Error Handling and Debugging

Exceptions, error handling and debugging techniques.

appliedinformatics

Overview

- The POCO Exception Classes
- Defining Your Own Exceptions
- Throwing and Catching Exceptions
- Debugging Utilities

The POCO Exception Classes

- All POCO exceptions are subclasses of Poco::Exception
- #include "Poco/Exception.h"
- Poco::Exception is derived from std::exception.
- Some of the subclasses of Poco::Exception:
  - Poco::LogicException (programming errors)
  - Poco::RuntimeException (runtime errors)
  - Poco::ApplicationException (application specific)
- See the reference documentation for the complete list

Poco::Exception

- Poco::Exception has:
  - a name
    (a short static string describing the exception)
  - a message
    (describing the cause of the exception)
  - an optional nested exception

Poco::Exception (cont'd)

- Construct with zero, one or two string arguments (for internal storage, the second string argument will be concatenated with the first one, separated by ": ").
- Construct with a string and a nested exception argument. The nested exception will be cloned.
- Copy-construction and assignment are supported.

Poco::Exception (cont'd)

- `const char* name() const`
  returns the name of the exception
- `const std::string& message() const`
  returns the message text passed in the constructor
- `std::string displayText() const`
  returns the name and the message text, separated by ": "

Poco::Exception (cont'd)

> const Exception* nested() const
returns a pointer to the nested exception, or 0 if there is none

> Exception* clone() const
returns an exact copy of the exception

> void rethrow() const
re-throws the exception

Defining Your Own Exceptions

- Defining subclasses of Poco::Exception is tiresome at best.
  - There are lot's of virtual functions that you must override, all one-liners.
- Therefore, you let some macros do the work.
  - POCO_DECLARE_EXCEPTION to declare the exception class
  - POCO_IMPLEMENT_EXCEPTION to implement it

```
// MyException.h

#include "Poco/Exception.h"

POCO_DECLARE_EXCEPTION(MyLib_API, MyException, Poco::Exception)
```

```
// MyException.h

#include "Poco/Exception.h"

POCO_DECLARE_EXCEPTION(MyLib_API, MyException, Poco::Exception)
```

$$\downarrow$$

```
class MyLib_API MyException: public Poco::Exception
{
public:
    MyException();
    MyException(const std::string& msg);
    MyException(const std::string& msg, const std::string& arg);
    MyException(const std::string& msg, const Poco::Exception& nested);
    MyException(const MyException& exc);
    ~MyException();
    MyException& operator = (const MyException& exc);
    const char* name() const;
    ...
};
```

```
// MyException.cpp

#include "MyException.h"

POCO_IMPLEMENT_EXCEPTION(MyException, Poco::Exception,
    "Something really bad happened...")
```

```
// MyException.cpp

#include "MyException.h"

POCO_IMPLEMENT_EXCEPTION(MyException, Poco::Exception,
    "Something really bad happened...")
```

↓

```
...
const char* MyException::name() const throw()
{
    return "Something really bad happened...";
}
...
```

Throwing and Catching Exceptions

- In good old C++ tradition, you should always throw by value and catch by (const) reference.
- Use displayText() for logging the exception.
- You can store an exception and rethrow it at a later time.

```
#include "Poco/Exception.h"
#include <iostream>

int main(int argc, char** argv)
{
    Poco::Exception* pExc = 0;
    try
    {
        throw Poco::ApplicationException("just testing");
    }
    catch (Poco::Exception& exc)
    {
        pExc = exc.clone();
    }
    try
    {
        pExc->rethrow();
    }
    catch (Poco::Exception& exc)
    {
        std::cerr << exc.displayText() << std::endl;
    }
    delete pExc;
    return 0;
}
```

Assertions

- POCO has various macros for runtime checks.
- poco_assert(cond)
  throws an AssertionViolationException if cond ≠ true
- poco_assert_dbg(cond)
  similar to poco_assert, but only "armed" in debug builds
- poco_check_ptr(ptr)
  throws a NullPointerException if ptr is null
- poco_bugcheck(), poco_bugcheck_msg(string)
  throws a BugcheckException

Assertions (cont'd)

- poco_assert, poco_assert_dbg, poco_check_ptr and poco_bugcheck add the current file and line number to the exception text.
- In a debug build, and if a debugger is present (e.g., under Visual C++), a breakpoint will be triggered.

```
void foo(Bar* pBar)
{
    poco_check_ptr (pBar);

    ...
}

void baz(int i)
{
    poco_assert (i >= 1 && i < 3);

    switch (i)
    {
    case 1:
        ...
        break;
    case 2:
        ...
        break;
    default:
        poco_bugcheck_msg("i has invalid value");
    }
}
```

NestedDiagnosticContext

- Based on Neil Harrison's article "Patterns for Logging Diagnostic Messages" in PLOP3.
- A NDC maintains a stack of context information, consisting of
  - an informational string (method name), and
  - source code file name and line number.
- NDCs are especially useful for tagging log messages with context information (stack traces).
- Every thread has its own private NDC.

NestedDiagnosticContext (cont'd)

- #include "Poco/NestedDiagnosticContext.h"
- Class NDCScope takes care of pushing a context onto the context stack upon entry of a method and popping it from the stack upon exit.
- poco_ndc(func) or poco_ndc_dbg(func)
  declares a NDCScope (poco_ndc_dbg only in a debug build)
- use NDC::dump() to output a stack trace
- Note: NestedDiagnosticContext is typedef'd to NDC

```
#include "Poco/NestedDiagnosticContext.h"
#include <iostream>

void f1()
{
    poco_ndc(f1);

    Poco::NDC::current().dump(std::cout);
}

void f2()
{
    poco_ndc(f2);

    f1();
}

int main(int argc, char** argv)
{
    f2();

    return 0;
}
```

Debug and Release Builds

- POCO supports separate debug and release builds.
- In a debug build, additional runtime checks are performed, and additional debugging features are available.
- You can use that in your own code, too.
- For a debug build, the preprocessor macro _DEBUG must be defined.
- In a debug build, the macros poco_debugger() and poco_debugger_msg(message) can be used to force a breakpoint (if the code is running under control of a debugger)

Debug and Release Builds (cont'd)

- Note that poco_assert, poco_check_ptr and poco_bugcheck are enabled both in debug and in release builds.
- In debug builds, if a debugger is available, a breakpoint is triggered before the exception is thrown.
- poco_assert_dbg and poco_debugger are enabled in debug builds only.

Debugger Interface

- Class Poco::Debugger provides an interface to the debugger.
- #include "Poco/Debugger.h"
- Use bool Debugger::isAvailable() to check whether you are running under a debugger.
  - On Unix systems, to enable debugger support, set the environment variable POCO_ENABLE_DEBUGGER.
- Use void Debugger::enter() to force a breakpoint.
- Use void Debugger::message() to write a message to the debugger log, or to standard output.

applied informatics