descore Programmer's Guide

Notice

Copyright

Trademarks

Contents

1.0 Logging 5

1.1 Basic logging functions 5
1.2 Logging with streaming semantics 5
1.3 Controlling the log file 5
1.4 Customizing the log header 6
1.5 Quiet mode 6
1.6 Redirecting console output 7
1.7 Writing to the system log 7
1.8 Custom log files 8
1.9 Controlling custom log files 9
1.10 Re-opening log files 10
1.11 Log prefix 10
1.12 Logging and threading 10

2.0 Assertions 12

2.1 Basic assertion macros 12
2.2 Catching assertion failures 12
2.3 Deferring exceptions 14
2.4 Declaring new exception types 15
2.5 Assertion context 17
2.6 Error hooks 17
2.7 Fatal hooks 18
2.8 Abort on error 19
2.9 Warnings 19
2.10 Disabling assertions and warnings 19
2.11 Debugger breakpoints 19
2.12 Accessing parameters from code 20
2.13 Compilation issues 20

3.0 String Conversions and Helper Functions 21

3.1 std::string 21
3.2 istrcaststream 21
3.3 ostrcaststream 23
3.4 String conversion for pointers 24
3.5 Parsing input streams 25
3.6 Delimited strings 25
3.7 Conversion to string 26
3.8 Conversion from string 27
3.9 string_cast 29
3.10 Conversion of STL containers 29

4.0 Tracing 31

4.1 Basic tracing 31
4.2 Trace contexts 31
4.3 Specifying the trace context explicitly 32
4.4 Modifying the global trace context 33
4.5 Trace keys 33
4.6 Associating trace keys with log files 35
4.7 Determining if tracing is enabled 35
4.8 Combining trace keys 36
4.9 Controlling tracing 37
4.10 The TraceContext helper class 38
4.11 Specifying tracing from the command line 38
4.12 Listing the available trace keys 39
4.13 Controlling trace output 40
4.14 Unconditional trace output 40
4.15 Summary of macros and functions 41

5.0 Archiving 44
5.1 Overview 44
5.2 Archive modes 44
5.3 Opening and closing an archive 44
5.4 Archiving data 45
5.5 Archiving constant strings 46
5.6 Detecting end-of-archive 47
5.7 User types 47

6.0 Parameters 49
6.1 Declaring parameters 49
6.2 Parameter descriptions 50
6.3 Enumeration parameters 50
6.4 Accessing parameters 51
6.5 Static helper functions 52
6.6 Setting indexed parameters 53
6.7 Parameter modification callbacks 53
6.8 Determining if a parameter has been modified 53
6.9 Parameter groups 54
6.10 Parameter group arrays 55
6.11 Setting parameters from a configuration file 56
6.12 Setting parameters from the command line 57
6.13 Automatically providing parameter help 58
6.14 Generating configuration files 58
6.15 Creating new parameter types 58

7.0 Threads 60
7.1 Creating threads 60
7.2 Static helper functions 61
7.3 Exceptions 61
7.4 Synchronization 62
7.5 Static thread-local data 63
7.6 Dynamic thread-local data 63

8.0 Iterators 65
8.1 Iterator 65
8.2 ConstIterator 66
8.3 Maps and multimaps 66
8.4 Sorted containers 66

9.0 Map Iterators 67
9.1 Iterating over map values 67
9.2 Iterating over map keys 67
9.3 Iterating over (key, value) pairs 68

10.0 Enumerations 69
10.1 Basic usage 69
10.2 String conversion 70
10.3 Initialization values 70
10.4 Enumeration types 70
10.5 Enumeration interface 71
10.6 Use in parameters 72

11.0 Type Traits 73
11.1 Type macros 73
11.2 Type traits structure 73

12.0 Coverage Assertions 74
12.1 Line coverage 74
12.2 Basic coverage 74

12.3 Cross-coverage 75
12.4 Covering return values 76
12.5 Covering conditional tests 76
12.6 Controlling coverage 77
12.7 Other error messages 78
12.8 Templates 78

13.0 String Buffers 79

13.1 Member functions 79

14.0 Miscellaneous 81

14.1 Types 81
14.2 Macros 81
14.3 Helper functions 82
14.4 String table 83
14.5 'Using' declarations 84
14.6 Hexadecimal helpers 84
14.7 Detecting memory leaks with AllocTracker 85
14.8 Using pointers in STL trees 85
14.9 Comparison of STL trees 86
14.10 Pretty-printing a set of strings 86
14.11 Pretty-printing a table of strings 86
14.12 Statistics 88

1.0 Logging

```
#include <descore/descore.hpp>
```

The descore `log()` function is intended for use in place of `printf()` or `cout`.[1] It can be used with exactly the same syntax (either printf or streaming semantics), but the output is directed to both `stdout` and the log file, ensuring that all output is preserved.

1.1 Basic logging functions

```
void log      (const char *message, ...)
void logv     (const char *message, va_list args)
void log_puts (const char *message)
```

Output a message to both the log file (if it is open) and `stdout` using printf (`log`) or vprintf (`logv`) semantics, or as a raw string (`log_puts`). Internally, output is buffered until a newline character is encountered, then everything up to the last newline is comitted to both the log file and `stdout`.

```
void logerr  (const char *message, ...)
void logerrv (const char *message, va_list args)
```

Output a message to both the log file (if it is open) and `stderr` using printf (`log`) or vprintf (`logv`) semantics. Internally, output is buffered until a newline character is encountered, then everything up to the last newline is comitted to both the log file and `stderr`.

1.2 Logging with streaming semantics

Calling `log()` or `logerr()` with no arguments returns an `std::ostream`. As with the other logging functions, output is internally buffered until a newline character is encountered.

Examples

```
log() << "numPoints = " << numPoints << "\n";    // output to log and stdout
logerr() << "bad vector: " << myVec << "\n";     // output to log and stderr
```

Note: During static initialization, descore redirects `cout`, `cerr` and `clog` so that the standard streams will also be captured in the log.

1.3 Controlling the log file

```
void descore::initLog (const char *filename,
                       int csz = 0, char **rgsz = NULL)
```

Open the log file (closing any existing open log file) with a header that includes the current time and optionally the command line. If the parameter `log.append` is false (default), any existing log file with the same name will be overwritten. Otherwise, the existing file will be opened for appending.

[1] With the exception of single-line progress output using `"\r"` that only makes sense at run-time and truly does not belong in the log.

void descore::flushLog ()

Forces output of all internal buffers, even if there is no newline character. Calls `fflush()` on `stdout`, `stderr`, and the log file.

void descore::closeLog ()

Close the log file. A new log file can be opened with `descore::initLog()`.

In some cases the command line may be destructively parsed before calling `initLog()`, but it is still desirable to call `initLog()` with the original command line. The helper structure `descore::CopyArgs` is provided to create a temporary copy of the command line that can then be passed to `initLog()`. Its constructor takes `csz` and `rgsz` as arguments, and it has public members `csz` and `rgsz`.

Example

```
descore::CopyArgs args(csz, rgsz);
Parameter::parseCommandLine(csz, rgsz);
descore::initLog("test.log", args.csz, args.rgsz);
```

1.4 Customizing the log header

The `descore::initLog()` function will create the following header at the start of every log file:

```
#
#  Log started at Tue Feb 24 16:46:46 2009 on host 'Windows machine'
#
```

A customized log header can be prepended to this header by calling the function

void descore::setLogHeader (void (*fn) ())

The specified function will be called by `initLog()` for the purpose of creating a custom log header whenever a new log file is opened. Call with `NULL` to unregister the log header function.

1.5 Quiet mode

In some cases it is desirable to suppress the output to `stdout/stderr` and only send logging output to the log file. This *quiet mode* of operation is controlled by the following function:

bool descore::setLogQuietMode (bool quiet = true)

This function enables/disables quiet mode according to the *quiet* argument. It returns the current quiet mode status.

The log quiet mode can also be controlled via the boolean parameter `log.quiet`.

1.6 Redirecting console output

The output to `stdout/stderr` can be redirected by calling the function

```
void descore::setLogConsoleOutput (
                                  void (*fn) (const char *, FILE *))
```

Instead of sending output to `stdout/stderr`, the specified function will be called with the string that would have been written and the file (`stdout/stderr`) to which it would have been written. Call with `NULL` to un-register the console output function and restore output to `stdout/stderr`.

1.7 Writing to the system log

Two functions are provided for writing to the system log (this applies to Linux builds only; on Windows these functions have no effect):

```
void logsys  (const char *message, ...)
void logsysv (const char *message, va_list args)
```

Output a message to the log file, `stderr`, and the system log using printf (`logsys`) or vprintf (`logsysv`) semantics. Internally, output is buffered until a newline character is encountered, then everything up to the last newline is comitted to the log file, `stderr`, and the system log.

The level, identity and facility properties for the system log output are controlled by the following parameters:

Property	Parameter	Default
level	`log.syslog.level`	LOG_INFO
identity	`log.syslog.identity`	"descore"
facility	`log.syslog.facility`	LOG_LOCAL1

By default, all system log output is copied to `stderr`, but this can be suppressed by setting `log.syslog.quiet` to true (the default is false).

All error output (output generated by `logerr` or `logerrv`) can optionally be copied to the system log by setting `log.syslog.log_errors` to true (the default is false). Errors copied to the system log use the same identity and facility as `logsys` and `logsysv`, but the log level is set to LOG_ERR.

The system log can also be written using streaming semantics; calling `logsys()` with no arguments returns an `std::ostream`.

Examples

```
logsys("Found %d corrupted bits\n", numCorruptedBits);
logsys() << "Fatal error on node " << node << "\n";

$ more /var/log/local1
Sep 15 09:29:36 drdlogin0032 descore[14765]: Found 27 corrupted bits
Sep 15 09:29:36 drdlogin0032 descore[14765]: Fatal error on node (1,2,4)
```

1.8 Custom log files

Custom log files can be created to gain more detailed control over the log output using one of the following three functions:

```
descore::LogFile descore::openLog   (string filename)
descore::LogFile descore::appendLog (string filename)
descore::LogFile descore::openLog   (descore::ILogOutput *output)
```

Each of these functions creates a log file whose output is sent to each of `stdout`, the main log file, and a new file or a logging interface. The first function creates a new log file with the specified filename, replacing it if it already exists. The second function is similar but an existing file will be appended to instead of replaced. The third function specifies a logging interface instead of a file:

```
struct descore::ILogOutput
{
    virtual ~ILogOutput () {}
    virtual void write (const char *line) = 0;
    virtual void close () {}
};
```

An implementation of this interface must be supplied; each line of the log output (delimited by carriage returns) is written to the interface via the `write()` callback. This function will also be called with partial output (i.e. a partial line with no carriage return) if the custom log file's internal line buffer is non-empty and `descore::flushLog()` is called. Finally, the optional `close()` callback is called when the log file is closed.

Additional forms of the logging functions are provided for custom log files that take a `descore::LogFile` as their initial argument:

```
void log      (descore::LogFile f, const char *msg, ...)
void logv     (descore::LogFile f, const char *msg, va_list args)
void log_puts (descore::LogFile f, const char *msg)
```

Output a message to the specified log file using printf (`log`) or vprintf (`logv`) semantics, or as a raw string (`log_puts`). By default, the output is copied to both `stdout` and the main log file.

Logging to a custom log file with streaming semantics is also supported by passing the log file as the single argument to `log()`.

Example

```
descore::LogFile f = openLog("custom.log");
log(f, "deleting %d records\n", numDeletedRecords);
log(f) << "deleting " << numDeletedRecords << " records\n";
```

There are three pre-defined `descore::LogFile` constants:

```
descore::LOG_STDOUT
descore::LOG_STDERR
descore::LOG_SYS
```

The function calls

```
log(msg, ...)
logerr(msg, ...)
logsys(msg, ...)
```

are simply convenience wrappers for the equivalent function calls

```
log(descore::LOG_STDOUT, msg, ...)
log(descore::LOG_STDERR, msg, ...)
log(descore::LOG_SYS, msg, ...)
```

A call to `descore::flushLog()` will flush all log files, including custom log files. A custom log file can be closed using the function

```
void descore::closeLog (descore::LogFile f)
```

1.9 Controlling custom log files

By default, the output of a custom log file is copied to (1) the file or logging interface specified when the log file was created, (2) the main log file, and (3) `stdout`. The following functions provide more control over a log file's output.

```
bool descore::setLogQuietMode (descore::LogFile f, bool quiet)
```

Enable or disable the copying of the specified log file's output to `stdout` or `stderr`. Returns the previous quiet mode setting for the log file. Quiet mode defaults to false (so that output is copied to `stdout` or `stderr`). If the global quiet mode has been enabled using `setLogQuietMode()` (Section 1.5) then the per-log-file quiet mode setting is ignored and output to `stdout`/`stderr` is suppressed for all log files.

```
void descore::setLogConsoleFile (descore::LogFile f,
                                 FILE *console_out);
```

Specify whether the console output for the specified log file is sent to `stdout` or `stderr` (`console_out` must be one of these two files). Note that if console output has been globally redirected using `setLogConsoleOutput()` (Section 1.6), then the console output for this log file will be passed to the registered function with the `console_out` file specified as the second argument.

```
bool descore::setLogCopyMode (descore::LogFile f,
                              bool copy_to_main_log)
```

Enable or disabled the copying of the specified log file's output to the main log file (enabled by default). Returns the previous setting.

```
void descore::setLogSyslogEnabled  (descore::LogFile f,
                                    bool copy_to_syslog)
void descore::setLogSyslogIdentity (descore::LogFile f,
                                    string identity)
void descore::setLogSyslogFacility (descore::LogFile f,
                                    int facility)
void descore::setLogSyslogLevel    (descore::LogFile f,
                                    int level)
```

Control whether or not, and how, a log file's output is copied to the system log. The first function enables or disables system log output for this file; when system log output is enabled, every line (delimited by carriage returns) will be sent to the system log. The next three functions specify the system log identity, facility and level (respectively) that will be used when the system log output is enabled. These three functions are optional; when a log file is created, the syslog identity, facility and level are set from the corresponding parameters described in Section 1.7.

These functions can also be used to modify the predefined log files by specifying one of `descore::{LOG_STDOUT, LOG_STDERR, LOG_SYS}` as the log file argument, subject to the following restrictions:

- `setLogConsoleFile()` and `setLogCopyMode()` cannot be used with `descore::LOG_STDOUT` or `descore::LOG_STDERR`
- `setLogSyslogEnabled()` cannot be used with `descore::LOG_SYS`
- `setLogSyslogIdentity()`, `setLogSyslogFacility()` and `setLogSyslogLevel()` cannot be used with `descore::LOG_STDERR`

1.10 Re-opening log files

A duplicate handle to an existing log file can be created using the function:

```
void descore::LogFile descore::reopenLog (descore::LogFile)
```

The new `LogFile` behaves identically to the existing one, but can subsequently be modified using the functions described in the previous section without affecting the original file. The original `LogFile` can be either a user-defined log file or one of the three pre-defined log files. When a user-defined log file is re-opened, the underlying file or ILogOutput interface is closed when the *last* copy of the log file is closed.

1.11 Log prefix

An optional log prefix can be specified for any log file. This is a string that is prepended to the start of every line of output (where a line is defined by carriage returns and not by individual calls to `log()` functions). The string is processed by `strftime` and can contain any of the special escape sequences recognized by this function. A log prefix is specified using the following function:

```
void setLogPrefix (descore::LogFile f, string prefix)
```

Set the log prefix for the specified log file, which can be a custom log file or one of `descore::{LOG_STDOUT, LOG_STDERR, LOG_SYS}`. Each line in the log file will begin with this prefix.

Example

```
setLogPrefix(descore::LOG_STDOUT, "%m/%d/%y %H:%M:%S - ");
log("Discovered two options:\n    apple\n");
descore::Thread::sleep(2000);
log("    orange\n);

// Output
07/17/12 17:43:12 - Discovered two options
07/17/12 17:43:12 -     apple
07/17/12 17:43:14 -     orange
```

1.12 Logging and threading

Logging is thread-safe, and each line of output (as delimited by carriage returns) is guaranteed to be contiguous and interleaved with other lines in the output. Furthermore, when a single `log()` statement contains multiple lines (i.e. multiple carriage returns), then these lines are guaranteed to be consecutive.

Example

```
// Each of these lines will be intact in the log file, but output from
// another thread may be interleaved in between them.
log(“First line\n”);
log(“Second line\n”);

// Same output as above, but the two lines are guaranteed to be consecutive
log(“First line\nSecond line\n”);
```

2.0 Assertions

```
#include <descore/descore.hpp>
```

The descore assert macro replaces the standard assert macro and provides improved support for assertions through context information, optional error messages with printf semantics, and full integration with an exception mechanism. All error messages are copied to both `stderr` and the current log file.

2.1 Basic assertion macros

```
assert(expr)
assert(expr, msg, ...)
assert_always(expr)
assert_always(expr, msg, ...)
die(msg, ...)
```

The `assert` macro can be used in the standard way, e.g.

```
assert(size < maxSize);
```

The error output generated by an assertion failure consists of the prefix "`Assertion failed: `", followed by the failing expression, followed on the next line by the file name, line number and current function:

```
Assertion failed: size < maxSize
src/blah.cpp:97: void allocateArray(int)
```

An optional error message with printf semantics can be supplied:

```
assert(size < maxSize, "size (%d) exceeds maximum size (%d)", size, maxSize);
```

When a message is supplied, the first line of the error output is replaced by the prefix "`Error: `" followed by the message:

```
Error: size (70) exceeds maximum size (64)
src/blah.cpp:97: void allocateArray(int)
```

Optimized builds may or may not include assertions; to guarantee that a specific assertion is included in all build configurations use `assert_always`:

```
assert_always(size < maxSize);
```

Finally, the `die` macro is provided as a convenience and is completely equivalent to `assert_always(false, msg, ...)`, e.g.

```
die("Failed to allocate %d records", numRecords);
```

An error message *must* be supplied when using the `die` macro.

2.2 Catching assertion failures

An assertion failure throws an exception of type `descore::runtime_error`, which inherits from `std::runtime_error`. If this exception is not caught then the application's termination handler is invoked; descore installs a default termination handler that logs the error message and exits. A developer can choose to explicitly catch this exception, in which case the code in the catch block can decide whether or not to log the error message and whether or not to continue execution.

The `descore::runtime_error` class defines the following member functions that provide information regarding the error:

`const char *file () const`

Returns the name of the file in which the error occurred.

`int line () const`

Returns the line number within the file at which the error occurred.

`const char *function () const`

Returns the pretty-printed signature of the function in which the error occurred.

`const char *context () const`

Returns the assertion context string (Section 2.5).

`const char *expression () const`

Returns the expression that was tested and evaluated to false. For an assertion failure generated using the `die` macro, returns `"false"`.

`const char *message () const`

Returns the formatted message that was supplied to the assertion macro, or `""` if no message was supplied.

`virtual const char *what () const throw ()`

Returns a string containing the entire error output and marks the exception as "handled".

In addition to the above accessors, three member functions are provided for handling errors:

`void reportFatal () const`

Logs the full error output and calls the fatal error hook (Section 2.6). Marks the exception as "handled".

`void reportAndExit (int exitStatus = -1) const`

Calls `reportFatal()` and then exits with the supplied exit status.

`void handled () const;`

Marks the error as "handled" without actually reporting it.

Important note:

Every error that is caught *must* be either re-thrown or explicitly handled by calling one of `reportFatal()`, `reportAndExit()`, `handled()`, or `what()`. If an error object detects within its destructor that it has not been handled, it will noisily self-report (by disabling the log quiet mode before logging a complaint about an unhandled error along with the error message) and then immediately abort the application.

Example

```
try
{
     m_outfile = initDataOutputFile();
}
catch (descore::runtime_error &e)
{
     if (flags & OUTPUT_FILE_REQUIRED)
          throw;
     if (flags & OUTPUT_FILE_REPORT_FAILURE)
          logerr("%s", e.what());
     else
          e.handle();   // Required!!!  Can't just drop the error.
}
```

Finally, arbitrary strings can be appended to an exception's output using the `append()` member function:

```
void append (const char *msg, ...)
void append (const string &msg)
```

Example

```
try
{
     m_outfile = initDataOutputFile();
}
catch (descore::runtime_error &e)
{
     e.append("While creating output file for record %d\n", m_recordId);
     throw;
}
```

2.3 Deferring exceptions

Instead of immediately handling or rethrowing an exception, it is also possible to create a copy of the exception to be handled or thrown at some later point in time. Two member functions are provided to implement this behavior.

```
descore::runtime_error *clone () const
```

Creates a local copy of an exception. The original exception is marked as "handled"; the local copy is not and must be handled or rethrown before it is deleted.

```
void rethrow () const
```

The exception returned by `clone()` can be thrown at a later point in time by calling this function.

Example

```
try
{
     computeInterval();
}
catch (descore::runtime_error &e)
{
     m_error = e.clone();
}
…
if (m_error)
     m_error->rethrow();
```

2.4 Declaring new exception types

A helper macro is provided to define an arbitrary hierarchy of exception types that derive from `descore::runtime_error`:

```
DECLARE_EXCEPTION(exception_type)
DECLARE_EXCEPTION(exception_type, base)
```

The first form defines a new exception type that inherits directly from `descore::runtime_error`; the second form specifies which exception type to inherit from (which should also be an exception type defined `using DECLARE_EXCEPTION`). Use `assert_throw` to create an assertion that throws a custom exception type when the assertion fails:

```
assert_throw(exception_type, expr, ...)
```

The first argument specifies the exception type to throw on failure, and the remaining arguments are the same as for `assert`.

descore assumes that an assertion with a custom exception type is used to define program behavior rather than to provide self-verification in debug builds, so as with `assert_always` the `assert_throw` macros are guaranteed to be included in all builds.

Example

```
DECLARE_EXCEPTION(out_of_range_error);
DECLARE_EXCEPTION(negative_value_error, out_of_range_error);

custom_float sqrt (custom_float x)
{
     assert_throw(negative_value_error, x >= 0);
     assert_throw(out_of_range_error, x < MAX_SQRT_INPUT);
     …
}

custom_float sqrt_or_negone (custom_float x)
{
     try
     {
          return sqrt(x);
     }
     catch (out_of_range_error &e)
     {
          e.handle();  // Required!!!
     }
     return -1;
}
```

Custom exception types can also be constructed and thrown manually. Three constructors are provided for this purpose:

***exception_type* ()**

Constructs the exception with an empty `what()` string.

***exception_type* (const string &message)**

Sets the `what()` string to the specified message.

***exception_type* (const char *message, ...)**

Sets the `what()` string to the message specified using printf semantics.

Example

```
DECLARE_EXCEPTION(out_of_range_error);

custom_float sqrt (custom_float x)
{
     if (x < 0)
     {
          g_flags |= CPU_FLAG_SQRT_NEG_ARG;
          throw out_of_range_error("Negative argument supplied to sqrt()");
     }
     …
}
```

A manually-thrown exception will not have any associated context information, so in general the assertion macros should be preferred. When a manually-thrown error is reported, the output is prefixed with a message identifying the type of the error, e.g.:

```
Encountered run-time error of type 'struct out_of_range_error'
Negative argument supplied to sqrt()
```

To provide additional constructors, member variables, or member functions, a custom exception structure can be explicitly declared that inherits from `descore::runtime_error` (or from another exception). The structure must contain must contain the macro `IMPLEMENT_EXCEPTION`, whose arguments are the same as for `DECLARE_EXCEPTION`:

```
IMPLEMENT_EXCEPTION(exception_type);
IMPLEMENT_EXCEPTION(exception_type, base);
```

This macro ensures that the exception type is compatible with the assertion infra-structure (including `assert_throw` and the ability to defer exceptions), and it provides the same three constructors as `DECLARE_EXCEPTION` (default; string message; message with printf semantics).

If the exception structure contains any member variables, then a `copy()` function must be declared to copy those variables from another exception of the same type. The `copy()` function is automatically called by the assertion infrastructure, and it has the same signature as a copy constructor:

```
void copy (const exception_type &rhs);
```

Example

```
struct out_of_range_error : public descore::runtime_error
{
     IMPLEMENT_EXCEPTION(out_of_range_error);

     // Constructor used to manually throw this error
     out_of_range_error (int minval, int maxval, int arg) :
          descore::runtime_error("%d is not in [%d,%d]", arg, minval, maxval),
          bad_arg(arg)
     {
     }

     void copy (const out_of_range_error &rhs)
     {
          bad_arg = rhs.bad_arg;
     }

     int bad_arg;
};
```

2.5 Assertion context

The assertion macros assume the existence of a function with the exact name `getAssertionContext()`, which they use to obtain context information. This function must return a `string` (or a const reference to a `string`), and if the string is non-empty then it is displayed at the start of the error message. A default, global implementation of this function is defined that simply returns an empty string. Classes can implement `getAssertionContext()` to provide additional context information that will then be automatically included for assertion failures within class member functions. This allows the error message to indicate not only which line of code caused the error, but the specific class instance in which the error occurred.

Example

```
class Record
{
public:
     const string &getAssertionContext () const
     {
          return m_name;
     }
private:
     string m_name;
};
```

If a class defines a non-static `getAssertionContext()` function, then the standard assertion macros will not work within any of this class' static member functions. The following "static" macros, identical to the non-static macros but with an `s_` prefix, explicitly call `::getAssertionContext()`; they can therefore be used within static member functions.

```
s_assert(expr, ...)
s_assert_always(expr, ...)
s_assert_throw(expr, ...)
s_die(msg, ...)
```

The global assertion context can be modified by providing a context function that takes no arguments and returns a string; `::getAssertionContext()` will then defer to this function instead of returning an empty string. The following function is used to specify the global assertion context:

```
descore::assertion_context_function_t
setGlobalAssertionContext (descore::assertion_context_function_t fn)
```

This function registers a new global assertion context function and returns the previous one.

Example

```
string applicationContext ();
{
     return str("During phase %d of processing", g_phase);
}
…
descore::setGlobalAssertionContext(&applicationContext);
```

2.6 Error hooks

An application can register a function to be called when an error object is constructed. The callback function (the *hook*) has the opportunity to append additional information to the error output, but should not have any other side effects

in case the error is caught and ignored. The function must have the following prototype:

```
void error_hook (descore::runtime_error &error)
```

Pointers to this function type are typedefed to `descore::ErrorHook`. An error hook can be registered using the following function:

```
descore::ErrorHook descore::setErrorHook (descore::ErrorHook fn)
```

This function returns the current error hook and replaces it with the new one, allowing applications to temporarily modify the error hook, or to explicitly call the previous hook from the new hook.

Example

```
static descore::ErrorHook s_prevHook;

static void errorCallback (const descore::runtime_error &e)
{
     if (g_isArchiving)
          e.append("Error occurred while archiving data\n");
     s_prevHook(e);
}

int main (int csz, char *rgsz[])
{
     s_prevHook = descore::setErrorHook(&errorCallback);
     …
```

2.7 Fatal hooks

An application can also register a hook to be called after a fatal error is reported via `reportFatal()`. The hook again has the opportunity to provide additional error output, but should be careful to avoid relying on any stack-based data since the hook may be called from the termination handler after the stack has been fully unwound. The hook can also choose to terminate the application in the appropriate manner. The hook must have the following prototype:

```
void fatal_hook (const descore::runtime_error &error)
```

Pointers to this function type are typedefed to `descore::FatalHook`. A fatal hook can be registered using the following function:

```
descore::FatalHook descore::setFatalHook (descore::FatalHook fn)
```

This function returns the current fatal hook and replaces it with the new one, allowing applications to temporarily modify the fatal hook, or to explicitly call the previous hook from the new hook.

Example

```
static void handleFatalError (const descore::runtime_error &)
{
     logError("Fizz wizard encountered a fatal error: aborting.\n");
     exit(-1);
}

int main (int csz, char *rgsz[])
{
     descore::setFatalHook(&handleFatalError);
     …
}
```

2.8 Abort on error

The default error behavior is to throw an exception. If the Boolean parameter `assert.abortOnError` is true, then any error will instead be immediately reported following which the application will be terminated, and no exception will be thrown. This mimics the behavior of traditional assertion failures.

2.9 Warnings

The `warn` macro (and its static equivalent `s_warn`) can be used to create assertions that print a warning without throwing an exception.

```
warn(expr, msg, ...)
s_warn(expr, msg, ...)
```

`warn` has the same syntax as `assert`, except that a message in mandatory.

Example

```
warn(size > 0, “Creating zero-sized object”);
```

An error is generated when the maximum number of warnings is exceeded. This maximum number, which defaults to 5, can be changed via the integer parameter `assert.maxWarnings`.

The current running total of the number of warnings can be reset to zero using

```
void descore::resetWarningCount ()
```

2.10 Disabling assertions and warnings

Specific assertions and warnings implemented using the above macros can be disabled based on the full text of their error output. The parameter `assert.disabledAssertions` is a vector of strings, each of which is compared to the error output. If any of the strings (which can contain the * and ? wildcards) match any substring of the error output, then the assertion or warning is skipped and program execution resumes normally. This decision is made after any error hooks have been called to append to the error output.

Example

```
uint32 readMemory (uint32 addr)
{
     assert(addr, “Attempted to read address 0”);
     …
}

$ memtest -assert.disabledAssertions=’[“Attempted to read address 0”]’
```

The runtime error class has an `isDisabled()` member function that indicates whether or not the error is disabled based on its current output string and the value of `assert.disabledAssertions`. Error hooks can use this function to suppress any side effects that are undesirable for disabled errors.

2.11 Debugger breakpoints

When a non-disabled warning or assertion failure occurs while running within a debugger, a breakpoint is automatically triggered so that the program state may be inspected at the point of the error. This may be undesirable if the application

generates a large number of errors that are normally caught and ignored, so there are two ways to disable this behavior:

1. From the debugger, set the global variable `descore::enable_debug_breakpoint` to 0

2. From the command line, set the boolean parameter `assert.disableDebugBreakpoint`

Example

```
(gdb) run -assert.disableDebugBreakpoint
```

2.12 Accessing parameters from code

The various parameters that affect the behavior of assertions can be accessed directly from code within the structure `descore::assertParams`, where the parameter `assert.parameter_name` corresponds to the structure member `descore::assertParams.parameter_name`. The use of this structure requires that the file `descore/AssertParams.hpp` be included.

Example

```
#include <descore/AssertParams.hpp>
…
descore::assertParams.maxWarnings = 5;
descore::assertParams.abortOnError = true;
descore::assertParams.disableDebugBreakpoint = true;
descore::assertParams.disabledAssertions->push_back("Attempted to read address 0");
```

2.13 Compilation issues

To disable the `assert`, `s_assert`, `warn` and `s_warn` macros, compile the code with `_ASSERTOFF` defined. The other macros will still be enabled.

Unfortunately, the standard `assert.h` header does not contain #ifdef protection, so if it is somehow #included after `descore.hpp` then the assert macro will be redefined, likely causing the build to fail. This situation can be remedied by re-#including `descore/assert.hpp`.

To completely suppress the descore assert macro definitions so that another implementation can be used instead, define `DISABLE_DESCORE_ASSERT`.

3.0 String Conversions and Helper Functions

```
#include <descore/descore.hpp>
```

The descore library provides a few basic classes and helper functions to simplify the task of working with strings and converting primitives or objects to and from strings.

3.1 std::string

The `descore.hpp` header includes the declaration

```
using std::string;
```

Thus, the shorter '`string`' can be used everywhere in place of '`std::string`'. Additionally, the `*` operator is overloaded to convert a string to a `const char *` in place of `c_str()`:

```
inline const char *operator* (const string &s)
{
    return s.c_str();
}
```

Example

```
string name = "Bartholomew";
log("module name is %s\n", *name);
```

Finally, two convenience functions are provided to construct a string using printf or vprintf semantics:

```
string str (const char *fmt, ...)
string vstr (const char *fmt, va_list args)
```

Example

```
string Node::description () const
{
     return str("Node(%d,%d) [%d penguins]", m_x, m_y, m_numPenguins);
}
```

3.2 istrcaststream

There are a number of unfortunate problems with input streams in the standard library. The `istrcaststream` class provides a wrapper around either a string (by internally creating an `istringstream`) or a generic `istream` to address these problems:

```
istrcaststream (std::istream &is)
istrcaststream (const string &s)
```

The specific issues addressed by `istrcaststream` are as follows:

1. By default, input streams only recognize integers in decimal format. `istrcaststream` sets the stream flags to also recognized hexadecimal ('0x' prefix) or octal ('0' prefix) formats.

2. By default, the `signed char` and `unsigned char` types are treated by streams as single characters. This makes sense in the context of text processing, but has the unfortunate side effect that the `int8` and `uint8` types

are also treated as characters. This can lead to surprising results when, say, the type of a structure member that is used in streaming operations is changed from `int16` to `int8`. `istrcaststream` (and `ostrcaststream`; see next section) therefore adopts the principle of least surprise by streaming these types as integers rather than characters; characters can still be streamed using the `char` type.

3. An input stream can read booleans as either "0"/"1" or "true"/"false", but not both. `istrcaststream` will always recognize either representation.

4. According to a post at http://www.tech-archive.net/Archive/VC/microsoft.public.vc.stl/2007-01/msg00035.html, the default thousands separator for the "C++" locale is supposed to be ','. Unfortunately, this prevents input streams from reading comma-separated integers. `istrcaststream` therefore sets the locale to "C", which clears the thousands separator and allows integers in the input stream to be comma-separated.

5. An input stream's `eof()` function will not return true until *after* a read has failed. `istrcaststream` corrects this behavior with an `eof()` function that returns true when there is no more data to read.

6. Similarly, an input stream's `peek()` function behaves unpredictably at the end of the input: it sets the fail bit when called twice, but not when called once. `isrcaststream` corrects this behavior with a `peek()` function that always succeeds without setting the fail bit, returning the `eof` character when there is no more data to read.

Once created, an `istrcaststream` can be used in exactly the same manner as an `istream`, but note that it does not derive from `istream`.

Example

```
string s = "37 true";
int n;
bool b;
istrcaststream iss(s);
iss >> n >> b;
```

`istrcaststream` provides the following member functions:

`inline operator std::istream & ()`

Returns a reference to the underlying stream object. This allows an `istrcaststream` object to be passed as an argument to a function expecting an `istream`. Note, however, that since the *underlying* `istream` object is returned and `istrcaststream` itself does not derive from `istream`, some of the properties of the `istrcaststream` will be lost. In particular, the stream will read the `int8` and `uint8` types as characters, it will not recognize both forms of booleans, and `eof()` will misbehave. It *will* still recognize hexadecimal and octal integers as well as comma-separated integers, since these behaviors are implemented by modifying the `istream` that is wrapped within the `istrcaststream`.

inline std::istream *operator-> ()

The pointer operator returns a pointer to the underlying `istream`, providing convenient access to all of its member functions.

Example

```
istrcaststream iss("I want my telemundo");
…
std::pos_type pos = iss->tellg();
```

inline bool operator! () const

Delegates to `istream::operator!()`. Returns true if the stream is bad.

inline operator void * () const

Delegates to `istream::operator void * ()`. Returns NULL if the stream is bad.

inline bool eof () const

Returns true if there is no more data available from the stream.

inline int peek () const

Returns the next character, or `istringstream::traits_type::eof()` if there is no more data available from the stream. Does not modify the stream's state.

void skipws ()

Skip past any whitespace (defined as characters <= ' ') in the input stream. This is useful before calling `peek()`.

Example

```
iss.skipws();
bool negative = (iss->peek() == '-');
```

3.3 ostrcaststream

As with input streams, there are a number of unfortunate problems with output streams in the standard library. The `ostrcaststream` class provides a wrapper around either a string (by internally creating an `ostringstream`) or a generic `ostream` to address these problems:

ostrcaststream (std::ostream &os)
ostrcaststream ()

The specific issues addressed by `ostrcaststream` are as follows:

1. Again, by default `int8` and `uint8` are treated as characters. They are treated as integers by `ostrcaststream`.

2. By default, an output stream formats booleans as "0"/"1". `ostrcaststream` formats booleans as "true"/"false".

3. By default, an output stream truncates floating-point values. `ostrcaststream` uses enough precision to exactly represent floats and doubles.

Once created, an `ostrcaststream` can be used in exactly the same manner as an `ostream`, but note that it does not derive from `ostream`.

Example

```
ostrcaststream oss;
oss << "PI is close to " << 3.1416 << "\n";
string s = oss.str();
```

`ostrcaststream` provides the following member functions:

`inline operator std::ostream & ()`

Returns a reference to the underlying stream object. This allows an `ostrcaststream` object to be passed as an argument to a function expecting an `ostream`. Note, however, that since the *underlying* `ostream` object is returned and `ostrcaststream` itself does not derive from `ostream`, the stream will write the `int8` and `uint8` types as characters. It will still write booleans as "true"/"false" and use enough precision for floating point numbers, since these behaviors are implemented by modifying the `ostream` that is wrapped within the `ostrcaststream`.

`inline std::istream *operator-> ()`

The pointer operator returns a pointer to the underlying `ostream`, providing convenient access to all of its member functions.

Example

```
ostrcaststream oss;
oss->put('!');
```

`inline bool operator! () const`

Delegates to `ostream::operator!()`. Returns true if the stream is bad.

`inline operator void * () const`

Delegates to `ostream::operator void * ()`. Returns NULL if the stream is bad.

`inline string str () const`

Delegates to `ostringstream::str()`. For use when the `ostrcaststream` wraps an `ostringstream` (i.e. it was created with no constructor arguments). Returns the string that has been constructed.

3.4 String conversion for pointers

In addition to addressing various issues with `istream` and `ostream`, the classes `istrcaststream` and `ostrcaststream` provide specialized support for streaming pointers. `ostrcaststream` formats pointers gdb-style with a type prefix, e.g.

```
(int *) 0x7fffedb2232c
```

When a pointer is streamed in, `istrcaststream` will accept either the format generated by `ostrcaststream` or a raw integer (with no type prefix). In the former case, the type prefix must exactly match the type of the pointer being read; otherwise the stream's fail bit will be set. Note that this discussion does not apply to `char` pointers since these are streamed as strings; if the intention is to stream a `char` pointer directly then it must be converted to/from an `intptr_t`.

3.5 Parsing input streams

One of the limitations of text input streams is that they are primarily designed for reading whitespace-separated data (or, with some heroics, data separated by other statically-defined separators). Thus, for example, it is difficult to parse a vector represented as "`(1,2,3)`" or a complex number represented as "`2+7i`".

The descore library simplifies such parsing tasks by defining the following parsing operator:

```
std::istream &operator>> (std::istream &is,
                          const char *delimstring)
```

This version of the streaming operator extracts the constant string `delimstring` from the input stream. If the stream's `std::ios_base::skipws` flag is set (skip whitespace), then leading whitespace will be ignored in *both* the stream and the delimiter string. Next, as many matching characters as possible are read from the stream, stopping when the entire delimiter has been matched (success), or when the next character in the stream would not match the next character in the delimiter string (failure: the mismatched character is *not* read from the stream), or when there are no more characters to read from the stream (failure). In the two failure cases, the stream's failure bit is set.

Example

```
struct Complex
{
     float real;
     float imag;
};

Complex readComplex (std::istream &is)
{
     Complex c;
     is >> c.real >> "+" >> c.imag >> "i";
     assert(is, "Failed to read complex number from stream");
     return c;
}
```

3.6 Delimited strings

A complication arises for string data with no space between the string and the following delimiter, because the default input streaming operator for strings will consume the delimiter as part of the string. In other cases, it may be necessary to parse a string containing whitespace. For these cases, descore provides the helper class `descore::delimited_string`. This class allows the exact set of delimiters to be specified. A delimited string is read from an input stream as follows:

- Leading whitespace is discarded.
- If the string is quoted, then the string input consists of everything within (but not including) the quotations marks; the set of delimiters is ignored.

- Otherwise, the input is read up to the first top-level delimiter (i.e. the first delimiter not enclosed in parentheses, square brackets or curly braces).
- Trailing whitespace is discarded.

```
delimited_string::delimited_string (const char *delimiters,
                                    bool allowInternalWhitespace = false)
```

Construct a delimited string helper object by supplying the set of delimiters as a zero-terminated string. By default, any whitespace character is a delimiter. If `allowInternalWhitespace` is true, then whitespace characters are not delimiters unless they are explicitly specified in `delimiters`.

```
std::istream &operator>> (std::istream &is,
                          delimited_string &s)
istrcaststream &operator>> (istrcaststream &is,
                            delimited_string &s)
```

Parse a delimited string by discarding leading whitespace and reading characters up to (and not including) the first top-level delimiter, or, if the input string is quoted, by reading all characters within the quotation marks.

```
delimited_string::operator string ()
```

Automatic conversion from a delimited string to the parsed string value.

```
string delimited_string::val
```

Public member variable giving explicit access to the parsed string.

Example

```
// Parse a symbolic expression of the form a=b+c
void parseSum (std::istream &is, string &a, string &b, string &c)
{
     descore::delimited_string da("=");
     descore::delimited_string db("+");
     is >> da >> "=" >> db >> "+" >> c;
     a = da;
     b = db;
}
```

3.7 Conversion to string

The descore library defines a templated `str()` function that can be used to convert an arbitrary type to a string:

```
template <typename T> string str (const T &val)
```

Conversion from a user type T to a string can be defined by overloading either `str()` or the `ostream` streaming operator for that type:

```
std::ostream &operator<< (std::ostream &os, const T &val)
```

If the `str()` function is overloaded for a type, then the streaming operator for that type will not automatically be available, but the type can still be streamed fairly conveniently as a string.

Example

```
struct Complex
{
     float real;
     float imag;
};

template <> string str (const Complex &c)
{
     return str("%f+%fi", c.real, c.imag);
}

…
Complex c = { 1.5, 2.5 };
std::cout << "c = " << str(c) << std::endl;
```

If the output streaming operator is overloaded for a type, then the `str()` function *will* automatically be available because the default implementation delegates to streaming. In defining the streaming operator, `ostrcaststream` should generally be used for streaming data primitives to avoid the issues listed in Section 3.3.

Example

```
struct Complex
{
     float real;
     float imag;
};

std::ostream &operator<< (std::ostream &os, const Complex &c)
{
     ostrcaststream oss(os);
     oss << c.real << '+' << c.imag << 'i';
     return os;
}

…
Complex c = { 1.5, 2.5 };
log("c = %s\n", *str(c));
```

3.8 Conversion from string

The descore library defines a templated `fromString()` function that can be used to assign an arbitary type from a string:

```
template <typename T> void fromString (T &val, const string &s)
```

Conversion from a string to a user type `T` can be defined either by specializing `fromString()` or by overloading the `istream` streaming operator for that type:

```
std::istream &operator>> (std::istream *&is, T &val)
```

Conversion from a string to a data type is thus quite similar to conversion from a data type to a string. The main difference is that while conversion to a string should always succeed, conversion *from* a string fails if the string is not properly formatted. The default implementation of `fromString()` delegates to streaming and throws a `strcast_error` exception (which derives from `descore::runtime_error`) on failure.

A second difference is that if the `fromString()` function is overloaded for a type, then it is slightly more difficult to also use streaming for that type. If the stream happens to be whitespace-delimited and the string representation of the

type contains no whitespace, then a temporary string can be used as in the following example, but in general more work is needed.

Example

```
struct Complex
{
     float real;
     float imag;
};

template<> void fromString (Complex &val, const string &s)
{
     if (sscanf(*s, "%f+%fi", &c.real, &c.imag) != 2)
          throw strcast_error("Failed to convert '%s' to Complex", *s);
}

…
Complex c;
string s;
std::cin >> s;
fromString(c, s);
```

If the input streaming operator is overloaded for a type, then the `fromString()` function *will* automatically be available because the default implementation delegates to streaming. In defining the streaming operator, `istrcaststream` should generally be used for streaming data primitives to avoid the issues listed in Section 3.2. In addition, failure should be indicated by setting the stream's 'fail' bit via:

```
iss->setstate(std::ios_base::failbit);
```

Note that this happens automatically if the parsing operator (Section 3.4) is used to read expected separator characters.

Example

```
struct Complex
{
     float real;
     float imag;
};

std::istream &operator>> (std::istream &is, Complex &c)
{
     // Note: Don't actually need istrcaststream here because we're not
     //       using any of its properties, but this is the general
     //       recommended usage for wrapping an istream.
     istrcaststream iss(is);
     iss >> c.real >> "+" >> c.imag >> "i";
     return is;
}
…
void Transformation::setComplexRotation (const string &s)
{
     fromString(m_rotation, s);
}
```

Two alternate conversion functions are always automatically available. The first is `tryFromString()`, which wraps `fromString()` within a try/catch block and returns a boolean to indicate success rather than throwing a `strcast_error`:

```
template <typename T>
bool tryFromString (T &val, const string &s)
```

The second is `stringTo()`, which returns the desired type directly instead of taking a reference to that type as an output parameter:

```
template <typename T>
T stringTo (const string &s)
```

Note that, because none of the `stringTo()` parameters depend on the template type, the type must be explicitly specified when using this function.

Example

```
void Transformation::setComplexRotation (const string &s)
{
     m_rotation = stringTo<Complex>(s);
}
```

3.9 string_cast

The `string_cast` structure encapsulates the `str()` and `fromString()` functions and can be used as a template parameter for templated classes that need to perform string conversions on their templated types. This allows users of the templated class to override the string conversion behavior if desired.

```
template <typename T> struct string_cast
{
     static string str (const T &val);
     static void fromString (T &val, const string &s);
};
```

Example

```
template <typaname T, typename strcast = string_cast<T> >
class Container
{
     …
     void setFromString (const string &s)
     {
          strcast::fromString(m_value, s);
     }
     string getValue () const
     {
          return strcast::str(m_value);
     }
     T m_value;
};
```

3.10 Conversion of STL containers

Conversions to and from strings are provided for all STL containers as well as `descore::queue` (Section **Error! Reference source not found.**). The following syntax rules are used:

- Container elements are comma-separated.
- Unsorted containers (e.g. lists, vectors) are delimited by square brackets `[ ]`
- Sorted containers (e.g. sets, maps) are delimited by curly braces `{}`
- The key-value relationship in maps is indicated by =>

Container values use their native conversion to/from strings. String values are automatically enclosed in quotation marks if they contain whitespace or any special characters that would make parsing ambiguous. When parsing a string that is *not* enclosed in quotation marks, any whitespace or special characters contained within (possibly nested) parentheses, square brackets or curly braces are ignored.

Examples

```
std::vector<int> x;
fromString(x, “[-2,0,17]”);
string s = str(x);  // s = “[-2,0,17]”

std::set<string> x;
fromString(x, “{c,a,\”b,d\”}”);
string s = str(x);  // s = “{a,\”b,d\”,c}”

std::map<int, std::list<string> > x;
fromString(x, “{3=>[one,two,three],2=>[],1=>[Monday,Tuesday]}”);
string s = str(x);  // s = “{1=>[Monday,Tuesday],2=>[],3=>[one,two,three] }”

std::vector<string> x;
fromString(x, “[(a,b),(c d)]”);  // x[0] = “(a,b)”, x[1] = “(c d)”
string s = str(x);  // s = “[(a,b),(c d)]”
```

4.0 Tracing

```
#include <descore/Trace.hpp>
```

Tracing provides a powerful mechanism for conditional logging. Tracing can be dynamically enabled and disabled based on a combination of a string context and named trace keys. The conditional test associated with a trace statement is fast: in release builds, the overhead of a disabled trace statement is only three instructions.

4.1 Basic tracing

Tracing with printf semantics has the same format as logging:

```
trace(const char *msg, ...)
```

A trace statement outputs its message to the log if tracing is enabled within the current scope. Tracing with streaming semantics is also supported as follows:

```
trace(0) << a << b << …;
```

Examples

```
trace(“found %d records\n”, numRecords);
trace(0) << “found ” << numRecords << “ records\n”;

// Output when tracing is enabled
found 5 records
found 5 records
```

4.2 Trace contexts

The *trace context* is the string returned by the function `getTraceContext()`. Tracing uses a set of macros that internally call this function to obtain the trace context within current scope; a default global implementation of this function is provided that returns “”. Classes may define `getTraceContext()` as a member function so that each class instance has its own trace context, which allows tracing to be dynamically controlled on a per-instance basis. The `getTraceContext()` function should have the following signature:

```
string getTraceContext () const
```

In addition to providing a context string, an object must also pre-compute and provide a `TraceKeys` object. This compact (16-bit) object efficiently encodes the set of traces that are enabled within a context. It can be precomputed using the helper function

```
TraceKeys descore::computeTraceKeys (const string &context)
```

then provided by the following member function:

```
TraceKeys getTraceKeys () const
```

As with `getTraceContext()`, this function is called by the trace macros to obtain the set of enabled traces within the current scope.

When the trace context is not the empty string, all trace output is given the prefix “*context*: ”.

Example

```
class StreamAverage
{
public:
     StreamAverage (const string &name) :
          m_name(name),
          m_sum(0),
          m_count(0),
          m_traceKeys(descore::computeTraceKeys(name))
     {
     }

     void addDataPoint (int data)
     {
          m_count++;
          m_sum += data;
          trace("Adding data point %d (average = %lf)\n", data, average());
     }

     double average () const
     {
          return (double) m_sum / (double) m_count;
     }

     // Returning a const string & is slightly more efficient
     const string &getTraceContext () const
     {
          return m_name;
     }
     TraceKeys getTraceKeys () const
     {
          return m_traceKeys;
     }

private:
     string m_name;
     int64  m_sum;
     int    m_count;
     TraceKeys m_traceKeys;
};

// Sample output with tracing enabled within an object named "test_data"
test_data: Adding data point 5 (average = 2.5)
```

4.3 Specifying the trace context explicitly

There are two instances where it is necessary to explicitly specify the trace context rather than relying on calls to the functions `getTraceContext()` and `getTraceKeys()` within the current scope. First, if an object defines these non-static member functions, then the `trace()` macro cannot be used within static member functions. Instead, the `s_trace()` macro must be used:

```
s_trace(const char *msg, ...)
s_trace(0) << a << b << …
```

This macro is identical to `trace()`, but is evaluated within the global context. Second, in order to evaluate a trace statement within the context of a specific object, the `obj_trace()` macro must be used, which takes either a reference or a pointer to the object as an additional first argument:

```
obj_trace(obj, const char *msg, ...)
obj_trace(obj,0) << a << b << …
```

Example

```
StreamAverage s("scores");
for (int i = 0 ; i < numScores ; i++)
     s.addDataPoint(scores[i]);
obj_trace(s, "final average = %lf\n", s.average());
```

4.4 Modifying the global trace context

The global trace context (i.e. the trace context for global or static functions) defaults to "", but can be explicitly modified using the following function:

```
string descore::setGlobalTraceContext (const string &context,
                        bool recomputeGlobalTraceKeys = true)
```

This function sets the global trace context name and returns the previous name. As with other trace contexts, the global trace context also has an associated precomputed `TraceKeys` object. If the `setGlobalTraceContext` is called with the `recomputeGlobalTraceKeys` set to true (default), then this object is dynamically recomputed. This can be expensive, however, so if the global trace context is frequently changed among a set of strings known in advance, then it is more efficient to precompute each of their `TraceKeys`, call the function with `recomputeGlobalTraceKeys` set to false, then explicly provide the corresponding `TraceKeys` object using the function:

```
TraceKeys descore::setGlobalTraceKeys (TraceKeys keys)
```

This function specificies the set of traces to enable in the global context and returns the previous set of traces.

Example

```
void run_phases ()
{
     descore::setGlobalTraceContext("prep");
     run_prep();

     TraceKeys calcTraceKeys = descore::computeTraceKeys("calc");
     TraceKeys cleanupTraceKeys = descore::computeTraceKeys("cleanup");

     for (int i = 0 ; i < 100 ; i++)
     {
          descore::setGlobalTraceContext("calc", false);
          descore::setGlobalTraceKeys(calcTraceKeys);
          run_calc(i);

          descore::setGlobalTraceContext("cleanup", false);
          descore::setGlobalTraceKeys(cleanupTraceKeys);
          run_cleanup();
     }

     descore::setGlobalTraceContext("");
}
```

4.5 Trace keys

Additional control over trace output can be obtained through the use of explicitly declared *trace keys*. A trace key is declared using the following macro:

```
TraceKey(key_name)
```

Each trace key can be enabled or disabled on a per-context basis. The trace macros described in the previous sections can be used with a trace key as follows:

```
trace(key_name, msg, ...)
trace(key_name) << a << b << …
s_trace(key_name, msg, ...)
s_trace(key_name) << a << b << …
obj_trace(obj, key_name, msg, ...)
obj_trace(obj, key_name) << a << b << …
```

The `TraceKey()` declaration must precede, at file scope, any trace statements that use the key. The macro declares a static key object, so the same key can be used in multiple files either by placing the `TraceKey()` declaration within a shared header file, or by placing a copy of the declaration within each source file.

When tracing with an enabled trace key, every line of the output is prefixed with the string "[*trace_key*]" (following the context prefix, if any). The key name specified in the `TraceKey()` macro can optionally be prefixed with a single underscore to avoid conflicts with existing symbol names or with C++ keywords:

`TraceKey(_key_name)`

This underscore is omitted in the output prefix and is ignored for the purpose of controlling tracing by key name.

Example

```
TraceKey(send);
TraceKey(_return);

class QueryClient
{
public:
     QueryClient (int index) :
          m_index(index),
          m_traceKeys(descore::computeTraceKeys(getTraceContext()))
     {
     }

     static void sendRequest (const request_t &req)
     {
          s_trace(send, "Sending request %s\n", *str(req));
          …
     }

     bool getReply (reply_t &reply)
     {
          if (!m_replyAvailable)
               return false;
          reply = m_reply;
          trace(_return, "Returning data %s\n", *str(reply));
          m_replyAvailable = false;
          return true;
     }

     string getTraceContext () const
     {
          return str("Query%d", m_index);
     }
     TraceKeys getTraceKeys () const
     {
          return m_traceKeys;
     }

private:
     int m_index;
     reply_t   m_reply;
     TraceKeys m_traceKeys;
};

// Sample output
[send] Sending request GET_FILE(7)
Query7: [return] Returning data base=0x29
```

We refer to tracing without a trace key as *anonymous tracing*, and to the corresponding implicitly-defined trace key as the *anonymous trace key*.

4.6 Associating trace keys with log files

By default, all tracing is sent to `descore::LOG_STDOUT`. In addition to providing conditional control over tracing, trace keys can be used to redirect tracing to a different log file. A trace key can be associated with a log file using one of the following functions.

```
void descore::setTraceLogFile (descore::LogFile f)
```

Set the log file used for anonymous tracing.

```
void descore::setTraceLogFile (const char *keyname,
                               descore::LogFile f)
```

Set the log file used for any trace key matching the wildcard string `keyname`.

```
void descore::setTraceLogFile (TraceKey key_name,
                               descore::LogFile f)
```

Set the log file used for a specific trace key name (that is, set the log file for all individual `TraceKey` instances having the same name as `key_name`, with any leading underscores removed).

Example

```
TraceKey(errors);
TraceKey(info);

void initialize ()
{
     setTraceLogFile(errors, descore::LOG_STDERR);
     setTraceLogFile(info, openLog("info.log"));
}

void process_input (int num_inputs)
{
     // Output to stdout and the main log file
     trace("processing input\n");

     // Output to stdout, the main log file, and "info.log"
     trace(info, "There are %d inputs\n", num_inputs);

     if (num_inputs > MAX_INPUTS)
     {
          // Output to stderr and the main log file
          trace(errors, "Too many inputs!\n");
     }
}
```

4.7 Determining if tracing is enabled

Three different `is_tracing()` macros are provided to determine if tracing is enabled, each of which can be used with or without a trace key:

```
is_tracing()
is_tracing(key)
s_is_tracing()
s_is_tracing(key)
obj_is_tracing(obj)
obj_is_tracing(obj, key)
```

These macros expand to expressions enclosed within parentheses, so they can be used as though they were ordinary functions. `is_tracing()` checks if tracing is enabled in the current context, `s_is_tracing()` checks if tracing is enabled in the global context, and `obj_is_tracing()` checks if tracing is enabled within

the context of a specified object. If a key is provided then the macros check if that specific key is enabled; otherwise they check if anonymous tracing is enabled.

Example

```
trace(mem, "Received memory request for address %x\n", req.addr);
if (is_tracing(mem) && req.write)
     log("    data = %x\n", req.data);
```

4.8 Combining trace keys

A trace key is conceptually a boolean whose value is true when the key is enabled and false when the key is disabled. Abitrary boolean expressions of trace keys can be formed and provided to the various tracing macros in place of the trace key argument; the value of the expression is obtained by appropriately substituiting true/false for each trace key within the expression, and tracing of the expression is enabled if and only if the expression evaluates to true. Within such a trace key expression, anonymous tracing is represented by the literal 0.

Example

```
TraceKey(packets);
TraceKey(send);
…

// Output if tracing of either 'packets' or 'send' is enabled
trace(packets || send, "Sending %s\n", *str(packet));

// Output if tracing of 'packets' and 'send' are both enabled
trace(packets && send, "Sending %s\n", *str(packet));

// Output if anonymous tracing is enabled, or tracing of 'packets' but not 'send'
trace(0 || (packets && !send), "Sending %s\n", *str(packet));
```

When tracing with an enabled trace expression, the output prefix is constructed based on the set of "contributing" trace keys, defined as trace keys that

1. are enabled,
2. appear in the trace expression,
3. are not contained within a negated subexpression ('!' operator),
4. are not contained within a logical AND expression that evaluates to false.

The output is prefixed with the string "[*trace_keys*]", where *trace_keys* is a comma-separated list of the contributing trace keys, in the order that they appear in the trace expression. The anonymous trace key is represented by the literal 0, and if it is the only contributing trace key or there are no contributing trace keys then the prefix is omitted.

Examples

```
// Anonymous tracing is enabled
TraceKey (_enabled);  // Enabled trace key
TraceKey (_disabled); // Disabled trace key

trace(_enabled || _disabled, "test\n");          // Output: [enabled] test
trace(_enabled && 0, "test\n");                  // Output  [enabled,0] test
trace (_disabled || 0, "test\n");                // Output: test
trace(0 && !(_enabled && _disabled), "test\n");  // Output: test
trace(!_disabled, "test\n");                     // Output: test
```

The set of log files to which a trace is sent is also based on the set of contributing trace keys. If this set contains multiple log files, then multiple copies of the output string will appear in both stdout and the main log file.

4.9 Controlling tracing

Two functions are provided to dynamically enable and disable tracing:

```
void descore::setTrace (const char *context = “”,
                        const char *keyname = NULL,
                        const char *filename = “*”);
void descore::unsetTrace (const char *context = “”,
                          const char *keyname = NULL,
                          const char *filename = “*”);
```

`context` is a wildcard string specifying the context(s) for which tracing should be enabled or disabled. `keyname` is a wildcard string specifying the name(s) of the trace key(s) to be enabled or disabled within matching contexts; if the argument is the empty string or NULL (default) then the function applies to *anonymous tracing* (trace statements with no trace key) within matching contexts. Finally, `filename` is a wildcard string that can be used to further refine the set of trace keys (particularly when the same key is defined in multiple files) by selecting only those keys declared in matching files. Note that the filename argument applies to the files in which the trace keys are declared, not the files in which they are used.

If `keyname` is NULL and `context` is a syntactically valid keyname (possibly with wildcards), then these functions will also enable/disable tracing for the key whose name is specified by `context`.

Modifying the set of enabled traces invalidates all precomputed `TraceKeys` objects, which must be therefore be recomputed before they can be used. A callback interface is used to provide automatic notification that the set of enabled trace has changed:

```
struct descore::ITraceCallback
{
    virtual void notifyTrace () = 0;
};
```

This interface has a constructor that self-registers with the tracing infrastructure when it is constructed, and a destrcutror that self-unregisters when it is destructed. So in particular, classes can simply derived from `ITraceCallback` and their `notifyTrace()` functions will be called automatically.

Example

```
class QueryClient : public descore::ITraceCallback
{
    …
    void notifyTrace ()
    {
        m_traceKeys= descore::computeTraceKeys(getTraceContext());
    }
    …
private:
     TraceKeys m_traceKeys;
};
```

4.10 The TraceContext helper class

The `descore::TraceContext` class is provided to encapsulate a standard trace context implementation. The class stores a string name, which can either be provided to the constructor or accessed using the member functions

```
void setName (const string &name);
const string &getName () const;
```

The class inheris from `descore::ITraceCallback` and stores a precomputed `TraceKeys` object, which is updated when either the class name or the set of enabled traces is modified. The class provides implementations of `getTraceContext()` and `getTraceKeys()`, so trace statements in any class that inherits from `descore::TraceContext` will be evaluated in the appropriate context. Finally, the class also implements `getAssertionContext()`, with the string name used as an assertion context as well.

Example

```
class QueryClient : public descore::TraceContext
{
public:
     QueryClient (int index) :
          m_index(index),
          descore::TraceContext(str("Query%d", index))
     {
     }
     static void sendRequest (const request_t &req);
     bool getReply (reply_t &reply);

private:
     int m_index;
};
```

4.11 Specifying tracing from the command line

A helper function is provided to parse and remove trace directives from the command-line arguments:

```
void descore::parseTraces (int &csz, char *rgsz[]);
```

This function searches the arguments for instances of

```
-trace <specifiers>
```

For each such instance the `<specifiers>` string is parsed and used to set traces, then the two arguments are removed from the list of command-line arguments by subtracting 2 from `csz` and shifting the remaining arguments down within the `rgsz` array.

The `<specifiers>` string is parsed in two stages. First, it is expanded into a list of strings using the following rules:

- Top-level semicolons delimit separate strings
- Any substring contained in curly braces is recursively expanded into a list of strings, then the curly braces are replaced by each string in this list

For example, the string `"1.{a;b}.{x;y};cat"` expands to the list of strings `"1.a.x"`, `"1.a.y"`, `"1.b.x"`, `"1.b.y"`, `"cat"`. This algorithm for string expansion is made available for general use via the helper function

```
std::vector<string>
descore::expandSpecifierString (const string &str);
```

Next, each string within the list is parsed as a specifier of the form

```
context[/keyname][:filename]
```

where `context`, `keyname` and `filename` form arguments to `setTrace()` as described in Section 4.6. The default values are also the same: if `keyname` is omitted then it defaults to "" (anonymous tracing), and if `filename` is omitted then it defaults to "`*`" (specifier applies to all files).

For convenience, if a keyname in the global context is specified then the frontslash may be omitted. Thus, the following command-line specification

```
-trace yellow
```

will enable tracing anonymous tracing for any contexts named "yellow" *and* tracing of the key named "yellow" in the global context.

If either the specifier's context or keyname is prefixed with a minus sign, then the specifier *unsets* any matching traces; `context`, `keyname` and `filename` are passed as arguments to `unsetTrace()`. This allows more complicated trace specifications; for example the specifiers string "`graph/{*;-sort}`" traces all keys within the context "graph" (including anonymous tracing) with the exception of the key "sort".

A specifiers string may also be supplied directly to the function

```
void descore::setTraces (const char *traces)
```

This function parses the `traces` string as though it were supplied on the command line as described above.

Examples

```
// Enable anonymous tracing for the context "prep", or the named trace "prep"
// in the global context
-trace prep

// Enable the named trace "network" in the context named "sim"
-trace sim/network

// Enable both anonymous tracing and named traces starting with "data", except
// for "data_verbose", in any context whose name ends with ".loader"
-trace "*.loader{;/data*;-/data_verbose}"
```

4.12 Listing the available trace keys

In addition to parsing trace directives, `parseTraces()` recognizes the following two command line options that can be used to list the available traces:

```
-showtraces [name]  // summary list of all matching trace key names
-showkeys   [name]  // full list of matching trace keys including filenames
```

If either of these options are encountered, then the trace key information is displayed on stdout and the application exits. `-showtraces` lists only the trace key names, while `-showkeys` lists both the key names and the set of files in which each trace key is declared. `name` is an optional wildcard string that defaults to *.

4.13 Controlling trace output

A global tracer object is responsible for formatting the header string and performing the actual trace output. By default, the header string consists of the context and keyname prefixes, and the trace output is sent to the log. Applications can define their own tracers to customize this behavior by deriving a class from `descore::Tracer` and implementing one or both of the following two virtual functions:

```
void traceHeader (const string &context, const string &keyname)
void vappendTrace (const char *msg, va_list args)
```

`traceHeader()`outputs a header string; it can call the base class implementation to include the default prefixes. `vappendTrace()` outputs a trace string using vararg semantics. These functions can modify and/or filter the trace output in an application-specific manner.

A custom tracer can be installed using the function

```
descore::Tracer *descore::setTracer (descore::Tracer *tracer)
```

This installs the new tracer as the global tracer object and returns a pointer to the previous tracer.

Example

```
struct WizTracer : public descore::Tracer
{
    virtual void traceHeader (const string &context, const string &keyname)
    {
        if (Wiz::tracing)
            log("[WIZ] ");
        descore::Tracer::traceHeader(context, keyname);
    }
    virtual void vappendTrace (const char *msg, va_list args)
    {
        if (Wiz::tracing)
            descore::Tracer::vappendTrace(msg, args);
    }
} g_wizTracer;
…
descore::setTracer(&g_wizTracer);
```

4.14 Unconditional trace output

While the primary purpose of tracing is to provide conditional logging, the formatted header prefix can also be desirable on its own for unconditional output. Three `force_trace()` macros are provided which are identical to their `trace()` counterparts (and produce the same output), the only difference being that the output is unconditional.

```
force_trace(msg, ...)
force_trace(0) << a << b << …
force_trace(key, msg, ...)
force_trace(key) << a << b << …
s_force_trace(msg, ...)
s_force_trace(0) << a << b << …
s_force_trace(key, msg, ...)
s_force_trace(key) << a << b << …
obj_force_trace(obj, msg, ...)
obj_force_trace(obj, 0) << a << b << …
obj_force_trace(obj, key, msg, ...)
obj_force_trace(obj, key) << a << b << …
```

4.15 Summary of macros and functions

```
TraceKey(key)
```

Declare a trace key named “key”. Must appear at file scope.

```
trace(msg, ...)
trace(0) << a << b << …
trace(key, msg, ...)
trace(key) << a << b << …
```

Anonymous or named-key tracing using either printf or streaming semantics in the current trace context.

```
s_trace(msg, ...)
s_trace(0) << a << b << …
s_trace(key, msg, ...)
s_trace(key) << a << b << …
```

Anonymous or named-key tracing using either printf or streaming semantics in the global context.

```
obj_trace(obj, msg, ...)
obj_trace(obj, 0) << a << b << …
obj_trace(obj, key, msg, ...)
obj_trace(obj, key) << a << b << …
```

Anonymous or named-key tracing using either printf or streaming semantics in the context of the specified object.

```
force_trace(msg, ...)
force_trace(0) << a << b << …
force_trace(key, msg, ...)
force_trace(key) << a << b << …
```

Anonymous or named-key unconditional output using either printf or streaming semantics in the current trace context.

```
s_force_trace(msg, ...)
s_force_trace(0) << a << b << …
s_force_trace(key, msg, ...)
s_force_trace(key) << a << b << …
```

Anonymous or named-key unconditional output using either printf or streaming semantics in the global context.

```
obj_force_trace(obj, msg, ...)
obj_force_trace(obj, 0) << a << b << …
obj_force_trace(obj, key, msg, ...)
obj_force_trace(obj, key) << a << b << …
```

Anonymous or named-key unconditional output using either printf or streaming semantics in the context of the specified object.

```
is_tracing()
is_tracing(key)
```

Deremines if anonymous or named-key tracing is enabled in the current trace context.

```
s_is_tracing()
s_is_tracing(key, msg, ...)
```

Determines if anonymous or named-key tracing is enabled in the global context.

```
obj_is_tracing(obj)
obj_is_tracing(obj, key)
```

Determines if anonymous or named-key tracing is enabled in the context of the specified object.

```
void descore::setTrace (const char *context = "",
                        const char *keyname = NULL,
                        const char *filename = "*");
void descore::unsetTrace (const char *context = "",
                          const char *keyname = NULL,
                          const char *filename = "*");
```

Enable or disable tracing for the specified key name (and optionally filename) within the specified context. All arguments are treated as wildcard strings.

```
std::vector<string>
descore::expandSpecifierString (const string &s)
```

Helper function to expand a string into multiple specifiers. Top-level semicolons delimit separate strings; any substring contained within curly braces {} is recursively expanded into a list of strings, then the curly braces are replaced with each string in this list.

```
void descore::setTraces (const char *traces)
```

Enable tracing based on a specifier string. The string is first expanded into multiple specifiers using `expandSpecifierString()`. Each specifier has the format `context[/key][:file]`.

```
void descore::parseTraces (int &csz, char *rgsz[])
```

Parse trace directives from the command line and remove them from the command-line arguments. Also recognizes `-showtraces [name]`, which prints a compact list of all matching trace key names then exits, and `-showkeys [name]`, which prints a full list of all matching trace keys then exits.

```
TraceKeys descore::computeTraceKeys (const string &context)
```

Precompute the set of trace keys that are enabled within the specified context. The returned `TraceKeys` object becomes invalid when the set of enabled traces is modified.

```
string descore::setGlobalTraceContext (const string &context,
                                       bool recomputeGlobalTraceKeys = true)
```

Specify the name of the global context (which defaults to "") and return the previous name. If `recomputeGlobalTraceKeys` is true, dynamically recompute the global `TraceKeys` object. If `recomputeGlobalTraceKeys` is false, then `descore::setGlobalTraceKeys()` must also be called.

```
TraceKeys descore::setGlobalTraceKeys (TraceKeys keys)
```

Specify the set of trace keys that are enabled in the global context. Returns the previous set of trace keys that were enabled in the global context.

```
descore::Tracer *setGlobalTracer (descore::Tracer *tracer)
```

Installs a new global tracer object and returns the previous global tracer.

```
void descore::setTraceLogFile (descore::LogFile f)
void descore::setTraceLogFile (const char *keyname,
                               descore::LogFile f)
void descore::setTraceLogFile (TraceKey key_name,
                               descore::LogFile f)
```

Set the log file associated with anonymous tracing (first version), or all trace keys matching the wildcard string `keyname` (second version), or all trace keys having the same name as the trace key `key_name` (third version).

5.0 Archiving

```
#include <descore/Archive.hpp>
```

The `Archive` class provides support for reading and writing program state from/to a compressed binary archive file.

5.1 Overview

The descore `Archive` class uses the `|` operator as a replacement for the standard `<<` and `>>` operators (which are available, but delegate to the `|` operator). For the most part, this allows the same code to be used for both storing and loading, which simplifies the code and reduces the number of archiving-related bugs. For example, the code

```
void SomeClass::archive (Archive &ar)
{
     if (ar.isLoading())
          ar >> m_count >> m_size >> m_head >> m_tail;
     else
          ar << m_count << m_size << m_head << m_tail;
}
```

can (and should) be replaced by

```
void MyClass::archive (Archive &ar)
{
     ar | m_count | m_size | m_head | m_tail;
}
```

5.2 Archive modes

In addition to the usual “load” and “store” modes, two archive modes are supported that can be useful for debugging. The first of these is "safe store” mode. When an archive is created in safe mode, check bytes are inserted before every single primitive value. These check bytes are validated when the archive is loaded, which detects mismatches between code for saving and loading state.

The second debugging mode is “validate” mode. In this mode, the Archive object acts as though it is storing data, but instead of writing to an archive it reads from an existing archive and sets a flag if the data being stored does not exactly match the data in the existing archive. This can be used to verify that a program’s state matches some known good state in an archive. It can also be used to isolate a difference between two archives that should match by first loading from one archive and then validating against the second.

5.3 Opening and closing an archive

There are two ways to use an Archive object to open an archive. The first is to supply the archive filename and mode to the object’s constructor:

`Archive::Archive (const char *filename, Mode mode)`

where `mode` is one of

```
Archive::Load
Archive::Store
Archive::SafeStore
Archive::Validate
```

The second method is to create the archive object with its default constructor and then separately open the archive using the `open()` member function:

```
void Archive::open (const char *filename, Mode mode)
```

The Archive object will generate an assertion failure if the file cannot be opened, or if `open()` is called when the archive is already open.

An archive is closed automatically when the `Archive` object destructor is called. Alternately, an archive can be closed explicitly using `close()`:

```
void Archive::close ()
```

After calling `close()`, the `Archive` object can be reused by calling `open()`.

Example

```
if (swapState)
{
     Archive ar("checkpoint1.dat", Archive::Store);
     m_userState.archive(ar);
     ar.close();
     ar.open("checkpoint2.dat", Archive::Load);
     m_userState.archive(ar);
}
```

5.4 Archiving data

The recommended prototype for an archive function is

```
void SomeClass::archive (Archive &ar);
```

Any "bag of bits" type (primitive types, enumerations, and classes/unions with the default copy constructor and copy assignment operator) can be archived using the `|` operator. The `|` operator is also provided for `std::string`, `std::pair` and `strbuff`; programmers can specify additional `|` operators for other types. Supported containers can be archived using the `|` operator whenever their constituents can; the supported containers are the descore `queue` and `fifo` containers, and the STL `list`, `vector`, `deque`, `set`, `multiset`, `map` and `multimap` containers.

Example

```
class DataSet
{
public:
     void archive (Archive &ar)
     {
          int version = m_version;
          ar | m_version;
          assert(version == m_version, "Version mismatch");
          ar | m_name | m_clientScores;
     }

private:
     int m_version;
     std::string m_name;
     std::map<std::string, std::set<int> > m_clientScores;
};
```

The << and >> operators can also be used to archive data; each simply delegates to `|`. An assertion failure is generated if << is used for a loading archive or if >> is

used for a storing archive. The direction of the archive can be tested using the `isLoading()` function:

```
bool Archive::isLoading() const
```

This allows special case code to be written for loading and/or storing when such code is necessary.

An arbitrary block of bytes can be archived using `archiveData()`:

```
void Archive::archiveData (void *buff, int size)
```

`buff` is a pointer to the data to archive (when storing), or a pointer to the buffer which receives the data (when loading). `size` is the total number of bytes to archive.

An array can be archived using the templated `archiveArray()` function:

```
template <class T> void Archive::archiveArray (T *data, int size)
```

`data` is a pointer to the data to archive (when storing), or a pointer to the buffer which receives the data (when loading). `size` is the total number of elements in the array. When storing, `size` is written to the archive. When loading, `size` is validated against the size in the archive, and an assertion failure is generated if the two do not match; this validation is the advantage of using `archiveArray()`. Each element is then archived individually using the `|` operator (so in particular this operator must be defined for the array type).

Example

```
class Switch
{
public:
     void archive (Archive &ar)
     {
          ar | m_configSize;
          if (ar.isLoading())
               m_configData = new byte[m_configSize];
          ar.archiveData(m_configData, m_configSize);
          ar.archiveArray(m_state, NUM_CLIENTS);
     }

private:
     int m_configSize;
     byte *m_configData;
     int m_state[NUM_CLIENTS];
};
```

5.5 Archiving constant strings

The `|` operator is also defined for `const char *`, and has slightly different semantics for storing and loading. For storing, the constant string is simply converted to a `string` and then archived, so it can later be read from the archive as a `string`. For loading, the constant string is used for validation: the next piece of data in the archive must be a string which matches the supplied constant string. An assertion failure is generated if the string in the archive does not match the supplied constant string.

Example

```
void archiveShape (Archive &ar, Shape *&shape)
{
     ar | "shape"; // Validation
     if (!isLoading())
          ar | (shape->isSquare() ? "square" : "circle");
     else
     {
          string type;
          ar | type;
          shape = (type == "square") ? new Square : new Circle;
     }
     shape->archive(ar);
}
```

5.6 Detecting end-of-archive

A conversion to bool is provided to determine whether or not additional data can be archived:

`Archive::operator bool ()`

If the archive is invalid then the conversion returns `false`. For a valid storing archive the conversion always returns `true`. Finally, for a valid loading archive the conversion returns `true` if there is additional data to be read from the archive, and `false` otherwise.

Example

```
Archive ar("lots_of_strings.dat", Archive::Load);
while (ar)
{
     string s;
     ar | s;
     log("%s\n", *s);
}
ar.close();
```

5.7 User types

New primitives can be declared using the macro

`DECLARE_ARCHIVE_PRIMITIVE(type)`

This defines the | operator for the type, which will archive the type as a "bag of bits" (so in particular this macro should *not* be used for types that contain pointers or virtual functions). For the most part this macro is unnecessary: it is only required if a type should be archived as raw bits even though it has a custom copy constructor or assignment operator (either of which disables the default | operator for that type). For example:

```
struct IndexedValue
{
     // Preserve the index when assigning from another IndexedValue
     IndexedValue &operator= (const IndexedValue &rhs)
     {
          value = rhs.value;
          return *this;
     }

     int index;
     int value;
};

// Required because of the custom copy assignment operator
DECLARE_ARCHIVE_PRIMITIVE(Point)
```

A version of this macro is provided for templated types:

```
template <class T>
struct IndexedValue
{
     IndexedValue &operator= (const IndexedValue &rhs)
     {
          value = rhs.value;
          return *this;
     }

     int index;
     T   value;
};

template <class T>
DECLARE_TEMPLATE_ARCHIVE_PRIMITIVE(Point<T>)
```

Classes which implement an `archive()` function can be declared as archivable using

`DECLARE_ARCHIVE_OBJECT(type)`

This allows objects of the specified type to be archived using the `|` operator, which simply calls the object's `archive()` function. Again, a version of this macro is provided for template types:

```
template <class T>
DECLARE_TEMPLATE_ARCHIVE_OBJECT(MyObject);
```

Finally, the `|` operator can be defined for arbitrary additional types using the following declaration:

```
template <>
inline Archive &operator| (Archive &ar, Type &value)
{
     ...
     return ar;
}
```

6.0 Parameters

```
#include <descore/Parameter.hpp>
```

The `Parameter` class allows global values to be set from the command line, accessed by name, and associated with help strings and default values.

6.1 Declaring parameters

A parameter can be declared using the macro

```
Parameter(<type>, <name>, <default> [, <description> [, <value>]* ])
```

This declares a parameter object named *<name>* with data type *<type>* and default value *<default>*. The default value can be specified either as a literal value (if applicable) or as a string, which will be converted to the appropriate type. The optional *<description>* argument is a text string describing the parameter. The remaining (and also optional) arguments list the valid values for the parameter; if no values are specified then there is no restriction on the parameter's value.

Any data type can be used that (1) can be converted to/from a string, and (2) has an equality comparison operator. This includes scalar types and STL containers. For convenience, the following type-specific macros are defined in `Parameter.hpp` for declaring scalar parameter types:

```
IntParameter   (name, default, ...)
UintParameter  (name, default, ...)
Int64Parameter (name, default, ...)
Uint64Parameter(name, default, ...)
StringParameter(name, default, ...)
BoolParameter  (name, default, ...)
FloatParameter (name, default, ...)
DoubleParameter(name, default, ...)
```

These macros are equivalent to `Parameter()` with the appropriate *<type>*.

Parameters declared in header files must be contained within a singleton object as shown in the following example:

```
// ModelParams.hpp:

struct ModelParams
{
     IntParameter       (NumCompanies,20);
     BoolParameter      (UseSimpleModel,     false,
                         "If true, ignore all second order Lippert-Xu effects");
     StringParameter    (Continent,          "North America",
                         "Select the continent on which to run the model",
                         "North America", "Asia", "Europe");

     Parameter(std::set<string>, ValidColours, "{red, green, blue}");
};
extern ModelParams modelParams;

// ModelParams.cpp:

ModelParams modelParams;
```

This example also illustrates the use of the optional *description* and *value* macro arguments. Parameters declared in a source file can be declared anywhere at file scope.

6.2 Parameter descriptions

The text string describing a parameter (the *description* argument to the parameter macros) is used to print help information for the parameter. The string is reformatted upon output to fit the terminal with a left indent of 4 spaces. Two special characters are recognized within the help string: a carriage return ('\n') forces a carriage return in the output, and a tab character ('\t') sets a new temporary indent. That is, if the tab character appears in column *N* of the output, then every time the output word-wraps at the terminal width, a left indent of *N* spaces will be used for the following line. The left indent is reset to 4 spaces by the next explicit carriage return ('\n'). White space is ignored unless it precedes a tab character, in which case it can be used to set the temporary indent.

Example

```
StringParameter(OnChipNetwork,         "Mesh",
    "Specifies the on-chip network model.  The on-chip network is used "
    "to connect the flex tiles, the HTIS tiles, and the channel clients "
    "on the ASIC.  The same on-chip network model will be used on every "
    "ASIC.\n\n"
    " 'Mesh':     \tA mesh connects tiles and a ring connects the mesh "
                   "to the torus ports.  The ring is connected to the "
                   "edges of the mesh.\n"
    " 'Crossbar': \tA single, large crossbar connects all on-chip "
                   "network ports.  The crossbar latency is determined "
                   "by the Parameter NetRouterHopLatency.",
    "Mesh", "Crossbar");
```

Help information output:

```
OnChipNetwork                  string          Mesh
    (Mesh, Crossbar)
    Specifies the on-chip network model.  The on-chip network is used to
    connect the flex tiles, the HTIS tiles, and the channel clients on the
    ASIC.  The same on-chip network model will be used on every ASIC.

     'Mesh':     A mesh connects tiles and a ring connects the mesh to the
                 torus ports.  The ring is connected to the edges of the mesh.
     'Crossbar': A single, large crossbar connects all on-chip network ports.
                 The crossbar latency is determined by the Parameter
                 NetRouterHopLatency.
```

6.3 Enumeration parameters

An enumeration parameter is declared using the following macro:

`EnumParameter(name, default, description, value*)`

This macro both declares a parameter with the specified name, and creates an enumeration type with the specified values. The name of the enumeration type is obtained by prepending 'E' to the parameter name. For example, we can rewrite the above example using an enumeration parameter (and a shorter description) as follows:

```
struct NetParams
{
    EnumParameter(OnChipNetwork, Mesh, "Specifies the on-chip network model",
                  Mesh, Crossbar);
};
```

The `OnChipNetwork` parameter has type `NetParams::EOnChipNetwork`, which has values `NetParams::Mesh` and `NetParams::Crossbar`. Note that for enumeration parameters, all macro parameters are mandatory.

The enumeration definition can be separated from the parameter by using a descore enumeration type (Section 10.0) with the regular `Parameter` macro.

6.4 Accessing parameters

For the most part, parameters behave exactly like the types they contain. They can be assigned to directly, and they will be automatically cast to the appropriate type. They can safely be accessed at static initialization time. They can also be treated as pointers to the contained type; to explicitly cast a parameter to the contained type, which is necessary if the compiler cannot deduce the type cast, use the dereference operator, e.g.:

```
log("Number of companies: %d\n", *modelParams.NumCompanies);
```

Additionally, if a parameter has member functions/variables, they can be accessed using the pointer dereference operator, e.g.:

```
log("Continent = %s\n", modelParams.Continent->c_str());
```

A parameter's valid values (or *options*), as defined by the parameter declaration macro, can be accessed using two functions that return the number of options and the i^{th} option respectively:

```
int numOptions () const
const T &getOption (int i) const
```

Example

```
for (int i = 0 ; i < modelParams.Continent.numOptions() ; i++)
{
     string continent = modelParams.Continent.getOption(i);
     …
```

A parameter's default value can be accessed using the functions:

```
T getDefault () const
void setDefault (const T &)
```

The `Parameter` class provides static functions to access parameters by name:

```
bool Parameter::setValueByString (string name, string val,
                                  bool logerrors = true)
```

Sets the value of the parameter named by *name* to the value encoded in the string *val*. Returns true if successful. If an error occurs (i.e. if there is no parameter of that name, or if the string value cannot be parsed, or if the value is not one of the permitted values) then an error message is logged (unless `logerrors` is false) and the function returns false.

```
template <typename T>
bool Parameter::setValue (string name, T val,
                          bool logerrors = true)
```

Same as above, but accepts a typed value instead of its string representation.

```
string Parameter::getValueAsString (string name)
```

Returns a string representation of the parameter's value. If there is no parameter of that name then an error message is logged and the function returns "".

```
template <typename T>
T Parameter::getValue (string name)
```

Same as above, but returns a typed value instead of its string representation.

6.5 Static helper functions

The `Parameter` class provides several additional static helper functions. Most of these helper functions take "include" and "exclude" specifier strings as arguments that are expanded into sets of wildcard strings, as described in Section 4.11, and then used to define the set of matching parameters. A parameter matches a wildcard pattern if some subset of its hierarchical name matches the pattern. For example, the parameter "machine.size" matches a pattern if one of "machine", "size" or "machine.size" matches the pattern. A parameter is included in an operation if it matches one of the include patterns but none of the exclude patterns.

```
void Parameter::help (string include = "*", string exclude = "")
```

Prints help information for all matching parameters. The first line of the help information for each parameter contains the parameter name, type and default value, formatted so that if the help information for multiple parameters is displayed then the corresponding fields will line up. The next line lists the valid values for the parameter, if any were specified. Finally, the formatted parameter description is displayed, if a description was specified.

```
void Parameter::archiveParameters (Archive &ar,
                                   string include = "*",
                                   string exclude = "")
```

Archives parameter values to/from an open archive. When storing to an archive, the include/exclude strings specify the set of parameters to archive. When loading from an archive, these specifier strings are ignored.

```
void Parameter::resetParameters (string include = "*",
                                 string exclude = "")
```

Resets all matching parameters to their default values, then re-applies any command-line overrides resulting from calls to `parseCommandLine()`.

```
typedef std::map<string, Parameter *> ParamMap;
const ParamMap &Parameter::getParameters ()
```

Returns the set of all parameters as a map from name to `Parameter` object.

```
ParamMap Parameter::getParameters (string include,
                                   string exclude = "*")
```

Returns the set of all matching parameters as a non-const map from name to `Parameter` object.

```
Parameter *Parameter::findParameter (string name)
```

Returns a pointer to the parameter with the specified name, or NULL If no such parameter exists.

```
void Parameter::checkpointParameters (string include = "*",
                                      string exclude = "")
```

Create a snapshot of the values of all matching parameters that can be restored by calling `Parameter::restoreParametersFromCheckpoint()`.

```
void Parameter::restoreParametersFromCheckpoint
                      (string include = "*", string exclude = "")
```

Restore all matching parameters to their snapshot values.

6.6 Setting indexed parameters

If a parameter's data type supports the indexing operator (for example if it is an `std::vector` or an `std::map`), then the string name supplied to `setValueByString()` can include an index, in which case only the specified portion of the parameter will be set. If the data type supports multiple levels of indexing (for example a vector of vectors), then the string name can also include multiple indices. Each index is validated; if an index is out of range then an error message to this effect will be displayed and `setValueByString()` will return `false`. For an `std::map`, all index values are considered to be in range, so `setValueByString()` can be used to add elements to the map.

Example

```
typedef std::map<string, float> WeightMap;
Parameter(std::vector<float>, ScaleFactors, "[1.0, 1.5, 0.6]");
Parameter(WeightMap, ComponentWeights, "{axle=>7.2, tire=>6.8}");
Parameter(std::vector<std::vector<int> >, Matrix, "[[1, 0], [0, 1]]");
…
Parameter::setValueByString("ScaleFactors[1]", "1.7");
Parameter::setValueByString("ComponentWeights[axle]", "7.4");
Parameter::setValueByString("ComponentWeights[battery]", "3.7");
Parameter::setValueByString("Matrix[0][1]", "-1");
```

6.7 Parameter modification callbacks

Callback functions can be registered with a parameter that are invoked any time its value changes. The following member function can be used to add a callback:

```
void Parameter::addCallback (IParameterChangeCallback *callback)
```

The `IParameterChangeCallback` interface defines a single virtual function:

```
struct IParameterChangeCallback
{
    virtual void notifyChange (Parameter *param) = 0;
};
```

Whenever a parameter's value changes, it invokes each of the callbacks that have been registered, passing `this` as the `param` argument.

Example

```
struct LoggerCallback : public IParameterChangeCallback
{
    virtual void notifyChange (Parameter *param)
    {
        log("%s changed to %s\n", *param->getName(), *param->getValueAsString());
    }
} parameterLogger;
…
Matrix.addCallback(&parameterLogger);
```

6.8 Determining if a parameter has been modified

Each parameter contains a 'modified' flag indicating whether or not the parameter's value has been set explicitly. The following member function provides access to this flag:

```
bool Parameter::modified () const
```

When a parameter is constructed, it is set to its default value and the 'modified' flag is initialized to false. If the parameter is subsequently set via assignment or

`setValueByString()`, then the 'modified' flag is set to true (even if the assigned value is the same as the previous value).

6.9 Parameter groups

Parameters can be placed within a parameter group by declaring them within a structure and then instantiating the structure with the `ParameterGroup` macro:

```
ParameterGroup(type, name [, "group_name"])
```

Within the C++ code, this is logically equivalent to the declaration

```
type name;
```

and in particular parameters within the structure are accessed as `name.parameter`. The macro has the effect of placing these parameters within a named parameter group. If only two arguments are provided to `ParameterGroup()`, then the group name is simply "name" (i.e. the stringification of the second argument). Alternately, the group name can be explicitly specified using the optional third string argument.

Placing parameters within a group automatically prefixes their string names with "group_name.". It does not change their behavior when they are accessed directly from C++ code.

The '`static`' and '`extern`' modifiers can precede the `ParameterGroup()` macro and they will have the usual meaning. So, for example, a parameter group declared as static within a source file will not be visible outside of that source file, and a parameter group declared in a header file should be declared 'extern'. In the later case, the actual declaration of the parameter group should use the macro `DeclareParameterGroup(name)`, e.g.:

```
extern ParameterGroup(NetworkParametres, netParams, "net");
DeclareParameterGroup(netParams);
```

To declare a parameter within a header file only, declare it as a singleton using the `SingletonParameterGroup()` macro:

```
SingletonParameterGroup(type, name [, "group_name"])
```

With this declaration the parameter group is created as a singleton rather than an object (Section **Error! Reference source not found.**). Parameters within the group are therefore accessed as `name->parameter` (using the pointer to member operator).

Parameter groups can be nested, as illustrated in the following example:

```
struct ArbiterParams
{
    BoolParameter(RoundRobin, true);
    BoolParameter(RollingUpdate, true);
};

struct RouterParams
{
    IntParameter(InputQueueDepth, 4);
    IntParameter(Latency, 4);

    ParameterGroup(ArbiterParams, arb);
};

extern ParameterGroup(RouterParams, routerParams, "router");
```

This example defines four parameters whose C++ names and string names are:

`routerParams.InputQueueDepth`	("router.InputQueueDepth")
`routerParams.Latency`	("router.Latency")
`routerParams.arb.RoundRobin`	("router.arb.RoundRobin")
`routerParams.arb.RollingUpdate`	("router.arb.RollingUpdate")

6.10 Parameter group arrays

Arrays of parameter groups are supported via the `ParameterGroupArray` macro:

```
ParameterGroupArray(type, name [, "group_name"] [, size])
```

This macro has the same format as `ParameterGroup`, but takes an additional optional argument specifying the array size. When the size is specified, the array is statically sized, and parameters are accessed and named in the natural manner using square brackets for array indices.

Example

```
struct ArbiterParams
{
    BoolParameter(RoundRobin, true);
    BoolParameter(RollingUpdate, true);
};

struct RouterParams
{
    IntParameter(InputQueueDepth, 4);
    IntParameter(Latency, 4);

    ParameterGroupArray(ArbiterParams, arb, 2);
};

extern ParameterGroup(RouterParams, routerParams, "router");
```

This example, similar to the previous one, defines six parameters whose C++ names and string names are:

`routerParams.InputQueueDepth`	("router.InputQueueDepth")
`routerParams.Latency`	("router.Latency")
`routerParams.arb[0].RoundRobin`	("router.arb[0].RoundRobin")
`routerParams.arb[0].RollingUpdate`	("router.arb[0].RollingUpdate")
`routerParams.arb[1].RoundRobin`	("router.arb[1].RoundRobin")
`routerParams.arb[1].RollingUpdate`	("router.arb[1].RollingUpdate")

When the size is omitted, the array is dynamically sized, and is automatically extended when parameters within the array are set by their string name. This is primarily intended for parameters set from a configuration file (Section 6.11), but also works for parameters set from the command line (Section 6.12) or using the function `Parameter::setValueByString`. The array can be extended by at most one element at a time. If it has size N and a parameter within group [N] is written by name then group [N] will be created (using default values for all of the parameters it contains) and the new array size will be N+1. If it has size N and a parameter within group [M] is written with $M > N$ then an error is generated and the array size is unchanged.

The size of a dynamic parameter group array can be retrieved using the `size()` member function, which returns an `int`.

Example

```
struct Student
{
     StringParameter(name, “”);
     IntParameter(age, 0);
};

struct Classroom
{
     IntParameter(grade, 0);
     ParameterGroupArray(Student, students);
};

extern ParameterGroup(Classroom, classroom);

<initialize parameters from command line or config file>

log(“Students in grade %d:\n”, *classroom.grade);
for (int i = 0 ; i < classroom.students.size() ; i++)
     log(“%s (age %d)\n”, **classroom.students[i].name, *classroom.students[i].age);
```

6.11 Setting parameters from a configuration file

A helper function is provided to parse parameter settings from a file:

```
void Parameter::parseFile (const string &filename,
                           Parameter::IParseCallback *callback = NULL);
```

This function parses parameter settings from a text file, with one setting per line, ignoring comments (which begin with `//`) and whitespace. Parameter settings within the file must be of the form:

```
<parameter> = <value>
```

Parameter settings can be hierarchical: parameters within a group can be set via

```
<group_name>
{
    parameter settings
}
```

For example,

```
router
{
    Latency = 5
    queues
    {
        InputDepth = 8
        OutputDepth = 6
    }
}
```

is the same as

```
router.Latency = 5
router.queues.InputDepth = 8
router.queues.OutputDepth = 6
```

Parameter group arrays can be specified in one of two ways. Each array element can be specified explicitly by appending the index (enclosed in square brackets) to the group name:

```
classroom.students[0]
{
     name = Joe
     age = 50
}
```

Alternately, the entire array can be specified at once by enclosing the array within square brackets, e.g.:

```
classroom.students
[
    {
        name = Joe
        age = 50
    }
    {
        name = Ken
        age = 39
    }
]
```

The optional callback object passed to `Parameter::parseFile()` is used to parse unrecognized lines. The interface has the following definition:

```
struct IParseCallback
{
    virtual bool parseLine (const char *line);
};
```

If the `parseFile()` function is unable to parse one of the lines in the file, then it passes the line to the `parseLine()` callback function, which should return `true` on success and `false` on failure. If the callback function fails, or if no callback object is provided, then an assertion failure is generated.

6.12 Setting parameters from the command line

A helper function is provided to parse parameter settings from the command line:

```
void Parameter::parseCommandLine (int &csz, char *rgsz[])
```

This function should be called with the standard `csz`, `rgsz` arguments passed to `main()`. It parses any parameter settings and modifies `csz`, `rgsz` in place by removing these settings, so that subsequent parsing of the command line does not need to deal with parameters.

Parameters settings on the command line are of the form:

```
-<parameter>=<value>
```

where *<parameter>* is the string name of the parameter. If the parameter name is not recognized then the function assumes the argument is not a parameter setting and ignores it. To force the name to be treated as a parameter, one of the following two forms can be used:

```
-setparam <parameter>=<value>
-tryparam <parameter>=<value>
```

For `-setparam`, an unrecognized parameter generates an assertion failure. For `-tryparam`, an unrecognized parameter is silently ignored and the two arguments are removed from the command line. In all cases, an assertion failure is generated if a recognized parameter cannot be set to the specified value.

For the special case of setting a Boolean parameter to true, the value can be omitted, so the following two settings are equivalent:

```
-<bool_parameter>
-<bool_parameter>=true
```

The `parseCommandLine()` function also recognizes (and removes from the command line) the following options:

```
-quiet                // alias for '-log.quiet=true'
-loadparams <file>    // calls Parameter::parseFile to parse the specified file
```

Example

```
my_test –NumTests=10 -config.Timeout=100 -EnableLogOutput
```

6.13 Automatically providing parameter help

The `parseCommandLine()` function recognizes (and removes from the command line) the following options, allowing users to examine the set of available parameters:

```
-showparams [pattern] // calls Parameter::help() and exits
-showconfig [pattern] // summarizes parameters in hierarchical config-file format
```

The optional *pattern* argument is a wildcard string that can be used to filter the set of parameters for which help information is displayed using the matching rules described in Section 6.5.

6.14 Generating configuration files

A subset of the current parameter settings can be saved to a file using the function

```
void Parameter::generateFile (const string &filename,
                              const string &include = "*",
                              const string *exclude = "")
```

This function creates a file using the specified filename and stores the settings for for all parameters that are in the include list but not the exclude list, using the matching rules described in Section 6.5. The generated file can be subsequently parsed by calling `Parameter::parseFile()`.

6.15 Creating new parameter types

There are three steps involved in creating a parameter with a custom type.

1. [Optional] Define a parameter macro for the type.

The parameter macro for a type `MyType` should be defined as follows:

```
#define MyTypeParameter(name, default, ...) \
    Parameter(MyType, name, default, ##__VA_ARGS__)
```

There are a few different ways to specify parameter values for custom parameter types. First, they can be specified as `MyType(<constructor args>)`. Second, they can be specified as strings that can be converted to the type. Both of these approaches are illustrated in the following example:

```
struct vec2
{
     vec2 () {}
     vec2 (float _a, float _b) : a(_a), b(_b) {}
     float a, b;
};
```

```
#define Vec2Parameter(name, default, ...) \
     Parameter(vec2, name, default, ##__VA_ARGS__)
…
Vec2Parameter (JuliaC, “(-.25, -.25)”, "C value for Julia set",
                        “(-.25,-.25)”, “(0,0)”, “(.25,.25)”);
Vec2Parameter (Origin, vec2(1.5, 1.5), “Default origin”,
                       vec2(0,0), vec2(1,1), vec2(1.5, 1.5));
```

The third approach is to alter the parameter macro definition by adding the type name before `default` as follows:

```
#define Vec2Parameter(name, default, ...) \
     Parameter(vec2, name, vec2 default __VA_ARGS__)
```

This allows the default value to be specified more concisely as constructor arguments with the type name omitted:

```
Vec2Parameter (JuliaC, (-.25, -.25), "C value for Julia set");
```

Defining a new macro is optional; the `Parameter` macro can also be used directly with an arbitrary type as follows:

```
Parameter(vec2, JuliaC,“(-.25, -.25)”, "C value for Julia set",
                        “(-.25,-.25)”, “(0,0)”, “(.25,.25)”);
Parameter(vec2, Origin, vec2(1.5, 1.5), “Default origin”,
                        vec2(0,0), vec2(1,1), vec2(1.5, 1.5));
```

2. Define the string conversion functions.

Ensure that both conversion to string and conversion from string are defined for the type as described in Sections 3.7 and 3.8.

3. Define the equality comparison operator.

All types used for parameters are required to define the `==` operator, which is used to validate that the value being assigned to a parameter is one of the permitted values.

Example

The following functions satisfy the requirements for the `vec2` type defined above:

```
std::ostream &operator<< (std::ostream &os, const vec2 &val)
{
     ostrcaststream oss(os);
     oss << “(“ << val.a << “,’ << val.b << “)”;
     return os;
}
std::istream &operator>> (std::istream &is, vec &val)
{
     is >> “(“ >> val.a >> “,’ >> val.b >> “)”;
     return is;
}
bool operator== (const vec2 &lhs, const vec2 &rhs)
{
     return (lhs.a == rhs.a) && (lhs.b == rhs.b);
}
```

7.0 Threads

```
#include <descore/Thread.hpp>
```

The descore library is thread-safe, and it provides a set of primitives for thread creation, inter-thread synchronization, and thread-local storage.

7.1 Creating threads

A descore thread is encapsulated within a `descore::Thread` object. The life cycle of a thread consists of four stages:

1. **Object creation**. A `descore::Thread` object has only a default constructor, so in particular it is possible to create an array of threads.

2. **Object initialization**. Currently the only supported initialization is to optionally specify the stack size by calling `setStackSize()`.

3. **Thread creation.** The actual thread is created and starts running when one of the `start()` functions is called with a function pointer and arguments.

4. **Thread termination.** The thread runs to completion, and the parent thread is responsible for calling `wait()` at some point to wait for thread termination and reclaim thread resources.

The following member functions of `descore::Thread` are used to initialize, create and finalize threads.

`void setStackSize (int sizeInBytes)`

Specifies the desired stack size (measured in bytes) for the thread.

`void start (function [,args]*)`

Creates a thread which calls the supplied function as soon as it starts. When the function returns, the thread exits. The function must not have a return value, and may have up to 8 arguments. The arguments supplied to `start()` will be passed to the function, and must be compatible with its signature. `function` must be a pointer to a global or static function.

`void start (object, function [,args]*)`

Alternate form of `start()` to be used when the thread entry point is a member function of an object. The first argument is a pointer to the object, and the second argument specifies a pointer to one of the object's member functions. The remaining arguments are passed to the function when it is called.

`bool running () const`

Returns true if a thread is still running, false if it has exited.

`void wait ()`

Waits for a thread to exit. When this function call returns the thread has terminated, and the thread object can be reused to create a new thread. This function is also called from `~descore::Thread()`, so it is not always necessary to call it explicitly.

Examples

```
static void threadFunction (int id)
{
     log("Starting thread %d\n", id);
}
…
// Start a thread using a global function
descore::Thread func_thread;
func_thread.start(&threadFunction, 5);
func_thread.wait();

struct ThreadObject : public descore::Thread
{
     ThreadObject (const string &name) : m_name(name)
     {
          start(this, &ThreadObject::entryPoint);
     }
     void entryPoint ()
     {
          log("Staring thread %s\n", *m_name);
     }
     string m_name;
};
…
// Automatically start a thread when ThreadObject is constructed
TheradObject obj_thread;
```

7.2 Static helper functions

The following static member functions of `descore::Thread` can be called from any thread (including the main thread, unless otherwise indicated).

`void sleep (int milliseconds)`

Suspend the thread for the specified number of milliseconds.

`void exit ()`

Causes the thread (not the entire program) to exit immediately. Internally implemented as an exception, so all stack and thread-local objects will be destructed. Cannot be called from the main thread.

`descore::Thread *self ()`

Returns a pointer to the thread object that encapsulates the current thread. While `descore::Thread` doesn't itself have any members or functions that are useful to the thread, this function may be convenient for threads encapsulated in custom thread objects that inherit from `descore::Thread`. If called from the main thread, returns a pointer to a dummy object.

`bool isMainThread ()`

Returns true if the current thread is the main thread, false otherwise.

7.3 Exceptions

When a thread throws an exception, an error message is logged immediately, but the exception itself is deferred and re-thrown when the thread's `wait()` function is called. Thus, exceptions generated by a thread are converted into exceptions within the context of the parent thread, allowing the parent thread to catch them. This allows computations that potentially generate exceptions to be transparently multithreaded without affecting their exception semantics.

Example

```
DECLARE_EXCEPTION(bad_chunk);

static void doWork (int chunk)
{
     Work *work = getWork(chunk);
     assert_throw(bad_chunk, work != NULL);
     …
}

void doWorkThreaded ()
{
     try
     {
          descore::Thread threads[NUM_CHUNKS];
          for (int i = 0 ; i < NUM_CHUNKS; i++)
               threads[i].start(&doWork, i);
     }
     catch (bad_chunk &e)
     {
          logerr("Bad chunk!\n");
          e.reportAndExit();
     }
}
```

7.4 Synchronization

Three inter-thread synchronization primitives are provided: mutexes, spin locks, and atomic increments/decrements.

The class `descore::Mutex` implements a recursive OS-level mutex that can be acquired and released using the `lock()` and `unlock()` member functions. If a thread fails to aquire a mutex, it will be put to sleep until the mutex becomes available. The same thread can lock the same mutex multiple times, so long as every call to `lock()` is matched by a subsequent call to `unlock()`.

The helper class `descore::ScopedLock` can be used to automatically lock/unlock a mutex in an exception-safe manner (the mutex is automatically unlocked when the `ScopedLock` object is destroyed). The constructor takes a referece to a mutex as its single argument.

Example

```
static set<string> g_names;
descore::Mutex g_nameMutex;

void add_client (Client *c)
{
     g_nameMutex.lock();
     g_names.insert(c->name);
     g_nameMutex.unlock();
     …
}
void delete_client (Client *c)
{
     descore::ScopedLock lock(g_nameMutex);
     g_names.erase(c->name);
}
```

The class `descore::SpinLock` implements a simple spin lock, also acquired and released using `lock()` and `unlock()`. This lock is *not* recursive; if a thread attempts to acquire a spin lock twice it will deadlock. The helper class `descore::ScopedSpinLock` is similar to `descore::ScopedLock`, but takes a reference to a spin lock as its constructor argument.

Example

```
static set<string> g_names;
descore::SpinLock g_nameLock;

void delete_client (Client *c)
{
    descore::ScopedSpinLock lock(g_nameLock);
    g_names.erase(c->name);
}
```

Finally, two functions are provided to atomically increment/decrement shared integer values.

`int atomicIncrement (volatile int &value)`

Atomically increment an integer and return the new value.

`int atomicDecrement (volatile int &value)`

Atomically decrement an integer and return the new value.

7.5 Static thread-local data

The `__thread` keyword may be applied to any global, file-scoped static, function-scoped static or class-static data whose type is trivial (no constructors, destructor or virtual functions).[2] This keyword causes each thread to have its own copy of the data, which is initialized with any specified constant expression when the thread is created.

Example

```
static __thread const char *t_threadName = NULL;

void doWork (const char *threadName)
{
    t_threadName = threadName;
    …
}
```

7.6 Dynamic thread-local data

descore provides a mechanism for thread-local data that can be dynamically declared and can be any type. So, for example, thread-local data can appear as a non-static class member. Thread-local data is declared using the templated class `descore::ThreadLocalData<T>`.

Example

```
struct ThreadedAccumulator
{
    int flags;
    descore::ThreadLocalData<int> sum;
};
```

`descore::ThreadLocalData` can for the most part be used as an object of the underlying type; it supports automatic conversion to the that type as well as assignment from that type. It can also be treated as a pointer and supports the pointer-to-member and dereference operators. Data access is efficient, requiring constant time independent of the number of threads or thread-local objects.

[2] The `__thread` keyword is native to gcc; for MSVC it is #defined to something equivalent.

When a `ThreadLocalData` object is constructed, a single instance of the specified type is constructed and belongs to the current thread. The type's constructor must take 0–4 arguments; the arguments should be supplied to the `ThreadLocalData` object which then uses them to construct the type instance.

When a new thread is created, thread-local instances are constructed for every previously-created `ThreadLocalData` object. In particular, a thread will *only* have access to thread-local instances for `ThreadLocalData` objects that were constructed before the thread was created. The new instances are not copies of the parent-thread instances; they are fresh instances constructed using the same constructor arguments that were supplied to the original `ThreadLocalData` object. When a thread exits, all of its thread-local data instances are destructed.

Examples

```
static descore::ThreadLocalData<string> t_threadName("unnamed thread");

class Processor
{
    …
    // Each thread has its own processor state
    descore::ThreadLocalData<ProcessorState> t_state;
};
```

At any point in time, an operation can be performed across all instances of a thread-local object using the `doAcross()` member function, defined as follows with up to four arguments:

```
void ThreadLocalData<T>::doAcross (void (T::*f) ())
void ThreadLocalData<T>::doAcross (void (T::*f) (A1), A1)
void ThreadLocalData<T>::doAcross (void (T::*f) (A1, A2), A1, A2)
void ThreadLocalData<T>::doAcross (void (T::*f) (A1, A2, A3),
                                   A1, A2, A3)
void ThreadLocalData<T>::doAcross (void (T::*f) (A1, A2, A3, A4),
                                   A1, A2, A3, A4)
```

Example

```
struct ThreadState
{
    int numCompleted;
    int numRemaining;

    void report (const char *what)
    {
        log("%d/%d %s completed\n",
            numCompleted, numCompleted + numRemaining, what);
    }
};

descore::ThreadLocalData<ThreadState> t_state;

void showProgress ()
{
    t_state.doAcross(&ThreadState::report, "thread tasks");
}
```

8.0 Iterators

```
#include <descore/Iterators.hpp>
```

descore provides several helpers for iterating over the elements of an STL container, a string, or a statically-sized array.

8.1 Iterator

The `Iterator` class is a thin wrapper around standard iterators that provides a concise and safe syntax for end-of-iteration tests. It is defined as follows:

```
template <typename Container,
          typename ValueType = typename Container::value_type,
          typename KeyType = typename Container::key_type>
class Iterator
```

The `Iterator` class can be used with the STL `list`, `vector`, `deque`, `set`, `map`, `multiset` and `multimap` containers. It can also be used for strings (`string`, `char *`, `const char *`) and for statically-sized arrays. The first template parameter is the container type. The second (optional) template parameter is the type obtained when an iterator is dereferenced; it cannot be used for strings or for statically-sized arrays. The third (optional) template parameter is only used for `map` and `multimap` iterators, and specifies the type obtained when the `key()` function is called (see Section 8.3). An `Iterator` is initialized from the targeted container, either as a constructor parameter or via assignment, and is initialized to the beginning of the container's elements. The following operators are supported:

`ValueType &Iterator::operator* ()`

The dereference operator, which returns a reference to the current element. Note that this will be a *const* reference for `set` and `multiset` iterators..

`ValueType *Iterator::operator-> ()`

The pointer operator, which returns a pointer to the current element (so that the iterator can be used in the same manner as an ordinary pointer to the element). Note that this will be a *const* pointer for `set` and `multiset` iterators.

`void Iterator::operator++ ()`
`void Iterator::operator++ (int)`

Prefix and postfix increment, each of which advances to the next element. Neither returns the iterator, so there is no semantic difference between the two.

`Iterator::operator bool ()`

Conversion to boolean, which returns true if the iterator is still valid and false if it has been advanced to the end of the container.

Example

```
struct Point
{
     int x;
     int y;
};

std::list<Point> points;
…
for (Iterator<std::list<Point> > it(points) ; it ; it++)
{
     if ((it->x > 0) && (it->y > 0))
          log("Point %s is in the first quadrant\n", *str(*it));
}
```

8.2 ConstIterator

The `ConstIterator` class is identical to the `Iterator` class except that the dereference and pointer operators return a const reference and pointer respectively. As such, it can be used to iterate over const containers.

Example

```
int sum (const std::set<int> &scores)
{
     int ret = 0;
     for (ConstIterator<std::set<int> > it(scores) ; it ; it++)
          ret += *it;
     return ret;
}
```

8.3 Maps and multimaps

When the `Iterator` or `ConstIterator` classes are used with an STL `map` or `multimap`, the dereference and pointer operators return a reference and pointer (respectively) to the value portion of the current (key, value) pair. For `Iterator` these operators return non-const references/pointers; for `ConstIterator` they return const references/pointers. The current key can be retrieved using the function

`const KeyType &key ()`

8.4 Sorted containers

For sorted containers (`set`, `multiset`, `map`, `multimap`), an additional constructor is supported which takes lower and upper bounds as its second and third arguments. Iterators initialized in this manner will iterate over all elements within the range [*lower*, *upper*] (for `map` and `multimap` the keys are in this range).

Example

```
typedef std::map<int, std::string> FunctionMap;
void showFunctions (const FunctionMap &fns, int begin, int end)
{
     for (ConstMapIterator<FunctionMap > it(fns, begin, end) ; it ; it++)
          log("Function %-20s defined at line %d\n", it->c_str(), it.key());
}
```

9.0 Map Iterators

```
#include <descore/MapIterators.hpp>
```

C++11 introduced range-based iteration, which tremendously simplifies iterating over STL containers. There are, however, two shortcomings with respect to iterating over maps. First, there is no clean way to iterate over just the keys of a map or just the values of a map. Second, when iterating over the (key, value) pairs, individual keys and values are accessed somewhat cryptically as "first" and "second". descore provides three helpers to improve the syntax of map iteration.

9.1 Iterating over map values

Use the macro `for_map_value` to iterate over the values of a map or multimap as follows:

```
for_map_values (type var, map)
    <body>
```

This is identical to

```
for (auto &kv : map)
{
    type var = kv.second;
    <body>
}
```

Example

```
std::map<string, int> test_scores;
…
int total = 0;
for_map_values (int score, test_scores)
    total += score;
log("Average score = %f\n", (float) total / test_scores.size());
```

9.2 Iterating over map keys

Use the macro `for_map_keys` to iterate over the values of a map or multimap as follows:

```
for_map_keys (type var, map)
    <body>
```

This is identical to

```
for (auto &kv : map)
{
    type var = kv.first;
    <body>
}
```

Example

```
std::map<string, int> test_scores;
…
log("The following tests were applied:\n");
for_map_values (const string &test, test_scores)
    log("    %s\n", *test);
```

9.3 Iterating over (key, value) pairs

The templated `MapItem` structure is a wrapper around `std::pair` that exposes `key` and `value` public data members. It can be used within a range-based for loop as follows:

```
for (MapItem<K,T> it : map)
{
    K key   = it.key;
    T value = it.value;
    …
}
```

This is identical to

```
for (std::pair<K,T> it : map)
{
    K key   = it.first;
    T value = it.second;
    …
}
```

In practice there is no need to immediately assign the key and value to stack variables; in the above code this is purely to illustrate how the key and value are accessed. The `MapItem` structure can either be templated on the key and value type (as above) or on the map type:

```
for (MapItem<std::map<K,T>> it : map)
```

The latter form may be more convenient when a typedef exists for the map type.

The `value` member is a non-const reference that can be modified. To iterate over a const map or multimap, use `ConstMapItem`, which can also be template on either the key and value types or the map type.

Example

```
typedef std::map<string, int> TestMap;
TestMap test_scores;
…
log("Test results:\n");
for (MapItem<TestMap> it : test_scores)
    log("    %s: %d\n", *it.key, it.value);
```

10.0 Enumerations

Standard enumerations have a number of shortcomings:

- They are always 4 bytes in size, making them unsuitable for data structures that need to be compact.
- Enumeration symbols are defined in the same namespace as the enumeration, leading to cumbersome symbol names to avoid name collisions with other constants.
- String conversion, desirable for logging and debug output, must be manually implemented and kept in sync with the enumeration.
- There is no way to automatically determine the number of values in an enumeration, which often leads to the addition of a sentinel value at the end which is not logically part of the enumeration.

The first two shortcomings are addressed by C++11 enum classes; descore provides an alternate macro-based mechanism for defining enumerations that addresses all of them. The macro simultaneously defines the enumeration values, a minimially-sized container type, conversions to and from strings, and a descriptor object that provides a generic reflection interface for enumerations.

10.1 Basic usage

Enumerations are declared using the macro

```
DECLARE_ENUMERATION(name, value1, value2, …)
```

Logically, this declares the enumeration `name`. The actual implementation of `name` is a structure containing a minimally-sized integer, so the size of a descore enumeration is generally 1 byte (it will be larger than 1 byte only if one of the enumeration values is at least 256).

`name` can be used in the same manner as an actual enumeration with one important difference: the enumerations values are all scoped by `name`, i.e. they must be referred to as `name::value1`, `name::value2`, etc.

Example

```
DECLARE_ENUMERATION(Direction, up, down, left, right);
…
void hop (Direction dir, int &x, int &y)
{
     switch (dir)
     {
          case Direction::up:
               y++;
               break;
          case Direction::down:
               y--;
               break;
          case Direction::left:
               x--;
               break;
          case Direction::right;
               x++;
               break;
     }
}
```

10.2 String conversion

A specialization of the `str()` function is automatically provided that takes a variable of an enumeration type and returns its string representation. The string is simply the text of the value supplied to the `DECLARE_ENUMERATION()` macro, without scoping. A specialization of the `stringTo()` function is automatically provided that takes a string and converts it to an enumeration value. The string must exactly match the text of one of the values that was supplied to the `DECLARE_ENUMERATION` macro; if the string conversion fails then a `strcast_error` is thrown.

Example

```
DECLARE_ENUMERATION(Direction, up, down, left, right);
…
void hop (Direction dir, int &x, int &y)
{
     log(“Hopping in direction ‘%s’\n”, *
     …
}
void evalHop (const string &dir, int &x, int &y)
{
     hop(stringTo<Direction>(dir), x, y);
}

// Sample output
Hopping in direction ‘left’
```

10.3 Initialization values

By default, the enumeration values are assigned incrementally starting with 0. As with regular enumerations, explicit initialization expressions can optionally be provided for the values within the `DECLARE_ENUMERATION()` macro. There are a few restrictions on these arithmetic expressions:

- Only integer expressions are supported.
- After any macros are expanded, the expression can only be in terms of integer literals and previous values of the current enumeration.
- The values must be strictly increasing.

Examples

```
DECLARE_ENUMERATION(ApplicationError,
     SyntaxError,
     ParseError,
     ReadError = 0x100,
     WriteError
);

#define HAS_OVERFLOW 1

DECLARE_ENUMERATION(StateFlags,
     Overflow = 1,
     Carry = 2,
     Exception = 4,
     AllFlags = HAS_OVERFLOW ? (Overflow | Carry | Exceptoin) : (Carry | Exception)
);
```

10.4 Enumeration types

The underlying integer type used within an enumeration is exposed as `value_t` (a typedef) within the structure. In addition, if `TEnum` is any enumeration type

(either a standard enumeration or a descore enumeration), then its underlying integer type (which is `int` for a standard enumeration) can be obtained as

```
descore::EnumerationValueType<TEnum>::value_t
```

10.5 Enumeration interface

The interface `descore::IEnumeration` is used to describe descore enumerations. It has the following functions:

`string getName () const`

Returns the name of the enumeration (the first argument supplied to `DECLARE_ENUMERATION`).

`int getValue (const string &symbol) const`

Given the string representation of one of the enumeration values, returns the integer value. If the string does not match any of the enumeration values, returns -1.

`string getSymbol (int value) const`

If the value is one of the enumeration values, returns the corresponding string. Otherwise, returns "*name*`::???`".

`int maxValue () const`

Returns the maximum value that appears in the enumeration (which is also the last value since descore enumeration values are required to be monotonically increasing). Returns -1 if the enumeration is empty.

`bool isValid (int value) const`

Returns true if the value is one of the enumeration values, false otherwise. If no initialization values are supplied to `DECLARE_ENUMERATION`, then this returns true if and only if `value` is in the range [0, `maxValue()`]. If initialization values are supplied then there may be holes in this range for which `isValid()` returns false.

There are two ways to obtain a pointer to an `IEnumeration`. The first is to call the following static function defined for all enumerations:

```
static descore::IEnumeration *<enumeration>::getType ()
```

The second, useful in a template with type T, is to call

```
descore::IEnumeration *descore::getEnumerationType<T>()
```

The second form returns NULL when T is not a descore enumeration, allowing this form to be used when the type of T may or may not be an enumeration.

Example

```
DECLARE_ENUMERATION(Direction, up, down, left, right);
…
descore::IEnumeration *e = Direction::getType();
string name = e->getName();                    // name = "Direction"
int downval = e->getValue("down");             // downval == Direction::down
string downstr = e->getSumbol(1);              // downstr = "down"
int maxval = e->maxValue();                    // maxval = 3
bool valid3 = e->isValid(3);                   // valid3 = true
bool valid4 = e->isvalid(4);                   // valid4 = false;
```

10.6 Use in parameters

A descore enumeration can be used as the type for a parameter. If a list of valid values is not explicitly supplied when the parameter is declared, then the parameter will use the full set of enumeration values as its options.

Example

```
DECLARE_ENUMERATION(Direction, up, down, left, right);

Parameter(Direction, InitialDirection, Direction::up,
          “Specifies the initial direction of travel”);


$ dotravel –showparams InitialDirection

name              type       default
----              ----       -------

InitialDirection  Direction  up
    (up, down, left, right)
    Specifies the initial direction of travel
```

11.0 Type Traits

```
#include <descore/descore.hpp>
```

The descore library provides a number of macros for static compile-time evaluation of type properties. These macros complement those already supported by the MSVC and gcc compilers.

11.1 Type macros

`__is_pointer(T)`

Statically evaluates to true if the type is a pointer, false otherwise.

`__is_const(T)`

Statically evaluates to true if the type is constant, false otherwise.

`__non_const_t(T)`

Evaluates to the type with const qualifier stripped off (if present). Can be used to declare variables.

`__is_bag_of_bits(T)`

Statically evaluates to true if the type is a "bag of bits", i.e. a primitive type, an enumeration, or a class/union with the default copy constructor and copy assignment operator. It is always safe to use `memcpy()` to copy the contents of a bag-of-bits type.

`__is_same_type(T1, T2)`

Statically evaluates to true if the two types are identical, false otherwise.

11.2 Type traits structure

The templated `type_traits` structure captures some of the above information within an object that can be used at run-time. In addition, a static function is provided to obtain a type's demangled name.

`bool descore::type_traits<T>::is_pointer`

Compile time constant indicating whether or not a type is a pointer.

`bool descore::type_traits<T>::is_const`

Compile time constant indicating whether or not a type is constant.

`bool descore::type_traits<T>::is_bag_of_bits`

Compile time constant indicating whether or not a type is a "bag of bits".

`const char *descore::type_traits<T>::name ()`

Obtain a type's demangled name.

12.0 Coverage Assertions

```
#include <descore/Coverage.hpp>
```

A coverage assertion ensures that one or more conditions are met at some point during program execution. Unlike a regular assertion which must be true every time it is evaluated, a coverage assertion keeps track of the various states of interest that are encountered during execution, and is not actually tested for success/failure until after the program has been run. Each coverage assertion is associated with a (file, line) pair and updates its state whenever execution reaches that particular line.

All coverage assertions must appear within coverage assertion *sections* which are delineated by the following macros:

```
BEGIN_COVERAGE(<section name>)
END_COVERAGE()
```

These macros must appear at file scope. Coverage sections cannot be nested or overlapped.

Example

```
BEGIN_COVERAGE("router arbitration");
...
<code and coverage assertions>
...
END_COVERAGE();
```

12.1 Line coverage

The simplest type of coverage assertion is a line coverage assertion:

`coverLine()`

This assertion simply validates that the program execution at some point encountered that specific line. The `coverLine()` macro relies on the automatically incrementing `__COUNTER__` macro which is available in Microsoft builds and GCC version 4.3 and up. It is disabled when the `__COUNTER__` macro is unavailable.

12.2 Basic coverage

The following coverage assertions each verify that a single variable or expression takes on all specified values during the course of program execution. The assertions support only 32-bit signed integer values (except for `coverBool`, which supports booleans).

`coverAssert(expr)`

Asserts that *expr* is true at least once.

`coverBool(expr)`

Asserts that the expression takes on each value (true, false) at least once.

`coverRange(expr, low, high)`

Asserts that the integer expression takes on every value from *low* to *high* (inclusive) at least once.

coverRangePredicated(expr, low, high, predicate)

Asserts that the integer expression takes on every value from *low* to *high* (inclusive) for which the predicate is true. *predicate* is an expression which contains a single variable named `value` and is evaluated for every integer in [*low*, *high*]. It should evaluate to a boolean; an integer is included in the set of coverage values if *predicate* evaluates to true when `value` is set to that integer.

coverValues(expr, value1, value2, ...)

Asserts that the integer expression takes on every explicitly listed value at least once.

Examples

```
coverAssert(x > 7);
coverBool(op == OP_NOP);
coverRange((adress >> 4) & 0x3f, 0, 63);
coverRangePredicated(base + offset, 0, 127, (value & 0xf) != 0);
coverValues(p, 2, 3, 5, 7, 11, 13, 17, 19);
```

12.3 Cross-coverage

A cross-coverage assertion verifies that multiple variables or expressions take on all possible combinations of values during program execution. A cross-coverage assertion is constructed in two steps. First, two or more named coverage items must be declared: each coverage item defines a set of values of interest for a single variable or expression. Second, the cross-coverage assertion itself is declared which specifies a set of items.

Coverage items are declared using the following macros:

CoverageItemBool(name, expr)
CoverageItemRange(name, expr, low, high)
CoverageItemRangePredicated(name, expr, low, high, predicate)
CoverageItemValues(name, expr, value1, value2, ...)

These macros have the same format and meaning as the corresponding coverage assertions but with two differences. First, they all take an initial *name* parameter which is used to declare a coverage item variable having that name (so in particular the name cannot collide with any other variable names). Second, rather than asserting that *expr* must take on all prescribed values, they simply define the set of values for later use.

Once two or more items, have been defined, a cross coverage assertion can be declared:

coverCross(item1, item2, ...)

The items are specified by name, which must match exactly the names used in the `CoverageItem` macros.

Example

```
CoverageItemBool(item_carry, carry);
CoverageItemValues(item_op, op, OP_ADD, OP_SUB, OP_MUL);
CoverageItemBool(item_compare, a > b);
coverCross(item_carry, item_op, item_compare);
```

A cross-coverage assertion can also supply a predicate which indicates which combinations of values should be covered:

coverCrossPredicated(predicate, item1, item2, ...)

predicate is an expression which contains the variables *item1*, *item2*, etc. Note that this differs from the predicate supplied to `coverRangePredicated()` which contains a single variable named `value`. In this case the items already have names, so these names are used directly within the predicate. Only those combinations of values for which the predicate holds are considered for coverage. The following example covers all combinations of two integers where the sum of the integers is odd:

```
CoverageItemRange(item_a, 0, 10);
CoverageItemRange(item_b, 10, 20);
coverCrossPredicated((item_a ^ item_b) & 1, item_a, item_b);
```

12.4 Covering return values

As a convenience, the following macros, which correspond exactly to the five macros defined in Section 12.2, can be used to simultaneously return a result from a function and provide coverage on its value:

returnCovered(expr)
returnCoveredBool(expr)
returnCoveredRange(expr, low, high)
returnCoveredRangePredicated(expr, low, high, predicate)
returnCoveredValues(expr, value1, value2, ...)

The return type should be bool for `returnCoveredBool` and should be an integer type with at most 32 bits (signed) for the others. These macros expand to a block enclosed by curly braces, e.g.

```
returnCoveredBool(isZero);
```

expands to

```
{
    bool _returnValue = isZero;
    coverBool(_returnValue);
    return _returnValue;
}
```

As such, it is safe to use these macros in single-line 'if' clauses, but the semicolon at the end of the line must be omitted if there is a subsequente 'else':

Example

```
if (done)
    returnCoveredBool(isZero)    // Semicolon must be omitted here
else
    return false;
```

12.5 Covering conditional tests

As a convenience, the following macro can be used to simultaneously begin an 'if' clause and cover the boolean conditional expression:

cover_if (expr)
…

This macro expands to:

```
coverBool(expr);
if (expr)
    …
```

so there are two restrictions on its use. First, *expr* cannot have side effects.

```
cover_if (++n == 7)    // WRONG!!!
    …

++n;
cover_if (n == 7)      // Correct
    …
```

Second, `cover_if` cannot appear in a single-line clause.

```
for (i = 0 ; i < 10 ; i++)
    cover_if (active[i])    // WRONG!!!
        numActive+++;

for (i = 0 ; i < 10 ; i++)
{
    cover_if (active[i])    // Correct
        numActive++;
}
```

12.6 Controlling coverage

The preprocessor symbol `_COVERAGE` must be defined to support coverage assertions. All coverage assertions are initially disabled; assertions can be dynamically enabled or disabled on a section-by-section basis using the following functions:

```
void CoverageAssertion::enable (const char *section)
void CoverageAssertion::disable (const char *section)
```

These functions enable/disable the specified coverage section(s), where *section* must match exactly the section name specified in the `BEGIN_COVERAGE()` macro. If multiple sections have the same name, then they are enabled/disabled as a group.

Enabling or disabling coverage assertions for a section has the effect of resetting all the assertions in that section, so these calls should only be used between runs to set up assertions for the next run.

At the end of a run, the coverage assertions must be explicitly evaluted to generate a report of failed assertions using the following function call:

```
bool CoverageAssertion::checkAndReset ()
```

This function validates all coverage assertions in all files with assertions enabled, then resets these assertions. Any coverage assertion failures will be logged and reported using `logerr()`. For coverage assertions with multiple values, the missing cases will be listed explicitly up to a maximum of 100 cases. This function returns true if all coverage assertions were satisfied and false otherwise.

An important technical note is that the automatically incrementing `__COUNTER__` macro, supported in Microsoft and GCC 4.3+ builds, is the only mechanism that can detect coverage assertions that are never visited. In these builds, missed assertions will be reported as coverage assertion failures with the message

```
Missing coverage point <#> in file <filename>
```

Every line, basic or cross-coverage assertion in a file is a coverage point; the coverage points in a file are numbered sequentially starting with 1. In other builds, missing coverage points will result in silent failures.

12.7 Other error messages

```
"Missing BEGIN_COVERAGE() before END_COVERAGE()"
```

Indicates an `END_COVERAGE()` with no corresponding `BEGIN_COVERAGE()`.

```
"Missing END_COVERAGE() for coverage section <name>"
```

Indicates a `BEGIN_COVERAGE()`with no corresponding `END_COVERAGE()`. Note that coverage sections cannot be nested.

```
Warning: inconsistent counter detected in file <filename>
    Line coverage has been disabled for this file.
```

Indicates that a header file with line coverage has a different value of `__COUNTER__` depending on which source file it is being included from. This problem can be fixed by ensuring that whenever two source files include a common header file which contains coverage assertions, then they include exactly the same set of header files with coverage assertions and in the same order. Where possible, this problem can also be avoided by moving the code (along with the coverage assertions) to a source file.

```
"Coverage assertion must appear within a coverage section"
```

Indicates that the specified coverage assertion is not in a coverage section defined by `BEGIN_COVERAGE()/END_COVERAGE()`.

```
Maximum size (0x10000000) exceeded for cross-coverage assertionc
```

The total number of value combinations in a cross-coverage assertion is not allowed to exceed 2^{28} (~256M combinations).

12.8 Templates

A coverage assertion within a templated function (or class) will in general give rise to multiple instances which share the same underlying bookeeping. It is important to ensure that the coverage values do not depend on the template parameters, but the variable/expression which is being covered (and takes on these values) may do so. For example, the following is incorrect:

```
template <int n> void someFunction (int k)
{
    coverValues(k, n-1, n, n+1);   // WRONG
}
```

whereas the following is correct:

```
template <int n> void someFunction (int k)
{
    coverValues(k-n, -1, 0, 1);
}
```

13.0 String Buffers

```
#include <descore/StringBuffer.hpp>
```

The `strbuff` class is a lightweight, efficient implementation of shared strings with reference counting that can be used to construct and append to strings using printf semantics when performance is a concern. A `strbuff` can also be archived using the | operator.

13.1 Member functions

`strbuff::strbuff ()`
`strbuff::strbuff (const char *fmt, ...)`

The `strbuff` constructor allows the string buffer to be optionally initialized using printf semantics.

`strbuff::strbuff (const strubff &rhs)`
`strbuff &strbuff::operator& (const strbuff &rhs)`

Copy constructor and copy assignment operator.

`void strbuff::putch (char ch)`

Append a single character to a string buffer.

`void strbuff::puts (const char *sz)`

Append a raw string to a string buffer.

`void strbuff::append (const char *fmt, ...)`
`void strbuff::vappend (const char *fmt, va_list args)`

Append to a string buffer using printf or vprintf semantics.

`int strbuff::len () const`

Return the length (in characters) of the current string.

`void strbuff::truncate (int len)`

Truncate the string buffer to the specified length (in characters). An assertion failure is generated if the specified length is longer than the current length of the string buffer.

`void strbuff::clear ()`

Reset the string buffer to the empty string. Equivalent to `truncate(0)`.

`const char * strbuff::operator* ()`
`strbuff::operator const char * ()`

A string buffer can be implicitly converted to a `const char *` by the compiler, and can be explicitly converted using the * operator. A string buffer contains a single pointer that points directly to the string, so with some compilers it is safe to pass a `strbuff` directly as a `%s` argument to a function with printf semantics.

```
bool strbuff::operator== (const char *rhs)
bool strbuff::operator!= (const char *rhs)
bool strbuff::operator== (const strbuff &rhs)
bool strbuff::operator!= (const strbuff &rhs)
```

String comparison operators.

Example

```
// Construct a string representing a vector of nodes
strbuff vecnodestr (const std::vector<Node *> &vecNodes)
{
     strbuff s("[");
     for (unsigned i = 0 ; i < vecNodes.size() ; i++)
          s.append((i ? ", %s" : "%s"), vecNodes[i]->getName());
     s.putch(']');
     return s;
}
```

14.0 Miscellaneous

14.1 Types

```
#include <descore/descore.hpp>
```

type	description
`uint8`	8-bit unsigned integer
`uint16`	16-bit unsigned integer
`uint32`	32-bit unsigned integer
`uint64`	64-bit unsigned integer
`int8`	8-bit signed integer
`int16`	16-bit signed integer
`int32`	32-bit signed integer
`int64`	64-bit signed integer
`byte`	8-bit unsigned integer (same as `uint8`)

14.2 Macros

```
#include <descore/descore.hpp>
```

`DECLARE_NOCOPY(Class)`

Use within a class declaration to declare a private copy constructor and assignment operator to prevent instances of the class from accidentally being copied or assigned.

Example

```
class FlexTile
{
     DECLARE_NOCOPY(FlexTile);
public:
     FlexTile ();
…
```

`ALWAYS_INLINE`

Use in place of the `inline` keyword to force inlining.

`ATTIRBUTE_UNUSED`

Place after an unused function argument to indicate that the argument is unused in gcc builds, avoiding warnings about unused arguments. Does nothing in MSVC builds. Note that unused argument warnings can also be avoided by omitting the argument name.

`ATTRIBUTE_PRINTF(`*`N`*`)`

Place after the declaration of a function with printf semantics to enable automatic verification of the arguments in gcc builds. *N* is the index of the formatting string argument, where the first argument has index 1, and for all member functions argument 1 is the implicit “this” argument. Does nothing in MSVC builds.

Example

```
void append_info (int source, const char *fmt, ...) ATTRIBUTE_PRINTF(2);
```

ASSUME_TRUE(x)
ASSUME_FALSE(x)

Macros that simply return x but, in gcc builds, direct the compiler to assume that the value is true or false (respectively) when it appears within a conditional, allowing the branches to be optimized.

Example

```
if (ASSUME_TRUE(x >= 5))
     commonCase();
else
     rareCase();
```

14.3 Helper functions

uint32 crc32 (uint32 crc, const byte *data, int len)

```
#include <descore/crc.hpp>
```

Computes a 32-bit CRC of the *len* bytes pointed to by *data*, starting with the initial CRC value *crc*.

Example

```
uint32 crcStrings (const std::vector<std::string> &v)
{
     uint32 crc = 0;
     for (unsigned i = 0 ; i < v.size() ; i++)
          crc = crc32(crc, (const byte *) *v[i], v[i].size());
     return crc;
}
```

bool wildcardMatch (const char *s1, const char *s2,
bool case_sensitive = true)

```
#include <descore/Wildcard.hpp>
```

Fully match two wildcard strings, where '*' matches any sequence of zero or more characters and '?' matches any single character (the two wildcard strings match if there exists some non-wildcard string that matches both of them). `case_sensitive` specifies whether or not the matching is case-sensitive.

Example

```
void openFile (const char *filename)
{
     if (wildcardMatch(filename, "*.txt"))
          openTextFile(filename);
     else
          openBinaryFile(name);
}
```

```
const char *wildcardFind (const char *haystack,
                          const char *needle,
                          bool case_sensitive = true)
```

```
#include <descore/Wildcard.hpp>
```

Return the first occurence of a wildcard needle within a non-wildcard haystack (i.e. the start of the first non-wildcard substring of haystack that matches needle), or NULL if the wildcard needle could not be found.

```
bool wildcardSubsumed (const char *s1, const char *s2,
                       bool case_sensitive = true)
```

```
#include <descore/Wildcard.hpp>
```

Returns true if any string that would match the first wildcard string would also match the second.

```
int popcount (<type> val)
```

```
#include <descore/utils.hpp>
```

Fast inline implementation of popcount for any integer type. Returns the number of 1's in the binary representation of *val*.

```
int lsb (<type> val)
```

```
#include <descore/utils.hpp>
```

Fast inline implementation of lsb for any integer type. Returns the index of the least significant 1 in the binary representation of *val*. If *val* is zero, returns the size in bits of <type>.

14.4 String table

```
#include <descore/StringTable.hpp>
```

Class that stores a set of static strings so that each string is stored exactly once. There are two member functions:

```
const char *StringTable::insert (const char *string)
```

Inserts `string` into the table (if it is not already in the table) and returns a pointer to the string that is guaranteed to be valid for the lifetime of the table. If `string` is `NULL` then this function returns `NULL`. The string table guarantees a one-to-one correspondence between distinct strings and pointers, so in particular two pointers which were obtained by calling `insert()` can be compared directly without using `strcmp`.

```
void clear ()
```

Clear the string table. This invalidates all pointers that were returned by `insert()`.

Example

```
descore::StringTable nameTable;
…
void Node::setGroupName (const char *groupName)
{
     m_groupName = nameTable.insert(groupName);
}
```

14.5 'Using' declarations

The `descore.hpp` header includes the following 'using' declarations:

```
using std::max;
using std::min;
using std::string;
```

These functions/classes can therefore be used without the `std::` prefix.

14.6 Hexadecimal helpers

```
#include <descore/hex.hpp>
```

Several arrays and functions are provided to assist in the creation and parsing of hexadecimal strings.

```
const char hex_to_sz[16][2];
const char hex_to_sz_upper[16][2];
```

`hex_to_sz[x]` returns a null-terminated single-character string containing the hex digit corresponding to `x`, with a–f in lower-case. `hex_to_sz_upper` is the same, but returns A–F in upper-case.

```
const char hex_to_ch[16];
const char hex_to_ch_upper[16];
```

`hex_to_ch[x]` returns the single-character hex digit corresponding to `x`, with a–f in lower-case. `hex_to_ch_upper` is the same, but returns A–F in upper-case.

```
const int8 * const ch_to_hex;
```

`ch_to_hex[ch]` converts a hexadecimal character `ch` to to its numerical equivalent (0–15), or returns -1 for non-hexadecimal characters. The array is defined from -128 to 255, so it can be indexed safely by either signed or unsigned characters.

```
bool is_hex (char ch)
```

Returns true if `ch` is a hexadecimal character, false otherwise.

```
int8 to_hex (char ch)
```

Returns `ch_to_hex[ch]`. Provided for convenience to eliminate compiler warnings about indexing an array with a char.

```
template <typanename T> int parse_hex (const char *ch, T &val)
```

Parses a hexadecimal string (`ch`) until it reaches the first non-hexadecimal character and stores the integer result in `val`. This function is much faster than `scanf("%x")`. Returns the number of characters the number of characters that were processed.

Example

```
void print_hex (uint32 x)
{
    for (int s = 28 ; s >= 0 ; s -= 4)
        putchar(hex_to_ch[(x >> s) & 15]);
}

uint32 from_hex (const char *ch)
{
    uint32 ret = 0;
    int d;
    while ((d = ch_to_hex[*ch++]) != -1)
        ret = (ret << 4) | d;
    return ret;
}
```

14.7 Detecting memory leaks with AllocTracker

```
#include <descore/AllocTracker.hpp>
```

`AllocTracker` is a helper base class that can automatically detect memory leaks by counting the number of constructions/destructions for a given type. To track allocations for a type `T`, the type should inherit from `AllocTracker<T>`:

```
class Node : public descore::AllocTracker<Node>
{
    ...
```

`AllocTracker` is a zero-sized base class and so does not increase the size of the derived class.

By default, the memory leak check is performed at program exit time, and an error message is printed for every type with a memory leak. To perform the check at an intermediate point when it is known that all instances of a type should have been deleted, call the static helper function `verifyAllDeleted()`:

```
descore::AllocTracker<Node>::verifyAllDeleted();
```

When this function is called and a memory leak is detected, then an exception is thrown (of type `descore::runtime_error`) whose `what()` string contains the error message.

14.8 Using pointers in STL trees

By default, descore disallows the use of STL sets and maps indexed by pointers (via a static assertion) since this is a common source of non-determinism. In cases where an STL tree indexed by a pointer is necessary, the static assertion can be avoided in one of three ways:

1. Manually define a custom `std::less<T*>` for the pointer type `T*`
2. Specify `descore::allow_ptr<T*>` as the last template argument, e.g.

```
std::map<void *, Node *, descore::allow_ptr<void *> > nodeMap;
```

3. Define `std::less<T*>` using `DESCORE_ALLOW_STL_TREE_TYPE(T*)`, e.g.

```
DESCORE_ALLOW_STL_TREE_TYPE(MyNodeType *);
std::set<MyNodeType *> nodes;
```

14.9 Comparison of STL trees

A number of issues with the default comparison operators for sets and maps are addressed within descore. First, the default comparisons use `operator<` to compare individual elements instead of `std::less<T>`. This produces surprising and inconsistent results, and defeats the intention of `std::less<>` specializations. Descore redefines the comparisons in terms of `std::less<T>`.

Second, the default comparison operators for multimaps do not produce correct results because they don't account for the fact that multiple instances of a given key can appear in any order. The descore comparison operators handle multimaps correctly.

Finally, the equality operators (==, !=) on sets and maps require the existence of `operator==` for the contained types, whereas the sets and maps themselves only require the existence of `std::less<T>`. Use the functions `descore::eq` and `descore::neq` when `operator==` is unavailable for the index type.

In some cases the standard comparison operators for sets and maps will generate a compiler error due to an ambiguous operator overload; to fix this problem replace the operator with one of the functions `descore::eq`, `descore::neq`, `descore::lt`, `descore::gt`, `descore::lte`, or `descore::gte`.

14.10 Pretty-printing a set of strings

```
#include <descore/PrintStrings.hpp>

void descore::printStrings (const std::set<string> &strings,
                            FILE *fout = stdout)
void descore::printStrings (const std::vector<string> &strings,
                            FILE *fout = stdout)
```

Each of these functions prints a set of strings, sorted lexigraphically, using a multi-column ls-style format. The second function calls the first, so duplicate strings are ignored. By default the formatted strings are printed to stdout, but the second parameter can be used to select an arbitrary file.

Example

```
std::set<string> s;
fromString(s, "{a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16,
               b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15, b16,
               common, forceTrace, fruit, rock, trace1, trace2, trace3}");
descore::printStrings(s);

Output:

a1   a12  a15  a3  a6  a9   b11  b14  b2  b5  b8      forceTrace  trace1
a10  a13  a16  a4  a7  b1   b12  b15  b3  b6  b9      fruit       trace2
a11  a14  a2   a5  a8  b10  b13  b16  b4  b7  common  rock        trace3
```

14.11 Pretty-printing a table of strings

```
#include <descore/PrintTable.hpp>
```

The helper class `descore::Table` is used to construct, and then pretty-print, a table of strings. The table is printed with a header row containing column names, followed by a horizontal divider, followed by the rows, with vertical dividers between the columns. Each column is minimally sized to fit its cells, including a single space of internal padding on the left and right of each cell. By default, numeric

columns are right-aligned, and all other columns are left-aligned. The header row and divider are omitted if the table has no column names. `descore::Table` has the following member functions:

```
Table::Table ()
Table::Table (string column1, …)
```

Create a table, optionally initialized with one or more column names.

```
void Table::addColumn  (string column)
void Table::addColumns (string column1, string column2, …)
```

Append one or more column names to the table.

```
void Table::addDivider ()
```

Add a horizontal divider at the current point in the table.

```
void Table::addRow ()
void Table::addRow (string cell1, …)
```

Add a new row to the table, optionally initialized with one or more cells.

```
void Table::rowAppend (string cell1, …)
```

Append one or more cells to the current table row.

```
void Table::print (descore::LogFile f = LOG_STDOUT)
```

Print the table to a log file.

A number of column flags can be used to modify the appearance of the columns. A column flag is specified by appending "|<FLAG>" to the column name, where <FLAG> is one of the flags in the following table. Multiple flags can be specified for a single column.

Flag	Description
NODIV	Don't separate this column from the previous one with a vertical divider
NOLPAD	Don't internally pad this column with a space on the left
NORPAD	Don't internally pad this column with a space on the right
NOPAD	Don't internally pad this column with a space on either side
RALIGN	Right-align this column (default for numeric columns)
LALIGN	Left-align this column (default for non-numeric columns)

Example

```
descore::Table profile("function", "time", "calls|NODIV|NOLPAD");
profile.addRow("getchar", "379.2", "57");
profile.addRow("putchar", "7.6", "2");
profile.print();

Output:

 function |  time calls
----------+-------------
 getchar  | 379.2    57
 putchar  |   7.6     2
```

14.12 Statistics

```
#include <descore/Statistics.hpp>
```

The descore library provides a mechanism for aggregating and reporting named statistics. Specifically, the following global functions are provided:

```
void Statistics::recordStat (float val, const string &descr)
void logStats (const char *format_str = "%.2f")
void logStatsNoClear (const char *format_str = "%.2f")
```

`recordStat()` records a single named statistic. Once all statistics have been recorded, `logStats()` writes the statistics to `LOG_STDOUT` using the specified floating-point format. The minimum, maximum and average value is reported for every distinct description string. If a description string is of the form "Header: description", then when the statistics are displayed all statistics with the same header will be grouped together. The statistics are cleared by `logStats()`, so it is necessary to re-record the statistics before they can be logged again. To log the statistics without clearin the, use `logStatsNoClear()`.

As a convenience, the `ADD_STAT` macro is provided which wraps the call to `recordStat`, providing a conversion to `float` if necessary:

```
ADD_STAT(value, description)
```

The recommended usage of statistics is as follows.

1. Declare class member variables to keep track of statistics:

```
int m_numCacheHits[NUM_CACHE_LINES];
int m_numCacheMisses[NUM_CACHE_LINES];
```

2. In each object which contains statistics, add code to update the statistics as appropriate, then add a `recordStats()` function (this particular function name is not required). Within this function, use the `ADD_STAT` macro to record individual named statistics:

```
void recordStats ()
{
     for (int i = 0 ; i < NUM_CACHE_LINES ; i++)
     {
          ADD_STAT(m_numCacheMisses[i], "Mem: Number of cache misses");
          ADD_STAT(m_numCacheHits[i],   "Mem: Number of cache hits")j;
     }
}
```

3. When execution finishes, call `recordStats()` for each object:

```
for (int i = 0 ; i < numCaches ; i++)
     m_cache[i]->recordStats();
```

4. Call `Statistics::logStats()` to display the statistics:

```
Statistics::logStats();
```