char*[]) {

        // Server code here...

        return 0;
    }
};

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

Note that `Application::main` is overloaded: you can pass 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. You also can call `main` with an optional file name or an `InitializationData` structure.

If you pass a configuration file name to `main`, the property settings in this file are overridden by settings in a file identified by the `ICE_CONFIG` environment variable (if defined). Property settings supplied on the command line take precedence over all other settings.

The `Application::main` function does the following:

1. It installs an exception handler for `Ice::Exception`. If your code fails to handle an Ice exception, `Application::main` prints the exception details on `stderr` before returning with a non-zero return value.
2. It installs exception handlers for `const std::string &` and `const char*`. This allows you to terminate your server in response to a fatal error condition by throwing a `std::string` or a string literal. `Application::main` prints the string on `stderr` before returning a non-zero return value.
3. It initializes (by calling `Ice::initialize`) and finalizes (by calling `Communicator::destroy`) a communicator. You can get access to the communicator for your server by calling the static `communicator()` member function.
4. It scans the argument vector for options that are relevant to the Ice run time and removes any such options. The argument vector that is passed to your `run` method therefore is free of Ice-related options and only contains options and arguments that are specific to your application.
5. It provides the name of your application via the static `appName` member function. The return value from this call is `argv[0]`, so you can get at `argv[0]` from anywhere in your code by calling `Ice::Application::appName` (which is often necessary for error messages).
6. It installs a signal handler that properly destroys the communicator.
7. It installs a per-process logger if the application has not already configured one. The per-process logger uses the value of the `Ice.ProgramName` property as a prefix for its messages and sends its output to the standard error channel. An application can also specify an alternate logger.

Using `Ice::Application` ensures that your program properly finalizes the Ice run time, whether your server terminates normally or in response to an exception or signal. We recommend that all your programs use this class; doing so makes your life easier. In addition, `Ice::Application` also provides features for signal handling and configuration that you do not have to implement yourself when you use this class.

Using `Ice::Application` on the Client Side

You can use `Ice::Application` for your clients as well: simply implement a class that derives from `Ice::Application` and place the client code into its `run` method. The advantage of this approach is the same as for the server side: `Ice::Application` ensures that the communicator is destroyed correctly even in the presence of exceptions.

Catching Signals in C++

The simple server we developed in Hello World Application had no way to shut down cleanly: we simply interrupted the server from the command line to force it to exit. Terminating a server in this fashion is unacceptable for many real-life server applications: typically, the server has to perform some cleanup work before terminating, such as flushing database buffers or closing network connections. This is particularly important on receipt of a signal or keyboard interrupt to prevent possible corruption of database files or other persistent data.

To make it easier to deal with signals, `Ice::Application` encapsulates the platform-independent signal handling capabilities provided by the class `IceUtil::CtrlCHandler`. This allows you to cleanly shut down on receipt of a signal and to use the same source code regardless of the underlying operating system and threading package:

C++

```
namespace Ice {
    class Application : /* ... */ {
    public:
        // ...
        static void destroyOnInterrupt();
        static void shutdownOnInterrupt();
        static void ignoreInterrupt();
        static void callbackOnInterrupt();
        static void holdInterrupt();
        static void releaseInterrupt();
        static bool interrupted();

        virtual void interruptCallback(int);
    };
}
```

You can use `Ice::Application` under both Windows and Unix: for Unix, the member functions control the behavior of your application for `SIGINT`, `SIGHUP`, and `SIGTERM`; for Windows, the member functions control the behavior of your application for `CTRL_C_EVENT`, `CTRL_BREAK_EVENT`, `CTRL_CLOSE_EVENT`, `CTRL_LOGOFF_EVENT`, and `CTRL_SHUTDOWN_EVENT`.

The functions behave as follows:

- `destroyOnInterrupt`
  This function creates an `IceUtil::CtrlCHandler` that destroys the communicator when one of the monitored signals is raised. This is the default behavior.

- `shutdownOnInterrupt`
  This function creates an `IceUtil::CtrlCHandler` that shuts down the communicator when one of the monitored signals is raised.
- `ignoreInterrupt`
  This function causes signals to be ignored.
- `callbackOnInterrupt`
  This function configures `Ice::Application` to invoke `interruptCallback` when a signal occurs, thereby giving the subclass responsibility for handling the signal. Note that if the signal handler needs to terminate the program, you must call `_exit` (instead of `exit`). This prevents global destructors from running which, depending on the activities of other threads in the program, could cause deadlock or assertion failures.
- `holdInterrupt`
  This function causes signals to be held.
- `releaseInterrupt`
  This function restores signal delivery to the previous disposition. Any signal that arrives after `holdInterrupt` was called is delivered when you call `releaseInterrupt`.
- `interrupted`
  This function returns `true` if a signal caused the communicator to shut down, `false` otherwise. This allows us to distinguish intentional shutdown from a forced shutdown that was caused by a signal. This is useful, for example, for logging purposes.
- `interruptCallback`
  A subclass overrides this function to respond to signals. The Ice run time may call this function concurrently with any other thread. If the function raises an exception, the Ice run time prints a warning on `cerr` and ignores the exception.

By default, `Ice::Application` behaves as if `destroyOnInterrupt` was invoked, therefore our server `main` function requires no change to ensure that the program terminates cleanly on receipt of a signal. (You can disable the signal-handling functionality of `Ice::Application` by passing the enumerator `NoSignalHandling` to the constructor. In that case, signals retain their default behavior, that is, terminate the process.) However, we add a diagnostic to report the occurrence of a signal, so our `main` function now looks like:

C++

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

class MyApplication : virtual public Ice::Application {
public:
    virtual int run(int, char*[]) {

        // Server code here...

        if (interrupted())
            cerr << appName() << ": terminating" << endl;

        return 0;
    }
};

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

Note that, if your server is interrupted by a signal, the Ice run time waits for all currently executing operations to finish. This means that an operation that updates persistent state cannot be interrupted in the middle of what it was doing and cause partial update problems.

Under Unix, if you handle signals with your own handler (by deriving a subclass from `Ice::Application` and calling `callbackOnInterrupt`), the handler is invoked synchronously from a separate thread. This means that the handler can safely call into the Ice run time or make system calls that are not async-signal-safe without fear of deadlock or data corruption. Note that `Ice::Application` blocks delivery of `SIGINT`, `SIGHUP`, and `SIGTERM`. If your application calls `exec`, this means that the child process will also ignore these signals; if you need the default behavior of these signals in the `exec`'d process, you must explicitly reset them to `SIG_DFL` before calling `exec`.

`Ice::Application` and Properties

Apart from the functionality shown in this section, `Ice::Application` also takes care of initializing the Ice run time with property values. Properties allow you to configure the run time in various ways. For example, you can use properties to control things such as the thread pool size or port number for a server.

Limitations of `Ice::Application`

`Ice::Application` is a singleton class that creates a single communicator. If you are using multiple communicators, you cannot use `Ice::Application`. Instead, you must structure your code as we saw in Hello World Application (taking care to always destroy the communicators).

The `Ice::Service` Class

The `Ice::Application` class is very convenient for general use by Ice client and server applications. In some cases, however, an application may need to run at the system level as a Unix daemon or Windows service. For these situations, Ice includes `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 system services. The `Ice::Service` class is defined as follows:

C++

```
namespace Ice {
    class Service {
    public:
        Service();

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

        int main(int& argc, char* argv[],
                 const Ice::InitializationData& = Ice::InitializationData());
        int main(Ice::StringSeq& args,
                 const Ice::InitializationData& = Ice::InitializationData());

        Ice::CommunicatorPtr communicator() const;

        static Service* instance();

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

        int run(int& argc, char* argv[], const Ice::InitializationData&);

#ifdef _WIN32
        int main(int& argc, wchar_t* argv[], const InitializationData& = InitializationData());

        void configureService(const std::string& name);
#else
        void configureDaemon(bool changeDir, bool closeFiles, 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 Ice::CommunicatorPtr initializeCommunicator(
            int& argc, char* argv[],
            const Ice::InitializationData&);

        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++

```
#include <Ice/Service.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 the argument vector for reserved options that indicate whether the program should run as a system service and removes these options from the argument vector (`argc` is adjusted accordingly). 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 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. Installs a signal handler.
2. Invokes the `initializeCommunicator` member function to obtain a communicator. The communicator instance can be accessed using the `communicator` member function.
3. 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`.
4. Invokes the `waitForShutdown` member function, which should block until `shutdown` is invoked.
5. Invokes the `stop` member function. If `stop` returns `true`, `run` considers the application to have terminated successfully.
6. Destroys the communicator.
7. 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 `CtrlCHandler` when a signal occurs. 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.

- `Ice::CommunicatorPtr initializeCommunicator(int & argc, char * argv[], const Ice::InitializationData & data)`
  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. This function returns false on Windows 95/98/ME.

- `Ice::CommunicatorPtr communicator() const`
  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 & argc, char * argv[], const Ice::InitializationData & data = Ice::InitializationData())`
  `int main(Ice::StringSeq& args, const Ice::InitializationData& = Ice::InitializationData());`
  `int main(int & argc, wchar_t * argv[], const Ice::InitializationData & data = Ice::InitializationData())`
  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` argument 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, char * argv[], const Ice::InitializationData & data)`
  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` argument.

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

Unix Daemons

On Unix platforms, 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 Unix).

See Also

- Hello World Application
- Properties and Configuration
- Communicator Initialization
- Logger Facility
- Portable Signal Handling in C++
- Windows Services