Applications

Building Command-Line and Server Applications.

Overview

- Applications Introduction
- Application Subsystems
- Command Line Applications
- Server Applications
- Command Line Option Handling

Applications with POCO

- The application frameworks in POCO support features needed in virtually every application:
  - command line arguments processing
  - configuration files
  - initialization and shutdown
  - logging

Applications with POCO (cont'd)

- POCO supports two kinds of applications:
  - Command line applications
    These are tools normally invoked from the command line
  - Server applications
    These are applications that typically run as daemons or Windows services (but can also be ran from the command line)

Application Subsystems

- An application can consist of different subsystems.
- Subsystems extend applications in a modular way.
- Subsystems help with the initialization and shutdown of an application.
  - When an application is initialized, all of its registered subsystems are initialized as well.
  - When an application is shutdown, all of its registered subsystems are shutdown as well.

The Subsystem Class

- Subsystems must implement a subclass of Poco::Util::Subsystem.
- const char* name() const
  Returns the name of the subsystem.
- void initialize(Application& app)
  Initializes the subsytem.
- void uninitialize(Application& app)
  Shuts the subsystem down.
- void reinitialize(Application& app)
  re-configures the subsystem (optional; the default implementation calls uninitialize() followed by initialize())

The Subsystem Class (cont'd)

- void defineOptions(OptionSet& options)
  Allows a subsystem to define its own command line arguments.
- If a subsystem wants to define its own command line arguments, it has to implement this member function.
- To effectively handle options, a subsystem should either bind the option to a configuration property or specify a callback to handle the option.
- Otherwise, the option would be passed to the application's handleOption() member function, which probably does not know what to do with it.

Command Line Applications

- Command line applications are implemented by creating a subclass of Poco::Util::Application.
- Application is a subclass of Subsystem.
- There are a few virtual member functions to override.
  - void initialize(Application& self)
  - void reinitialize()
  - void uninitialize()
  - void defineOptions()

  As known from Subsystem.

Command Line Application (cont'd)

- int main(const std::vector<std::string>& args)
  Implements the main logic of the application. The args vector contains all command line arguments that have not been processed as options. Should return the exit code of the application, which can be a value from the ExitCode enumeration (e.g., EXIT_OK, EXIT_USAGE, etc.).

Server Applications

- Server applications are implemented by creating a subclass of Poco::Util::ServerApplication (which is a subclass of Poco::Util::Application).
- Server applications can be ran from the command line, as a Windows service, or a Unix daemon.
- Normally, a server application does its work in a background thread. Therefore, the main() member function will launch the thread, and then wait for an external request to terminate the application (see waitForTerminationRequest()).

Configuration Files

- per default two configurations are created:
  writable MapConfiguration, PRIO_APPLICATION
  readonly SystemConfiguration, PRIO_SYSTEM
- an application named APP searches during startup for files named APP(d).{xml,ini,properties}

```
void MyApplication::initialize(Application& self)
{
        loadConfiguration(); // load default configuration files
        Application::initialize(self);
}
```

Command Line Options

- Applications can define and handle command line arguments (options).
- Command line arguments are usually specified in the format typical for the platform:
  - /option or /option=value on Windows
  - -o, -ovalue, --option or --option:value on Unix

Defining Options

- Options are defined by adding them to the application's OptionSet, using the OptionSet::addOption() member function.
- This is done by overriding the virtual defineOptions() member function.

Anatomy of an Option

- An option has a
  - full name,
  - an optional short name (one character),
  - a description (used for printing an usage statement),
  - and an optional argument name.
- An option can be optional or required.
- An option can be repeatable, which means that it can be given more than once on the command line.

Option Groups and Validation

- An option can be part of an option group.
- At most one option of each group may be specified on the command line.
- An option argument can be automatically validated, by specifying a validator object.

Handling Options

- An option can be bound to a configuration property.
- A callback member function can be specified for handling an option.
- The option can be handled in the application's handleOption() member function.

Validating Option Arguments

- Option arguments can be automatically validated, by specifying a Validator object for the option.
- Two Validator implementations are available:
  - IntValidator checks whether an argument is an integer within a certain range.
  - RegExpValidator verifies that an argument matches a given regular expression.
- Other kinds of validators can be implemented as well.

```
void defineOptions(OptionSet& options)
{
    Application::defineOptions(options);

    options.addOption(
        Option("help", "h", "display help information")
            .required(false)
            .repeatable(false)
            .callback(OptionCallback<MyApp>(this, &MyApp::handleHelp)));

    options.addOption(
        Option("config-file", "f", "load configuration data from a file")
            .required(false)
            .repeatable(true)
            .argument("file")
            .callback(OptionCallback<MyApp>(this, &MyApp::handleConfig)));

    options.addOption(
        Option("bind", "b", "bind option value to test.property")
            .required(false)
            .argument("value")
            .validator(new IntValidator(0, 100))
            .binding("test.property"));
}
```

Displaying Help Information

- The Poco::Util::HelpFormatter class can be used to display command line options help information.
- When the user requests help information, all further command line processing (especially enforcement of required options) should be cancelled. This can be done by calling stopOptionsProcessing().

```
void displayHelp()
{
    HelpFormatter helpFormatter(options());
    helpFormatter.setCommand(commandName());
    helpFormatter.setUsage("OPTIONS");
    helpFormatter.setHeader(
        "A sample application that demonstrates some of the features "
        "of the Poco::Util::Application class.");
    helpFormatter.format(std::cout);
}


void handleHelp(const std::string& name, const std::string& value)
{
    _helpRequested = true;
    displayHelp();
    stopOptionsProcessing();
}
```

Windows Services

- An application based on Poco::Util::ServerApplication can be ran as a Windows service.
- To do this, the service must be registered with the Windows Service Manager by starting the application from the command line and specifying the /registerService option.
- Once no longer needed, the service can be unregistered by specifying the /unregisterService option.
- A user friendly display name for the service can be optionally specified with the /displayName option.

Windows Services (cont'd)

- Once registered the service can be started or stopped using the Services MMC applet, or using the NET START and NET STOP commands on the command line.
- The name of the service executable (excluding directories and the .exe suffix) must be used to identify the service.
- The application can check whether it's running as a service by getting the value of the application.runAsService configuration property.

Windows Services Pitfalls

- Note that the working directory for an application running as a service is the Windows system directory (e.g., "C:\Windows\system32"). Take this into account when working with relative filesystem paths in configuration files or elsewhere.
- Never register a service from a path created by a subst cmd
- Make sure that all libraries are in a location where the executable will find them. Same directory as executable is recommended.

Unix Daemons

- On Unix platforms, an application built on top of the ServerApplication class can be optionally run as a daemon, by specifying the --daemon command line option.
- A daemon, when launched, immediately forks off a background process that does the actual work. After launching the background process, the foreground process exits.
- After the initialization is complete, but before entering the main() method, the current working directory for the daemon process is changed to the root directory ("/"), as it is common practice for daemon processes.

Unix Daemons (cont'd)

- An application can determine whether it is running as a daemon by checking for the application.runAsDaemon configuration property.
- Like with Windows services, be careful when using relative paths in configuration files, as the current working directory of a daemon is the root directory.

applied informatics

www.appinf.com | info@appinf.com
T +43 4253 32596 | F +43 4253 32096