SWORD Documentation

http://www.erik-n.net/sword/

Eric Nicolas <enicolas@dvdfr.com>, May 30, 2004

Table of Contents

1 Reference Guide

1.1 Exception handling

1.1.1 Introduction

When designing an exception handling strategy within an application code, the developer usually faces at least two problems:

- Define an exception classes hierarchy,
- Provide enough context information with each exception thrown.

Very often the solutions adopted are over-complicated: too complex exceptions tree, many variables to fill when creating the context. The immediate consequence is that on a reasonably sized project, developers stop providing the required context information and throw only base classes.

As a consequence, the exception handling strategy proposed in Sword is a lot simpler that what is usually offered, but proves to be more pragmatic and more effective in the end:

- Defined as small as possible exception classes hierarchies. Use exception classes inheritance only when you really need to selectively catch the errors,
- Do not provide typed information about the context, but provide a meaningfull textual message which will always be usefull both to the user and to the application support people.

1.1.2 Exception classes hierarchy

The base class for all exceptions in a Sword-based application is **sword::Exception**. It derives from std::exception, so can be caught together with the standard library exceptions and its error message can be inspected using the standard what() method.
User-defined exceptions usually can be defined in one short line such as:

```
class MyException : public sword::Exception {};
```

And a tree of exception classes can be further expanded such as in:

```
class MyDerivedException : public MyException {};
```

As the policy is to store only a textual message (which is embedded in sword::Exception), derived exception classes do not need to define any field. The message streaming mechanism (see next paragraph) uses only the default constructor in exceptions, so no additional constructor is usually required.

Sword also defines a few other standard exception classes, which all derive from sword::Exception:

- **sword::UnimplementedException**: use it to report that some code has not been implemented yet,
- **sword::InternalException**: use it to report an error that is most certainly due to an internal inconsistency of the application code.

1.1.3 Message streaming

When only textual messages are used in error reporting, one key point in the overall quality of the exception handling is how easy it is to create those messages. If too difficult, application developers will soon stop doing it and the error output will become totally cryptic.
Sword provides a very easy way for creating exception messages. The text stream **sword::estream** should be used for streaming the message content. As with usuall text streaming in the standard library, any time on which the '<<' operator apply can be streamed as well. When

the exception is raised, the accumulated messages is automatically used as the exception description.

1.1.4 Example

```
#include <sword/sword.h>
// no additional include file necessary
```

The following example uses the message streaming, and the sword::Exception class:

```
try
{
  sword::estream << “This is the error message and value=” << value;
  throw sword::Exception();
}
catch(sword::Exception &e)
{
  std::cerr << e.what() << std::endl;
}
```

1.2 Smart and Auto pointer

1.2.1 Introduction

Memory management in C++ can be a very difficult task. As difficult as it is in “plain C” if no special technique is used. Objects are allocated on the heap using “new” and must be returned to the heap using “delete”. This kind of manual memoy management introduces at least three problems:

1. Ownership: Each object must have a designated “owner” who is in charge of deleting the object when it is no longer required. But very often in complex programs using lots of libraries it becomes difficult to clearly identify the owner for each object,
2. Memory Leaks: If you forget a “delete”, you introduce a memory leak. If your program has to run for a large amount of time, memory leaks are a real danger because they can completely exhaust the available computer's memory,
3. Memory corruption: If you delete the object too soon, you introduce a memory bug: at best a clean crash when someone else is accessing the object, at worse an undected misbehaviour, which will cause a crash later, when the diagnostic is very difficult.

Fortunately, it is now possible to use better techniques for memory management. Such “automatic memory management” techniques have been introduced in the C++ Standard Library with the std::auto_ptr<> template class, and are completed with the sword::AutoArray<> and the sword::SmartPtr<> template classes.

1.2.2 Reminder: std::auto_ptr<>

The standard auto pointer allows you to replace a hand-written instance destruction such as:

```
{
  A *a = new A(...);
  // ...
  delete a;
}
```

By an automatic destruction when the scope of the dynamically allocated instance ends:

```
{
  std::auto_ptr<A> a(new A(...));
  // ...
} // <- automatic destruction of 'a'
```

However, this an be used only for “local” instances as it is illegal to move auto pointers around. For example, this code is a bug:

```
// this is wrong code with a memory bug
std::auto_ptr<A> f()
{
  std::auto_ptr<A> a(new A(...));
  return a;
}

std::auto_ptr<A> a1 = f();
```

For such situations, you should use sword::SmartPtr<>.
The auto pointer cannot be used for arrays either, because it implements an automatic “delete” where a “delete[]” would be necessary. You should use sword::AutoArray<> in such situations.

1.2.3 sword::AutoArray<>

```
#include <sword/sword.h>
#include <sword/base.AutoArray.h>
```

This template class behaves exactly as auto pointer, but for arrays. The only real difference is that it implements a “delete[]” instead of a “delete” upon instance destruction.

```
{
  sword::AutoArray<A> array(new A[10]);
  // ...
} // <- automatically calls “delete[] array”
```

1.2.4 sword::SmartPtr<>

```
#include <sword/sword.h>
#include <sword/base.SmartPtr.h>
```

This template class implements smart pointer management, using a reference counting strategy. Usually you can use smart pointers everywhere instead of raw pointers and never use the delete keyword anymore !

1.2.4.1 Exemple and general interface

Here is a small exemple about how to use it and how the reference counting works:

```
class A { ... };
typedef sword::SmartPtr<A> PA;

PA f()
{
  PA a(new A(...));    // <- allocation performed
                       //    1 reference is pointing to this instance
  // ...
  return a;            // <- now 2 refs (a, a1 : the return value)
}                      // <- now 1 ref  (a1, a has disappeared)

{
  PA a1;
  a1 = f();            // <- 1 ref (a1)
}                      // <- a1 dissapears : 0 ref left
                       //    the instance is deleted
```

A smart pointer can really be used as a raw pointer. All usual and safe operations on pointers are implemented:

```
SmartPtr<X> sp1, sp3;
SmartPtr<Y> sp2;

if (sp1)        // test for non-null
if (!sp1)       // test for null
```

```
if (sp1 == 0)   // test for null
if (sp1 != 0)   // test for non-null
if (sp1 == sp2) // test for equality (even if X and Y are unrelated)
if (sp1 != sp2) // test for difference (even if X and Y are
unrelated)

sp1 < sp3       // order relation between smart ptrs on the same type

sp1 = sp3       // assignment between smart ptrs on the same type
sp1 = sp2       // assignment between unrelated types. this implies a
                // dynamic_cast<>, and thus 'sp1' may be null after
                // the '=' if the types did not match

*sp1            // access the object
sp1->           // access the object

sp1.getPtr()    // retrieve the underlying pointer
                // (USE WITH GREAT CARE!!)
```

1.2.4.2 Smart pointer casting

With raw pointers, you can always cast up or down an inheritance tree, plus you can cast everything to “void*”. Up-casts (from descendents to parents) is always safe but down-casts are dangerous (because you are not sure that the result really implements the descendent's interface):

```
class A { ... };
class B : public A { ... };

A* a;
B* b;

a = b; // always valid : B implements A
b = a; // may be valid but can lead to a bug
```

With smart pointers, you can also cast up and down the inheritance tree. However, all downcasts are checked using dynamic_cast<>. This way, if the cast was invalid, the target smart pointer variable will receive *null* instead of an invalid reference.

```
class A { ... };
class B : public A { ... };
typedef sword::SmartPtr<A> PA;
typedef sword::SmartPtr<B> PB;

PA a;
PB b;

a = b; // always valid
b = a; // ok if 'a' reference an actual implementation for 'B'
       // leads to b = 'null' otherwise
```

1.2.4.3 Limitations

Smart pointers are very usefull, but are not completely “magical”. There are some limitations:

1. You must never mix smart pointers and raw pointers.
   As smart pointers keep a reference count, any raw pointer to the same instance will not be “counted” and thus the smart pointer may think there are no more references when there is one left and delete the instance too soon,

2. You must be carefull about cycles.
   If you have a cycle, you end up with N instances with for instance 1 reference to each one, in cycle. The smart pointer will never see the last reference get to zero and thus will never delete those instances. You can reclaim the memory from a cycle by manually breaking one link in it.

   ```
   class A;
   typedef sword::SmartPtr<A> PA;

   class A {
   public:
   ```

```
  PA next;
}

// intuitive version, which causes a memory leak

{
  PA a1 = new A;
  PA a2 = new A;
  a1->next = a2;
  a2->next = a1;

  // a cycle is now created
  // the SmartPtr won't be able to free this memory

} // <- Here a1 and a2 are reclaimed, but the cycle remains in memory

// correct version

{
  PA a1 = new A;
  PA a2 = new A;
  a1->next = a2;
  a2->next = a1;

  // manually break one link in the cycle
  a1->next = 0;

} // <- Here a1 and a2 are reclaimed, and the instances are freed
```

1.3 Logger

1.3.1 Introduction

It is often usefull while trying to make an application more robust, to add “log” information within the code. The “log” is usually made of textual output of the program run, such as debugging infos, error reports, and other informative message. The log usually presents the following caracteristics:

1. It is not directly displayed to the application user. It is rather a support tool. It may be written in files, or in some “syslog” (or event viewer) system,
2. The “level” of logs (i.e. quantity and precision) is not always the same. When hunting a bug, the developer may want to activate all log messages, even debug ones, whereas in a stable production environment only critical errors may be logged,
3. The performance of the application must not be impacted by the logging code when the actual output of the log messages is not activated.

The Logger implementation provided in SWORD provides all those features plus a standard set of possible log outputs (called “media”). The framework is also extensible: you can create your own media, and is built in such a way than log messages created but not displayed anywhere will generate very low performance overhead (you can leave many “Debug” logs in your program and display them only when you are troubleshooting).

1.3.2 Log message emission

```
#include <sword/sword.h>
// no additional include file necessary
```

The **sword::Log** class provides the following interface for log messages output:

```
class Log {
public:
       //! The various possible Log levels
       enum Level { Debug, Info, Warning, Error, Fatal };

       // End of line for message flushing
       static class Endl {} endl;
```

```
        //! Singleton access to the logging stream
        static LogStream &log(Level level);

private:
        /* ... */
};
```

At any point in your code, you can thus send a log message to all the log media connected using a stream syntax such as:

```
using sword::Log;
Log::log(Log::Info)
        << "This is a message and value=" << value << Log::endl;
```

Warning: all log message lines must be terminated by Log::endl (and not std::endl).

1.3.3 Log levels

The following log levels are available in this order of priority:

1. Debug: for information that you want to display only when debugging the application,
2. Info: for information usefull most of the time, even usable by the end-user,
3. Warning: for non blocking, non critical error messages,
4. Error: for blocking error messages,
5. Fatal: for critical error messages. Usually this is followed by an application exit.

1.3.4 Connecting outputs

Each log message created will be displayed on one or more log media. The program must create and register the log media once. Each log media registration also specifies which level is displayed on the media. It is thus possible to configure your program for this kind of logging:

1. Send all log levels to a rotated log file,
2. Send all informative logs to the console,
3. Send all errors and critical errors to the syslog.

All log media must implement the sword::LogMedia interface:

```
class LogMedia {
public:
        LogMedia(const std::string &name);
        virtual void onLog(const Log::Entry &entry) = 0;
};
```

The following standard log media are provided, but you can also implement your own by deriving from **sword::LogMedia** and implementing the onLog method:

- **sword::LogMediaConsole**: displays the messages on the standard output console,
- **sword::LogMediaDebugger**: displays the messages in the debugger output window,
- **sword::LogMediaFile**: the messages are sent to a disk file,
- **sword::LogMediaFileRotate**: the message are sent to a disk file, and a new file is created every day at midnight,
- **sword::LogMediaSyslog**: displays the message in the "syslog" daemon (/var/log/messages under Unix, the Event Viewer under Win32).

The **sword::LogMediaManager** class is used to register the log media into the framework:

```
class LogMediaManager {
public:
        LogMediaManager();
        void registerMedia(Log::Level level1, Log::Level level2,
                           PLogMedia media);
};
typedef ACE_Singleton<LogMediaManager> TheLogMediaManager;
```

After a registerMedia all log messages whose level is between 'level1' and 'leve2' will be displayed on the provided media. The above example configuration can thus be implemented like this:

```
static const char *timestampFileFormat = "%y%m%d_%H%i";
static const char *timestampLogFormat  = "%H:%i:%s.%L";

PLogMedia media1(new LogMediaFileRotate("filename.log",
       timestampFileFormat, timestampLogFormat));
PLogMedia media2(new LogMediaConsole(timestampLogFormat));
PLogMedia media3(new LogMediaSyslog("My Application"));

TheLogMediaManager::instance()->registerMedia(
       Log::Debug, Log::Fatal, media1);

TheLogMediaManager::instance()->registerMedia(
       Log::Info, Log::Info, media2);

TheLogMediaManager::instance()->registerMedia(
       Log::Error, Log::Fatal, media3);
```

1.4 Command Line Parsing

1.4.1 Introduction

Proper command line parsing can be a tedious task. You could accept short command line flags (such as '-g') as well as long version (such as '--debug') ; You could accept any flag order ; You could produce a good help when '-h' or '--help' is used. Etc...
Usually either a tiny manual parsing is re-implemented, with very few features available (not tolerant to arguments re-ordering for instance), or a complex library is used (GNU getopt for instance). Sword provides, via the sword::CommandLineManager and the sword::CommandLineOption/sword::CommandLineFlag classes a framework for powerfull command line parsing.

1.4.2 The command line manager

```
#include <sword/sword.h>
#include <sword/base.CommandLineManager.h>
```

The whole command line parsing operation is handled via the sword::CommandLineManager class:

```
class CommandLineManager {
public:
       CommandLineManager(
              const std::string &title,
              const std::string &version,
              const std::string &copyright,
              const std::string &usage,
              int &argc,
              char **argv,
              bool fail=true);

       void help(std::ostream &os) const;

       void version(std::ostream &os) const;

       bool wantHelp() const;

       bool wantVersion() const;

       void process();

       class ParsingException : public Exception {};
};
```

First, an instance of the manager should be created, usually as a local variable. The constructor arguments are:

- title: The (short) name of the application,
- version: a version string such as “1.0”,
- copyright: a copyright notice such as “Copyright © 2003 ACME Corporation”,
- usage: a description of the usage of the command line. Usually the options handled by the command line parsing framework are described as “[OPTIONS]”,
- argc & argv: the command line. These values will be modified by the command line manager. When the parsing is finished, *argc* will contain the number of non recognised arguments, and *argv* will contain those arguments.
- fail: if 'true', the manager will fail when unknown options are encountered. If 'false', the manager will simply ignore them and leave them in the argc/argv array.

Then, command line options must be attached to the manager (see next section).

Finally the processing takes place via the 'process' method. Once processed, wantHelp() returns 'true' if '-h' or '--help' was specified, and wantVersion() returns 'true' if '--version' was specified. The two helper methods 'help()' and 'version()' display rather detailed and well-formatted help and version messages on the provided stream.

1.4.3 Declaring and testing options

Let's start with a tiny bit of terminology. We'll call a *command line flag* a parameter which can be either true (active, enabled, ...) or false (inactive, disabled, ...). We'll call a *command line option* a parameter which requires a value.

1.4.3.1 Flags

```
#include <sword/sword.h>
#include <sword/base.CommandLineManager.h>
#include <sword/base.CommandLineFlag.h>
```

A command line flag is declared via an instance of sword::CommandLineFlag. The constructor signature is as follow:

```
CommandLineFlag(
      CommandLineManager &manager,
      char shortName, const std::string &longName,
      const std::string &help, const T &value = T());
```

The arguments are:

- A reference to the command line manager,
- shortName: the flag short name, for instance 'v' for a '-v' flag. The null character '\0' can be provided if no short version of the flag is to be available,
- longName: the flag long name, for instance 'verbose' for a '--verbose' flag. An empty string can be provided if no long version of the flag is offered,
- help: a short (one-line) help string,
- value : an (optional) default value in case the option is not specified on the command line.

The CommandLineFlag is also a template class. Its 'T' parameter can be used to specify the type of the result variable. However, most of the time a simple 'bool' parameter will do, which will be 'true' if the parameter was on the command line (exemple '-v') or 'false' if the parameter was not on the command line (or disabled explicitely via '-v-' for instance).

Once the command line parsing finished (via the CommandLineManager::process method), the flag value can be obtained using the 'value' method, and the 'isSet' method checks whether the flag was specified on the command line:

```
T value() const;
```

```
bool isSet() const;
```

1.4.3.2 Options

```
#include <sword/sword.h>
#include <sword/base.CommandLineManager.h>
#include <sword/base.CommandLineOption.h>
```

A command line option is declared via an instance of sword::CommandLineOption. The constructor signature is as follow:

```
CommandLineOption(
      CommandLineManager &manager,
      char shortName, const std::string &longName,
      const std::string &help, bool acceptMultiple,
      const T &value = T());
```

The arguments are similar to the CommandLineFlag ones, except 'acceptMultiple'. If 'true' several values can be specified on the command line, whereas only one is allowed if 'false'.

The CommandLineOption is also a template class. Its 'T' parameter defines the type of the result values. After the command line parsing the number of values set is in size(), the values can be obtained via value() and retrieved all at once in a vector using values():

```
word size() const;
T value(word i=0) const;
bool isSet() const;

typedef std::vector<T> Values;
const Values &values() const;
```

1.4.3.3 Result types

Any result type 'T' given as a template parameter to CommandLineFlag or CommandLineOption as long as a template specialization of the trait class CommandLineTraits is available. This specialization provides to the framework implementations for two methods on the type 'T':

```
static T    convert(const std::string &value);
static void toggle(T &value, bool onoff);
```

The 'convert' method translates the string read on the command line into a value of type 'T' (used for options). The 'toggle' method switches the value from on to off (used for flags).
Specializations for CommandLineTraits are provided by default for the common types : **std::string**, **int** and **bool**.

1.4.4 Example

As an exemple, we will use the framework to parse a command line which accepts:

- An 'output file' specification via '-o' or '--output',
- A 'debug level' specification via '-d' or '--debug',
- A 'verbose' flag via '-v' or '--verbose',
- A set of file names
- '-h' or '—help' displays some help,
- '-v' or '—version' displays the application version id.

The header inclusions for this example should look like:

```
#include "sword/sword.h"
#include "sword/base.CommandLineManager.h"
using namespace sword;
```

The command line manager instanciation provides the program name, version and copyright as well as part of the help information:

```
int main(int argc, char *argv[])
```

```
{
    // Create the manager
    CommandLineManager clm(
        "Foo", "1.0", "Copyright (C) ACME Corporation",
        "[OPTIONS] FILES", argc, argv);
```

The command line options and flags are created and registered into the command line manager:

```
    // Create the options and flags to recognize
    CommandLineOption<std::string> cl_o(
       clm, 'o', "output",
       "Output File Name", false);
    CommandLineOption<int>         cl_d(
       clm, 'd', "debug",
       "Debug Level",      false, 1);
    CommandLineFlag  <bool>        cl_v(
       clm, 'v', "verbose",
       "Switch to Verbose Level", false);
```

The processing is launched, help and version messages are displayed if required:

```
    // go and process the command line
    clm.process();

    // display the help if it was required
    if (clm.wantHelp())    { clm.help(std::cout);    return 1; }
    if (clm.wantVersion()) { clm.version(std::cout); return 1; }
```

Finally, the options are checked and used:

```
    // check some more conditions
    if (!cl_o.isSet())
    {
      std::cout << "Missing '-o' argument, try --help";
      return 1;
    }
    if (argc == 1)
    {
      std::cout << "Missing files arguments, try --help";
      return 1;
    }

    // use the results
    std::string verboseStr = (cl_v.value() ? "on" : "off");
    std::cout << "Output file is : " << cl_o.value() << std::endl;
    std::cout << "Debug level is : " << cl_d.value() << std::endl;
    std::cout << "Verbose is     : " << verboseStr << std::endl;
    std::cout << "Files are      : ";
    for(int i=1; i<argc; ++i)
       std::cout << argv[i] << ", ";
    std::cout << std::endl;

    return 0;
}
```

This program, if run with '--version' will display the version id string:

```
Foo, version 1.0
Copyright (C) ACME Corporation
```

And if run with '-h' or '--help' will display the help message:

```
Foo, version 1.0
Copyright (C) ACME Corporation

Usage: sb [OPTIONS] FILES

    -h, --help          Show this help
        --version       Show version information
    -o, --output=VALUE  Output File Name
    -d, --debug=VALUE   Debug Level
    -v, --verbose       Switch to Verbose Level
```

1.5 Time & Date

1.5.1 Introduction

The “Time & Date” part of Sword allows to you to:

1. Very efficiently (time & space) manipulate a timestamp (**sword::Time**),
2. Manipulate a date with all its components (day, month, year...), display it, parse it from some text (**sword::Date**),
3. Convert from a Time to a Date and vice versa in various cultural referentials (Gregorian, Julian), using the **sword::Calendar** set of classes,
4. Manage a set of non-working days in an optimised storage (**sword::HolidayRepository**),
5. Generate a holiday repository from a set of human readable, high level, rules such as “1 mon after easter_western” for easter monday (**sword::HolidayRule**).

Great care has been taken on two fronts when creating this module:

1. **Performance**. The clean isolation between sword::Time and sword::Date allows the programmer to perform the costly “day/month/year/... <-> timestamp serial” conversions less often. Also, most of the time only a sword::Time is passed around in the code, with minimal footprint (it is equivalent in performance to an int64 or a double),
2. **Precision**. The subtle cultural implications of date manipulations are cleanly implemented via the Calendar framework. Also, the working interval for timestamp is a lot less restricted than on usual time/date implementations (about 290 million years before and after chirst in SWORD vs. 1970 to 2038 in the C library for instance).

1.5.2 sword::Time

```
#include <sword/sword.h>
#include <sword/time.Time.h>
```

A sword::Time holds nothing but a point in time (also called timestamp). The precision is the millisecond. Whenever a timestamp is created it is marked invalid (it is not initialized by default to the current time!).
Internally, the storage is a 64-bit signed integer which represents the number of milliseconds since an arbitrary origin (1 Jan 1970 00:00:00 UTC). It can be either positive (point in time after this origin) or negative (point in time before this origin). The authorised dates are approximately in a 292 million years interval backward and forward the origin. There is one special unauthorised value (-2^63-1, the minimum 64-bit signed integer value), which is used to mark an invalid date.

1.5.2.1 Construction

A timestamp can be instanciated either with the invalid value or with a given integer “serial” timestamp using one of the constructors:

```
Time() throw();
Time(int64 value) throw();
```

An invalid timestamp or a timestamp initialized to the current time can be obtained using three static methods. now() computes the current local time (i.e. same as on your wrist watch), whereas nowUTC() computes the current universal time (i.e. the time in the greenwitch time zone). The additional timeZone() method returns the offset (time span) between the local time and the universal time (the formula is “UTC + Time Zone = Local Time”) :

```
static Time invalid();
static Time now();
static Time nowUTC();
static TimeSpan timeZone();
```

1.5.2.2 Attributes

A timestamp value can be altered or obtained using three different attributes:

1. **Serial**: the integer serial (number of milliseconds since the origin),
2. **Integer1900**: number of days since 1/1/1900,
3. **Double1900**: number of days since 1/1/1900 in the natural part, fraction of day in the decimal part (.0 = 0h in the morning, .5 = 12 noon, .9999 = reaching midnight).

Those attributes all have read and write accessors:

```
int64 serial() const throw();
Time& serial(int64 value) throw();

int  integer1900() const throw();
Time& integer1900(int value) throw();

double double1900() const throw();
Time& double1900(double value) throw();
```

For convenency, the write accessors also return a reference to the time itself, thus enabling several modifications in one call, for instance:

```
myTime.double1900(37944.63).truncMinute();
```

1.5.2.3 Validity

The validity of a time can be tested and altered:

```
bool valid() const throw();
void invalidate() throw();
```

1.5.2.4 Timestamp arithmetics

The **sword::TimeSpan** represents a time interval (duration), and can be used in simple timestamp arithmetics:

```
friend TimeSpan operator-(const Time& lhs, const Time &rhs);
friend Time operator+(const Time& lhs, const TimeSpan &rhs);
friend Time operator+(const TimeSpan &rhs, const Time& lhs);
```

Times can be adjusted to the nearest second, minute, hour or day. For example, “31may2002 6:53:51” rounded to the minute will be “31may2002 6:54:00”:

```
Time& roundDay() throw();
Time& roundHour() throw();
Time& roundMinute() throw();
Time& roundSecond() throw();
```

Times can also be truncated to the greatest but lower second, minute, hour or day. For example, “31may2002 6:53:51” truncated to the minute will be “31may2002 6:53:00”:

```
Time& truncDay() throw();
Time& truncHour() throw();
Time& truncMinute() throw();
Time& truncSecond() throw();
```

1.5.2.5 Ordering and streaming

sword::Time supports standard comparison operators as well as streaming onto the sword streams framework:

```
bool operator< (const Time &rhs) const;
bool operator<=(const Time &rhs) const;
bool operator> (const Time &rhs) const;
bool operator>=(const Time &rhs) const;

bool operator==(const Time &rhs) const;
bool operator!=(const Time &rhs) const;

friend Istream &operator>>(IStream &lhs, Time &rhs);
friend OStream &operator<<(OStream &lhs, const Time &rhs);
```

1.5.3 sword::Date

```
#include <sword/sword.h>
#include <sword/time.Date.h>
```

A sword::Date holds a timestamp split into its components (day, month, year, hour, minute, second, millisecond). The actual meaning of those components depends on the cultural conventions. For instance, occidental countries use the Gregorian calendar whereas some oriental countries use the Orthodox (Julian) calendar. Those calendars differ only in their definition of leap years, but over the centuries the gap increases (it is now 13 days).
So whenever you need to convert from a sword::Time to a sword::Date or vice versa, an implementation of the sword::Calendar interface is necessary. Two of those are currently available:

- **sword::CalendarGregorian**: available via the global variable TheCalendarGregorian, it implements the Gregorian rules which are valid in most western countries since 1582 (a year is a leap year if it is multiple of 4, but not multiple of 100 except if it is also a multiple of 400),
- **sword::CalendarJulian**: available via the global variable TheCalendarJulian, it implements the Julian rules which has been valid until 1582 in western countries and is still used in oriental countries (it is also known as the Orthodox calendar, and a year was a leap year only if it is divisble by 4).

In order to maintain consistency, once a date has been initialised with a Calendar, it keeps a reference to it and all other manipulations will use it. However, for simplicity and code clarity most operations are also available without Calendar, and the Gregorian one is used by default.

1.5.3.1 Construction / conversions

A Date can be instanciated empty, from its components or from a timestamp. If a calendar is not provided, the Gregorian calendar is used by default. If the hour/minute/second/millis is not specified, “0:0:0” is assumed.

```
Date() throw();
Date(const Date &rhs) throw();
Date &operator=(const Date &rhs) throw();

Date(word day, word month, word year, word hour, word minute, word
second, word millis, const Calendar &calendar) throw();
Date(word day, word month, word year, word hour, word minute, word
second, word millis) throw();

Date(word day, word month, word year, const Calendar &calendar) throw
();
Date(word day, word month, word year) throw();

Date(Time time, const Calendar &calendar);
Date(Time time);
```

Additionally, a date can be converted to/from other formats:

```
void fromTime(Time time, const Calendar &calendar);
void fromTime(Time time);
Time toTime() const;
```

It is also possible to obtain a date initialised to the current local time (i.e. same as on your wrist watch) with now(), whereas nowUTC() computes a date initialised to the current universal time (i.e. the time in the greenwitch time zone):

```
static Date now(const Calendar &calendar);
static Date now();
static Date nowUTC(const Calendar &calendar);
static Date nowUTC();
```

The “packed integer” date representation is also available. It consists of the date only (no hour) packed into an integer with the 'yyyymmdd' format. For instance, 8th november 1972 is packed into the integer 19721108.

```
Date(int packed, const Calendar &calendar);
Date(int packed);
void fromPackedDate(int packed, const Calendar &calendar);
void fromPackedDate(int packed);
int toPackedDate() const;
```

1.5.3.2 Attributes

A sword::Date exposes those attributes:

- Day: the day of the month. From 1 to 31 in usual calendars.
- Month: the month of the year. From 1 to 12 in usual calendars.
- Year: the year. 2003 for instance.
- Hour: the hour of the day. From 0 to 23.
- Minute. The minute within the hour. From 0 to 59.
- Second. The second within the minute. From 0 to 59.
- Millis. The millisecond within the second. From 0 to 999.

Those attributes all have read and write accessors:

```
// --- Read Accessors
word day() const throw();
word month() const throw();
word year() const throw();
word hour() const throw();
word minute() const throw();
word second() const throw();
word millis() const throw();

// --- Write Accessors
void day(word value) throw();
void month(word value) throw();
void year(word value) throw();
void hour(word value) throw();
void minute(word value) throw();
void second(word value) throw();
void millis(word value) throw();

// --- Composite write accessors
void set(word day, word month, word year,
         word hour, word minute, word second, word millis,
         const Calendar &calendar) throw();
void set(word day, word month, word year,
         const Calendar &calendar) throw();
```

Some additional read-only, computed attributes are available:

- WeekDay: the day of the week, 0 for Sunday up to 6 for Saturday
- WeekNumber: the week number. From 1 to 52.
- DayWithinYear: the day within the year. From 1 to 366.

```
word weekDay() const;
word weekNumber() const;
word dayWithinYear() const;
```

1.5.3.3 Validity

The existence and validity of a sword::Date can be tested and altered. An empty date is a special value where all components are zero. An invalid date is a date which does not mean anything in its calendar (for instance 31 feburary...):

```
bool empty() const throw();
void clear() throw();
bool valid() const;
```

1.5.3.4 Simple Arithmetic

A few simple date arithmetic operations are available. You can add years, months, days, hours, seconds or milliseconds to a date (and these are overlapping operations, meaning that adding 2 days to the 30th day of the month may lead to the 1st or 2nd day of the next month). When adding months, you can also specify whether you want to maintain the end-of-month (endToEnd=true) so that adding one month to 30th april will bring you to 31st may.
Finally, prev() brings you to the previous day while next() computes the next day.

```
void addYears(int years, bool endToEnd = false);
void addMonths(int months, bool endToEnd = false);
void addDays(int days);
void addHours(int hours);
void addMinutes(int minutes);
void addSeconds(int seconds);
void addMillis(int millis);
void next();
void prev();
```

1.5.3.5 String output and parsing

A date can be printed using a custom format string, and the textual representation can later be parsed to recreate a date. Those two operations use the following 'formatters' within their format string (any character not recognised as a formatter is directly copied into the output text or skipped during parsing):

- %a : 'am' or 'pm'
- %A : 'AM' or 'PM'
- %d : Day of the month, 2 digits (01..31)
- %D : Day of the week, textual, 3 letters (eg. Fri)
- %F : Month, textual, long (eg. January)
- %g : Hour, 12-hour cycle, no leading 0 (1..12)
- %G : Hour, 24-hour cycle, no leading 0 (1..24)
- %h : Hour, 12-hour cycle, 2 digits (01..12)
- %H : Hour, 24-hour cycle, 2 digits (01..24)
- %i : Minutes, 2 digits (00..59)
- %j : Day of the month, no leading 0 (1..31)
- %l : Day of the week, textual, long (eg. Friday)
- %L : Milliseconds (000...999)
- %m : Month, 2 digits (01..12)
- %M : Month, textual, 3 letters (eg. Jan)
- %n : Month, no leading 0 (1..12)
- %O : Different to greenwitch, in hours (eg. +0200)
- %s : Seconds, 2 digits (00..59)
- %S : English day suffix (st, nd, rd or th)
- %w : Day of week, numeric (0=Sunday to 6=Saturday)
- %W : Week number, numeric (1..52)
- %Y : Year, 4 digits (eg. 1984, 2003)
- %y : Year, 2 digits (eg. 84, 03)
- %% : Print the '%' character

The two methods using a format string are:

```
std::string toString(const std::string &format) const;
void fromString(const std::string &format, const std::string &data);
```

1.5.3.6 Ordering

sword::Date also supports standard comparison operators :

```
bool operator< (const Date &rhs) const;
```

```
bool operator<=(const Date &rhs) const;
bool operator> (const Date &rhs) const;
bool operator>=(const Date &rhs) const;

bool operator==(const Date &rhs) const;
bool operator!=(const Date &rhs) const;
```

1.5.4 Holidays management

Sword is also good at managing sets of working/non-working days (holidays definitions). Some software programming tasks require that the program is aware of holidays (for instance financial computation programs may need to compute “payment schedules” and adjust the actual payment dates so that they always fall on business days, or a graphical calendar/agenda program may display holidays in a different color).

1.5.4.1 The Holidays Repository

```
#include <sword/sword.h>
#include <sword/time.HolidayRepository.h>
```

A sword::HolidayRepository maintains a set of working / non-working days under a common name. As usually the definition of holidays is country-dependant, the programmer will create one HolidayRepository instance for each country handled by his program.

When creating a repository, a name is provided, which should be a unique identifier for this repository (country id for instance). When the 'haveWeekEnds' flag is true will, saturdays and sundays are automatically flagged as non-working. The 'cacheYearFrom' and 'cacheYearTo' values define a year interval [from..to] in which the working / non-working lookup is optimised. It is to note that the memory used by the HolidayRepository is proportional to
roughly 45 byte per year in this interval, so when you initialize your HolidayRepository instance, you have to balance access speed and memory footprint.

```
HolidayRepository(const std::string &name,
                  bool haveWeekEnds,
                  word cacheYearFrom, word cacheYearTo);
```

Once created and initialised using rules (see next paragraph), the repository can be queried using one of the two 'isHoliday' methods:

```
bool isHoliday(const Time &time) const;
bool isHoliday(const Date &date) const;
```

1.5.4.2 Feeding a repository with individual days

Within the dates interval defined by the 'cacheYearFrom' and 'cacheYearTo' values, the holidays repository keeps track of the working / non-working status of each day. It is possible to manually set the status of one specific day in this interval using the 'addToCache' method:

```
void addToCache(const sword::Time& time);
void addToCache(int integer1900);
```

1.5.4.3 Feeding a repository with rules

```
#include <sword/sword.h>
#include <sword/time.HolidayRule.h>
```

Within one holiday repository, the definition of which days are working or non-working uses textual rules close to the natural (english) language. Here is a set of examples of working rules which defines well-known non-working days:

```
FR New year's day          rule = "1 jan"
FR Easter sunday           rule = "easter_western"
FR Easter monday           rule = "1 mon after easter_western"
```

```
US Memorial day            rule = "1 mon before 1 jun"
US Independance Day        rule = "4 jul"
US Independance Day (bis)  rule = "1 after 4 jul if_sun"
US Independance Day (ter)  rule = "2 after 4 jul if_sat"
GR Orthodox Easter monday  rule = "1 mon after easter_orthodox"
```

The general syntax for a rule is very simple. It consists of:

1. An optional offset, either as a number of day ('2 after') or as a number of day-in-the-week ('1 mon after'),
2. An absolute rule, either as a date ('4 jul') or as a special keyword ('easter_western'),
3. A day of week filter so that the rule is valid only on specific days of the week ('if_sun'),
4. An application range, so that the rule is valid only in a given date interval ('from 5 jan 1994')

These special keywords represent all the special cultural date events within a year:

- easter_western: Easter Sunday in western (christian mainly) cultures,
- easter_orthodox: Easter Sunday in orthodox religions (based on the Julian calendar),
- vernal_equinox: Date of the vernal equinox,
- automnal_equinox: Date of the automnal equinox,
- winter_solstice: Date of the winter solstice,
- summer_solstice: Date of the summer solstice,
- chinese_newyear: Date of the chinese new year (second new moon after the winter solstice).

A sword::HolidayRule instance is created with a name and the textual rule. During the instanciation the text is parsed, so any error according to the recognised grammar will raise a 'HolidayRuleParsingException' error.

```
HolidayRule(const std::string &name, const std::string &rule);
```

Once created, the rule can be queried. For a given year, the rule will answer either 'false' (there is no holiday matching this rule within this year), or 'true' (there is one holiday this year) and the exact date of the holiday:

```
bool compute(word year, Time &time) const;
```

But most of the time the HolidayRule is only used to initialize a holiday repository:

```
HolidayRepository hr(“FR”, true, 1970, 2020);

hr.addRule(
  HolidayRule(“New Year”, “1 jan”));
hr.addRule(
  HolidayRule(“Easter Monday”, “1 mon after easter_western”));
```

As a reference, the (yacc compatible) grammar for the rules is:

```
rule
       : dayOfWeekOffset absoluteRule fromRange untilRange
       | dayOffset       absoluteRule fromRange untilRange
       |                 absoluteRule fromRange untilRange
       ;

dayOfWeekOffset
       : NUMBER DAYOFWEEK DIRECTION
       ;

dayOffset
       : NUMBER DIRECTION
       ;

absoluteRule
       : NUMBER MONTH year dayOfWeekFilter
       | SPECIAL           dayOfWeekFilter
       ;

dayOfWeekFilter
       : /* empty */
```

```
        | IF_DAYOFWEEK
        ;

year
        : /* empty */
        | NUMBER
        ;

fromRange
        : /* empty */
        | FROM NUMBER MONTH NUMBER
        ;

untilRange
        : /* empty */
        | UNTIL NUMBER MONTH NUMBER
        ;
```

1.5.4.4 The Holidays Manager

```
#include <sword/sword.h>
#include <sword/time.HolidayManager.h>
```

The Holidays Manager can be used as a central storage for all your holiday repositories. It is a singleton where repositories can be accessed from their names. The 'insert' method is used to add repositories to the manager, whereas the 'get' method access them:

```
PHolidayRepository get(const std::string &name) const;
void insert(PHolidayRepository hr);
```

It will typically be used this way:

```
// --- initialization phase

PHolidayRepository hr;

// Add 'FR' holidays
hr = new HolidayRepository(“FR”, true, 1950, 2020);
hr->addRule(HolidayRule(“...”, “...”));
...
TheHolidayManager::instance()->insert(hr);

// Add 'US' holidays
hr = new HolidayRepository(“US”, true, 1950, 2020);
hr->addRule(HolidayRule(“...”, “...”));
...
TheHolidayManager::instance()->insert(hr);

// --- using the manager
Time time = .....;

// is 'time' a holiday in France ?
if (TheHolidayManager::instance()->get(“FR”)->isHoliday(time))
{
  // yes!
}
```

1.5.5 Date arithmetics

Many applications (especially in the financial domain : accounting and trading) need to perform more accurate date arithmetics than the plain “timestamp difference” as found in sword::Time or “addMonth” as found in sword::Date.

1.5.5.1 sword::DateOperation

A DateOperation allows to apply a more complex date arithmetic operation onto a date. The operation is usually defined using a string representation. For format for this “date operation description” is made of a delay, an adjustement and a holiday repository.

The delay can be one of ('999' can be replaced by any integer value):
1. 999d : Adds 999 calendar days, which are just regular days with no difference whether they are opened (business) days or closed (vacancy) days,
2. 999bd : Adds 999 business days,
3. 999m : Adds 999 months using the “end-to-end” convention, which means that if the start date is at the end of a month, the target will be at the end of a month (example: 30-apr-2003 plus “3m” gives 31-jul-2003),
4. 999ms : Adds 999 months using the “short” convention, which means that the day is kept as it is (and the delay is sometimes shorter than with the “end-to-end” convention). (example: 30-apr-2003 plus “3ms” gives 30-jul-2003),
5. 999y : Adds 999 years using the “end-to-end” convention,
6. 999ys : Adds 999 years using the “short” convention,
7. 999w : Adds 999 weeks (one week is just 7 calendar days)

The adjustement (or business day convention 'BDC') can be one of:
- mf : “modified following”. If the end date is a holiday, skip to the next business day unless it would change month, in which case skip to the previous business day.
- nb : “next business day” : If the end date is a holiday, skip to the next business day.
- pb : “previous business day” : If the end date is a holiday, skip to the previous business day.

Finally, the holidays repository specification connects a date operation to a set of closed/business days. By default, if no holidays repository is provided, only the week-ends will be considered as closed.

Those three elements (delay, adjustement, repository) can be assemble in any of those syntaxes to form the operation description:
1. delay[adjustment][:holiday]
2. delay[,adjustment][:holiday]
3. adjustment[:holiday]

A DateOperation can be created either empty or from an operation description:

```
DateOperation();
DateOperation(const std::string& description);
```

It can be cleared, and can be tested to see if it's empty or partially empty:

```
bool empty() const;
bool emptyDelay() const;
bool emptyBdc() const;
void clear();
```

It can be in a special “invalid” state, which can be tested. It is an error to try to apply an invalid date operation onto a date:

```
bool valid() const;
void invalidates();
```

A DateOperation exposes the following attributes:
- Days : number of days in the '999cd' delay or 0 if it was not specified,
- WorkDays : number of days in the '999bd' delay or 0 if it was not specified,
- Months : number of months in the '999s' delay or 0 if it was not specified,
- MonthsEndToEnd : boolean : is the month convention “end-to-end”,
- Years : number of years in the '999y' delay or 0 if it was not specified,
- YearsEndToEnd : boolean : is the years convention “end-to-end”,
- BDC : the adjustement or NONE if it was not specified,
- Holidays : the holidays repository.

Those attributes all have read and write accessors:

```
// --- Read Accessors
int  days() const;
bool workDays() const;
int  months() const;
bool monthEndToEnd() const;
int  years() const;
bool yearEndToEnd() const;
BDC  bdc() const;
PHolidayRepository holidays() const;

// --- Write Accessors
void days(int days);
void workDays(bool workDays);
void months(int months);
void monthEndToEnd(bool endToEnd);
void years(int years);
void yearEndToEnd(bool endToEnd);
void bdc(BDC bdc);
void holidays(const std::string& holidaysName);
void holidays(PHolidayRepository holidays);
```

You can apply this operation to a date, or selectively apply the delay and the adjustment:

```
void apply(Date& date, int times=1) const;
void applyDelay(Date& date, int times=1) const;
void applyBdc(Date& date) const;
```

1.6 Database access

There are not so many C++ database access libraries out there, and none of the production-proof ones have both a simple interface and a sufficient set of features. Sword database interface does not intend to answer all database client needs. If you need to code very precise and tuned database client behaviour, you should use the native API for your database provider anyway. This library rather provides a very simple and effective interface, which allows you to:

- connect a database using one of the provided drivers,
- define a textual query, which can range from simple SQL to stored procedure calls,
- retrieve and navigate the query result, including multiple-sets results.

It won't allow you to do very fancy stuff like:

- bind in or out parameters of stored procedures to variables,
- name stored procedures parameters,
- random navigation within result sets (only forward navigation supported).

1.6.1 Database Connection

```
#include <sword/sword.h>
#include <sword/db.DbConnection.h>
```

Database connections are created via a factory static method of sword::DbConnection. In order to get your new connectio object, you have to provide a driver name and name for this connection. The name is just for your conveniency, so you can name it something meaningfull for your program (it will be reported in error messages).

```
PDbConnection factory(
  const std::string &driver, const std::string& name);
```

Now your connection object is created, but no database has been opened yet. You can open or close the connection using the following methos. The parameters string is in format which is driver-dependant, so be sure to read the chapter about the driver you want to use below.

```
void open(const std::string& parameters);
void close();
```

From the connection you can obtain a new query object:

```
PDbQuery query();
```

Finally, transactions are controlled on an opened connection using:

```
void begin();
void commit();
void rollback();
```

1.6.2 Database Query

```
#include <sword/sword.h>
#include <sword/db.DbQuery.h>
```

DbQuery objects are never created directly, but rather obtained from an opened connection via the query() method. SQL statements which are executed via a DbQuery object can be either a 'modify' command (UPDATE, INSERT) a 'select' command (SELECT), or a stored procedure call. The exec() method will return true if some data is available (SELECT result sets for instance), or false otherwise (no result on INSERT for instance).

```
bool exec(const std::string& sql, bool reportWarningsAsErrors);
```

By default, even database warnings are reported as errors (it throws DbException). If reportWarningAsErrors is set to false, warnings will be silently ignored.

1.6.2.1 Result sets navigation

The data returned is organised as cells within rows within sets. Each set can be represented as a rectangular array of cells. A set can also be empty.

Just after the query, the first result set is implicitely selected. As result sets can be empty, the first row is never implicitely selected. So the first thing to do is to call nextRow() to fetch the first row of data within this set. The result of this operation can be tested to see whether the set was empty or not.

So the two navigation methods are:

```
bool nextRow();
bool nextSet();
```

Then the nagivation uses nextSet() to go forward one set, and nextRow() for one row. Those two methods return true if some data is available, or false if the end of the results have been reached already.

A typical navigation would look like:

```
PdbQuery query = ...;
query->exec("... SQL STATEMENT ...");

int set = 0;
do
{
  ++set;
  int row = 0;
  while(query->nextRow())
  {
    ++row;
    //
    // ... do stuff with data on this row
    //
    std::cout << "Set " << set << ", Row " << row << std::endl;
  }
}
while(nextSet());
```

1.6.2.2 Data retrieval

Once the required row of the required set has been selected using data navigation, individual cells can be accessed, either by column number (0-based) or by column name. First you can query the number of columns within this result set:

```
int columns();
```

Then each data type has its own accessor:

```
int getInt(word column);
int getInt(const std::string& column);

double getDouble(word column);
double getDouble(const std::string& column);

std::string getString(word column);
std::string getString(const std::string& column);

Time getTime(word column);
Time getTime(const std::string& column);

Variant get(word column) = 0;
Variant get(const std::string& column) = 0;
```

1.6.3 ODBC driver

The sword ODBC database driver is available if the HAVE_DB_ODBC compilation flag has been specified in the config.h main configuration header. It requires ODBC3 compatible libraries (these are part of the standard SDK under MsWindows, you may require an additional library under other operating systems).

You create an ODBC database connection object using the factory with “ODBC” for driver:

```
PDbConnection con = DbConnection::factory(“ODBC”, “My Connection”);
```

Then you open the connection. The parameter string must be a valid data source name:
1. A user or system-DSN name,
2. A path to a file-DSN,
3. a stringified-DSN, which syntax depends on the actual driver you are using.

For instance, for connecting a MySQL database, you can use the following code:

```
std::string parametes =
  “DRIVER={MySQL ODBC 3.51 Driver};”
  “SERVER=localhost;”
  “USER=root;”
  “PASSWORD=;”
  “DATABASE=test”;
con->open(parameters);
```

1.6.4 SQLITE driver

The sword SQLITE driver is available if the HAVE_DB_SQLITE compilation flag has been specified in the config.h main configuration header. SqLite is a “database in a file” library: it allows you to store data in a disk file using SQL statements. SqLite home page is at http://www.hwaci.com/sw/sqlite/.

You create a SQLITE database connection object using the factory with “SQLITE” for driver:

```
PdbConnection con = DbConnection::factory(“SQLITE”, “My Connection”);
```

Then you open the connection. The parameter string must be the path to the database file. For instance:

```
con->open("/home/user/data/hisdatabase.dat");
```

1.7 Lexical Cast

```
#include <sword/sword.h>
#include <sword/base.LexicalCast.h>
```

The C library used to provide a few very usefull functions to convert characters strings into primitive types and vice versa (atof, atod, printf...). Those functions are cumbersome to use in C++ because they work on "char*" rather than "std::string" and require manual character buffer management (allocation, overflow detection, ...).

The usual C++ alternative is to use std::ostringstream and std::istringstream, but these are very heavy (both in term of API and performance wise).

Sword fills the gap and offers a very convenient cast operator in order to transform from textual representations to primitive data types and vice versa : the lexical_cast<>. The general syntax is modeled after the usual C++ casts:

```
TARGET target = lexical_cast<TARGET>(const SOURCE& source);
```

The simple operations of writing an integer to a string and conversely parsing an integer from a string are thus written:

```
int value = 5;
std::string str = lexical_cast<std::string>(value);
// now str == "5"

int value2 = lexical_cast<int>(str);
// now value2 == 5
```

Note: the lexical_cast<> concept has first been invented by Kevlin Henney for the Boost library.

1.7.1 Formatting

Converting a primitive type into a std::string may require to specify formatting details (such as alignment, precision, so on so forth). The lexical_cast<> operator optionally takes a LexicalFormatter as its second argument:

```
LexicalFormatter formatter("format");
std::string    str    =    lexical_cast<std::string>(SOURCE    source,
formatter);
```

The formatter is initialized with a textual "printf-like" description of the format specification. It can be shared amongst several formatting, or simply initialized upon each conversion (in which case a simplified notation allows just to give the formater specification string as the second argument):

```
// shared formatter
LexicalFormatter formatter("=8d");
std::string s1 = lexical_cast<std::string>(v1, formatter);
std::string s2 = lexical_cast<std::string>(v2, formatter);

// single-usage formatter
std::string   s3   =   lexical_cast<std::string>(v3,   LexicalFormatter
("d"));
std::string   s4   =   lexical_cast<std::string>(v4,   LexicalFormatter
("X"));

// shorter notation for the single-usage formatter
std::string s5 = lexical_cast<std::string>(v5, "d");
std::string s6 = lexical_cast<std::string>(v6, "X");
```

While providing full customization of formatting (with even more options than the traditionnal printf), the lexical cast implementation is type safe (if you pass a double to a int formatter a LexicalFormatterException exception is thrown), and memory safe (memory buffers of sufficient size are automatically allocated for you).

Finally, the target type for lexical_cast formatting can be any class which can be built from a “const char*” zero-terminated buffer. Usually std::string will be used, but any other class which implements this contract is suitable. A good example is that users of the QT library can use the lexical cast to feed QString directly:

```
QString s = lexical_cast<QString>(value);
```

1.7.1.1 Integer formats

The general syntax for an integer format specification is:

```
[flags][width][.precision][type]
```

With “flags”:

+	For signed integers, print a plus sign if positive
(	For signed integers, display negative values in parenthesis
#	Prefix the octal and hexadecimal representation with '0' (octal) or '0x' (lower case hexadecimal) or '0X' (upper case hexadecimal)
'	Separate digits in groups
<	Left align within field
=	Center within field
>	Right align within field

With “width”:

The minimum width of the field

With “precision”:

The minimum number of significant digits

With “type”

d	Signed integer, decimal representation
u	Unsigned integer, decimal representation
x	Lower-case hexadecimal representation
X	Upper-case hexadecimal representation
o	Octal representation

Examples:

Format	Value	Result
d	12485	“12485”
8d	12485	“12485 ”
<8d	12485	“ 12485”
=8d	12485	“ 12485 “
(d	-12485	“(12485)”
('d	-12485	“(12_485)”
('.6d	-12485	“(012_485)”
('12.6d	-12485	“(012_485) ”

1.7.2 Parsing

... to be documented ...

... basically the format string for parsing is same as for printing ...

1.8 String and STL tools

1.8.1 String tools

```
#include <sword/sword.h>
#include <sword/base.StringTools.h>
```

The C++ Standard Library std::string is a powerfull and efficient characters string container, but lacks some important manipulation tools. Sword “string tools” fills the gap using a pragmatic approach : a set of global functions.

1.8.1.1 Simple string manipulations

The 'trim' operations get rid of the spacing characters (actual 'space', tabulation, end of line), from the left end of the string, its right end or both. Those functions do not modify the string, but rather return a modified copy:

```
string trimLeft (const string &str);
string trimRight(const string &str);
string trim     (const string &str);
```

The 'strlwr' and 'strupr' operations modify the case of a string. 'strlwr' like its C equivalent converts all upper-case characters to lower-case equivalents. 'strupr' converts lower-case characters to upper-case. Those functions exist in two flavours: one operates directly on the provided string (in-place operation), whereas the other returns a modified copy:

```
void   strlwr(string &str)
void   strupr(string &str)
string strlwr(const string &str)
string strupr(const string &str)
```

The 'stricmp' and 'strnicmp' operations have the same behaviour as the C 'strcmp' and 'strncmp' except that they are not case sensitive. So that 'A Message' will match 'a message' with the 'i' version of those operations:

```
int stricmp (const string &s1, const string &s2);
int strnicmp(const string &s1, const string &s2, size_t length);
int stricmp (const char *s1, const char *s2);
int strnicmp(const char *s1, const char *s2, size_t length);
```

The 'replace' family of functions provides extended search-and-replace operations on strings. 'replace' allows a one-occurence replacement, whereas 'replacem' allows a replacement of all occurences of the searched string fragment (multiple replacements):

```
void replace (string &str, const char *before, const char *after);
void replace (string &str, const string &before, const string
&after);

void replacem(string &str, char before, char after);
void replacem(string &str, const char *before, const char *after);
void replacem(string &str, const string &before, const string
&after);
```

The 'split' operation cuts a long string into a vector of smaller strings using a single-character separator. For instance the string 'label;3.5;value2' can be split into the vector ('label', '3.5', 'value2'). In the first version of this function, the target vector must be provided as an argument and the segments found in the input strings are added to it (the vector is not cleared beforehand, so that you can add more and more segments to a single vector using several consecutive 'split' calls). A second flavour returns a brand new vector with the split content:

```
void split(const string &str,
           std::vector<string > &segments, char separator);

std::vector<string>
     split(const string &str, charT separator);
```

The exact opposite of 'split' is 'join', which joins the elements from a vector into a single string, inserting the specified separator between the elements:

```
std::string join(const std::vector<std::string> &segments,
                 char separator)
```

The 'extractNameValue' performs the common operation of cutting a name/value pair string such as 'aname=avalue' into the name 'aname' and the value 'avalue':

```
void extractNameValue(const string &line,
                      string &name, string &value,
                      char separator = '=');
```

1.8.1.2 Parsing simple types

The C library used to provide a few functions to convert characters strings into simple types (atof, atod...). Sword extends this concept and provides a set of 'parse' operations which convert strings into all kind of integers and floating-point types:

```
int8   parseInt8  (const std::string &str);
int16  parseInt16 (const std::string &str);
int32  parseInt32 (const std::string &str);
int    parseInt   (const std::string &str);

word8  parseWord8 (const std::string &str);
word16 parseWord16(const std::string &str);
word32 parseWord32(const std::string &str);
word   parseWord  (const std::string &str);

float  parseFloat (const std::string &str);
double parseDouble(const std::string &str);
```

The error handling of those functions is very primitive. In situations where the string cannot be interpreted as a value of the required type, '0' is returned.

1.8.1.3 Displaying simple types

A full set of 'format' operations can be used to convert numeric values to their string representations with a variety of display options.

For **floating point numbers**, two very simple functions use a default format which should be convenient for most displays:

```
string format(float value);
string format(double value);
```

But for more precise formatting the following options are available:

- 'precision': how many digits after the decimal point,
- 'decimalChar': which character to use as the decimal separator,
- 'thousandChar': which character to use as the thousand separator ('\0' for none),
- 'negativeInParenthesis': display negative number with a minus sign or within parenthesis.

```
string format(double value,
              word precision,
              char decimalChar = '.',
              char thousandChar = '\0',
              bool negativeInParenthesis = false);
```

Examples of floating point formatting using this functions are:

```
format(3.14159, 2)                   ==> "3.14"
format(3.14159, 3, ',')              ==> "3,142"
format(12548.21, 2, '.', ' ')        ==> "12 548.21"
format(-12548.21, 2, ',', ' ', true) ==> "(12 548,21)"
```

Characters and **pointers** can also be printed:

```
string format(void *value);
string format(char value);
```

1.8.2 std::vector<> tools

```
#include <sword/sword.h>
#include <sword/stltools.Vector.h>
```

Using this header, it becomes possible to stream any std::vector either as text (on a std::ostream) or in the Sword streams framework (sword::OStream and sword::IStream). The only condition is that the underlying data type of the vector instance is also streamable on the same stream:

1. std::ostream. The textual format used is '<value1, value2, value3>',
2. sword::OStream,
3. sword::IStream.

```
ostream& operator<<(ostream& lhs, const std::vector<T>& rhs);
OStream& operator<<(OStream& lhs, const std::vector<T>& rhs);
IStream& operator>>(IStream& lhs, std::vector<T>& rhs);
```

It also provides a single operation to get the sum of all elements within a vector, provided that the underlying data type of the vector has a meaningfull '+' operator:

```
T sum(const std::vector<T>& vect)
```

1.8.3 std::map<> tools

```
#include <sword/sword.h>
#include <sword/stltools.Map.h>
```

Using this header, it becomes possible to stream any std::map either as text (on a std::ostream) or in the Sword streams framework (sword::OStream and sword::IStream). The only condition is that the underlying data types (key and value) of the map instance are also streamable on the same stream:

1. std::ostream. The textual format used is '<key1=>value1, key2=>value2, key3=>value3>',
2. sword::OStream,
3. sword::IStream.

```
ostream& operator<<(ostream& lhs, const std::map<IDX,T>& rhs);
OStream& operator<<(OStream& lhs, std::map<K, V>& rhs);
IStream& operator>>(IStream& lhs, std::map<K, V>& rhs);
```

1.8.4 std::in<> utility

Each one of the “stltools.Container.h” also define a specialized implementation for the std::in<> utility template function. The general syntax is:

```
bool std::in<T>(const T& value, const Container<T>& container);
```

Given a value and a container based on the value's type, it returns “true” if the value can be found in the container, or false otherwise. For associative containers, the search happens on keys.

1.9 Configuration tools

A very common and not-so-easy task every large application programmer face is to read and write configuration files. The requirements are usually mostly the same:

1. organize the configuration as a tree of name / value pairs,
2. be able to type the values with all basic data types (string, int, double, bool)
3. read/write a config tree from/to a textual file format which can be easily read and modified by humans

The file format used by the Sword configuration tools is a simple key / value pairs hierarchical layout using blocks. For instance, if we consider the configuration tree:

```
/
+--- NODE1
|    +--- KEY1 => value1
|    +--- KEY2 => value2
|
+--- NODE2
|    +--- KEY3 => value3
|
+--- KEY4 => value4
```

The configuration is made of nodes (which can contain leafs or other “sub” nodes) and leafs (which directly contain values). Another way of looking at this configuration is via a single set of key/value pairs, but with “compound” key names:

```
NODE1/KEY1 = value1;
NODE1/KEY2 = value2;
NODE2/KEY3 = value3;
KEY4 = value4;
```

This tree will be saved in a textual file:

```
NODE1 {
  KEY1 value1;
  KEY2 value2;
}
NODE2 {
  KEY3 value3;
}
KEY4 value4;
```

The global application configuration can also be organised in a set of config files which can be glued together via the INCLUDE directive. Using this feature, the same configuration tree can be layed out in two physical files:

```
// file1.cf
INCLUDE file2.cf
NODE1 {
  KEY1 value1;
  KEY2 value2;
}

// file2.cf
NODE2 {
  KEY3 value3;
}
KEY4 value4;
```

1.9.1 sword::ConfigNode

```
#include <sword/sword.h>
#include <sword/config.ConfigNode.h>
```

The ConfigNode is the code representation of a configuration node or leaf. In order to resolve all “memory ownership” problems, config nodes are usually used via the PConfigNode smart pointer:

```
typedef SmartPtr<ConfigNode> PconfigNode;
```

A node or a leaf can be created via one of the constructors. The name used for the creation must be a simple name (the actual name of the entry in the tree), never a compound name (path to the node):

```
ConfigNode(const std::string& name);
```

```
ConfigNode(const std::string& name, const std::string& value);
ConfigNode(const std::string& name, const char *value);
ConfigNode(const std::string& name, int value);
ConfigNode(const std::string& name, double value);
ConfigNode(const std::string& name, bool value);
```

Basic attributes of a config node can be retrieved via standard get accessors:

```
std::string getName() const;
bool isNode() const;
bool isLeaf() const;
```

Further manipulation on nodes include: does it have children, how many, access a specific child, remove one or all children or iterate over all the children:

```
bool isEmpty() const;
size_t size() const;

PConfigNode getChild(const std::string& name) const;
PConfigNode tryGetChild(const std::string& name) const;

void removeChildren();
void removeChild(const std::string& name);

const_iterator begin() const;
const_iterator end() const;
iterator begin();
iterator end();
```

There is a difference between “getChild” and “tryGetChild” when the searched child does not exist. getChild throws a KeyNotFoundException exception whereas trygetChild simply returns a null smart pointer.

Further manipulation on leafs include 3 read and 2 write accesses for each supported data type (std::string / XXX=String, int / XXX=Integer, double / XXX=Double and bool / XXX=Boolean). All those functions are part of the “::sword::Config” namespace:

```
namespace Config {

  XXX  getXXX(PConfigNode node) const;

  XXX  getXXX(PConfigNode node,
       const std::string& name) const;

  XXX  getXXX(PConfigNode node,
       const std::string& name, const XXX& defaultValue) const;

  void setXXX(PConfigNode node,
       const XXX& value);

  void setXXX(PConfigNode node,
       const std::string& name, const XXX& value);

} // namespace Config
```

1. getXXX(node)
   obtains this leaf's value,
2. setXXX(node, value)
   changes this leaf's value,
3. getXXX(node, name)
   obtains a sub-node value. The name can be a simple or a compound name. The leaf must already exist in the configuration tree or else a KeyNotFoundException is thrown,
4. setXXX(node, name)
   changes a sub-node value. The name can be a simple or a compound name. If the leaf does not exist yet, it is implicitely created.

5. getXXX(node, name, defaultValue)
   obtains a sub-node value. The name can be a simple or a compound name. If the leaf does not exist, the default value is returned.

1.9.2 Config file reading & writing

```
#include <sword/sword.h>
#include <sword/config.ConfigIO.h>
```

Configuration trees can be very simply read from or write to standard streams (files via ifstream/ofstream, strings via istringstream/ostringstream, etc...). Two functions in the sword namespace can be used for this purpose:

```
void configRead (PConfigNode node, std::istream& iss);
void configWrite(PConfigNode node, std::ostream& oss);
```

Additionally, it is possible to provide a ConfigReadHandler instance when reading from a stream. This handler implementation will be called whenever an "INCLUDE" directive is encountered so that your code can switch to the new disk file (for instance). It is also called upon errors in order to build the error string (so that you can append the current file name for instance).
The handler interface you have to implement is:

```
class ConfigReadHandler {
public:
  virtual std::string position(
       word32 lineNumber) = 0;
  virtual void onInclude(
       PConfigNode node,
       const std::string& includeName,
       word32 lineNumber) = 0;
};
```

If you want to provide your own handler, then use the more specific read function:

```
void configRead(PConfigNode node, std::istream& iss,
                ConfigReadHandler& handler);
```

Two standard handlers are provided:
1. *ConfigReadHandler_NOP*: the simplest one, which does nothing special upon "INCLUDE". This handler is used by default in the simplest "configRead" function,
2. *ConfigReadHandler_DiskFile*: a standard implementation for disk-file-based configuration trees. It will follow "INCLUDE" directives on disk. It provides an additional "process" method which will handle all the file opening, reading... automatically.

1.9.3 Example: file read and conf access

Here is a simple example which is capable of reading config files on the disk (possibly following INCLUDE directives) and finally gets a few values.

```
// create an empty root node
sword::PconfigNode root(new sword::ConfigNode("/"));

// read the configuration from the disk
sword::ConfigReadHandler_DiskFile crf("myconfig.cf");
crf.process(root);

// now the configuration is in memory
// obtain a few specific values
int    v1 = getInteger(root, "KEY1/VALUE1", 3);
double d2 = getDouble (root, "KEY1/VALUE2", 3.14159);
```

1.9.4 Example: conf creation and file write

Here is a simple example of code-driven configuration creation and writing into a disk file:

```
// create a few nodes
PConfigNode root(new ConfigNode(""));

PConfigNode node1(new ConfigNode("NODE1"));
PConfigNode node2(new ConfigNode("NODE2"));
PConfigNode node3(new ConfigNode("NODE1_3"));

root->addChild(node1);
  node1->addChild("NODE1_1", "Love Box");
  node1->addChild("NODE1_2", 256);

root->addChild(node2);
  node2->addChild("NODE1_1", "Vertigo");
  node2->addChild("NODE1_2", 128);
  node2->addChild(node3);

root->addChild("OPTION1", "Goodbye Nightclub (Hello Country)");
root->addChild("OPTION2", 512);
root->addChild("OPTION3", 3.14159);
root->addChild("OPTION4", true);

// write all that to a disk file
std::ofstream ofs(“myconfig.cf”);
sword::configWrite(root, ofs);
```

The result is a new file “myconfig.cf” which now contains:

```
NODE1 {
       NODE1_1 "Love Box";
       NODE1_2 256;
}
NODE2 {
       NODE1_1 "Vertigo";
       NODE1_2 128;
       NODE1_3 {
       }
}
OPTION1 "Goodbye Nightclub (Hello Country)";
OPTION2 512;
OPTION3 3.14159;
OPTION4 "TRUE";
```

2 Sword/UI Reference Guide

2.1 Introduction

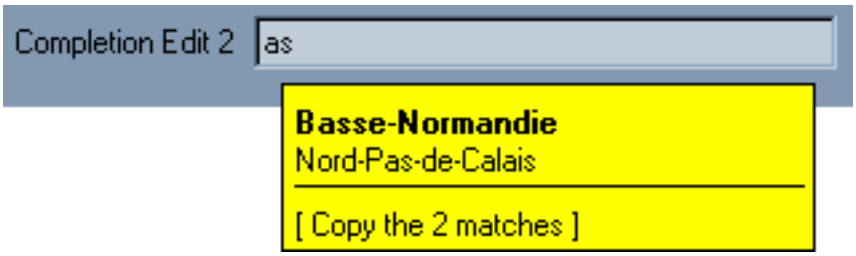
Sword/UI is a set of visual classes (widgets) which complement the QT library. The Sword/UI library sits on top of Sword + QT and can be used in any project based on those two toolkits. The QT library is a portable graphical user interface toolkit made by Trolltech (www.trolltech.com). It is available on Unix/X11, MacOSX and MsWindows with different licensing policies (GPL or Commercial on Unix/X11 and MaxOSC, Commercial only on MsWindows). QT is well known to be the basic build block undermining Linux's KDE (www.kde.org).

2.2 Simple widgets

2.2.1 Completion Editor

```
#include <sword/sword.h>
#include <sword/ui/CompletionEditor.h>
```

The Completion Editor is a specialised QLineEdit. While the user type completion are searched into a CompletionData “data source”. Matches are then displayed in a tooltip window for the user to choose:



2.2.1.1 Implement a CompletionData source

The completion editor looks up possible matches in a “virtual container” which can be any class implementing the CompletionData interface:

```
class CompletionData {
public:
  CompletionData() {}
  virtual ~CompletionData() {}
  //
  virtual void begin() = 0;
  virtual bool hasNext() = 0;
  virtual void next() = 0;
  virtual std::string value() = 0;
};
```

This “virtual data source” is then browsed as a list of string entries in order to look for matches:

```
CompletionData& cd = /*...*/;

cd.begin();
while(cd.hasNext())
{
  std::string value  = cd.value();
  /* ... look for a match between 'value' and the widget entry ... */
  cd.next();
}
```

If your list of possible strings is stored in a std::vector<std::string>, then the implementation for CompletionData is trivial:

```
class StdCompletionData {
public:
  typedef std::vector<std::string> Strings;
  StdCompletionData(const Strings& strings) : strings_(strings) {}
  //
  virtual void begin() { it_ = strings_.begin(); }
  virtual bool hasNext() { return it_ != strings_.end(); }
  virtual void next() { ++it_; }
  virtual std::string value() { return *it_; }

private:
  const Strings& strings_;
  Strings::const_iterator it_;
};
```

2.2.1.2 Use the CompletionEditor

The CompletionEditor extends the standard QLineEdit. The only API difference is the constructor:

```
CompletionEditor(QWidget *parent, const char *name,
                 CompletionData *completionData,
                 bool caseSensitive = true,
                 bool matchPart = true);
```

An implementation of CompletionData must be provided so that the completion editor knows where to find the possible matches. Then two flags control the behaviour of the widget:

1. **caseSensitive**: specify whether the match between the typed value and a possible entry must be case sensitive (upper or lower case matters) or case insensitive (case doesn't matter),

2. **matchPart**: specify whether typed value can match any part of a list entry (if true) or if only the match with the beginning of an entry is allowed (false).

2.2.1.3 User guide

While typing, the possible matches are displayed in a tooltip menu.

Keyboard navigation:
- Choose between several possible choices : **Up** and **Down** arrows
- Accept the current choice : **Enter**
- Hide the choices menu : **Escape**
- Show the menu again : **Ctrl+Space**

Mouse navigation:
- Click on a choice : choose it
- Double click on a choice : accept it
- Click on “Copy the X choices” : the list of possible matches is copied in the clipboard

2.3 Enhanced Table

3 Performances

3.1 Lexical Cast

3.1.1 Formatting

The objective is always to convert a primitive type to a std::string. We will compare the printf implementation, the lexical cast implementation and the C++ stream implementation (std::ostringstream).
We will not consider formatting implementations that simply output in a “char*” because most of the time in C++ we will need to transfer this into a std::string anyway. If even more speed is required and a “char*” result is acceptable, then the Sword formatting primitives available within the lexical cast implementation can be used directly.

The results have been obtained on a Pentium IV, 2,66 Ghz, Windows 2000 with Microsoft Visual C++ 7.1 compiler in release mode. They are expressed in term of “number of conversions per second”. The printf implementation speed (the reference) has been normalized to 100%.

The test program does a high number of conversion and computes the average number of conversions per second. Before each test another completely independent code runs several times in order to clear all processor caches and prediction algorithms.

3.1.1.1 Integer formatting

The default formatting does not specify any formatting constraints:

```
// printf implementation
char buffer[50];
sprintf(buffer, "%d", i);
std::string s = buffer;

// lexical cast implementation
std::string s = lexical_cast<std::string>(i);

// C++ stream implementation
std::ostringstream oss;
oss << i;
std::string s = oss.str();
```

A specific formatting can be required and specified upon each call:

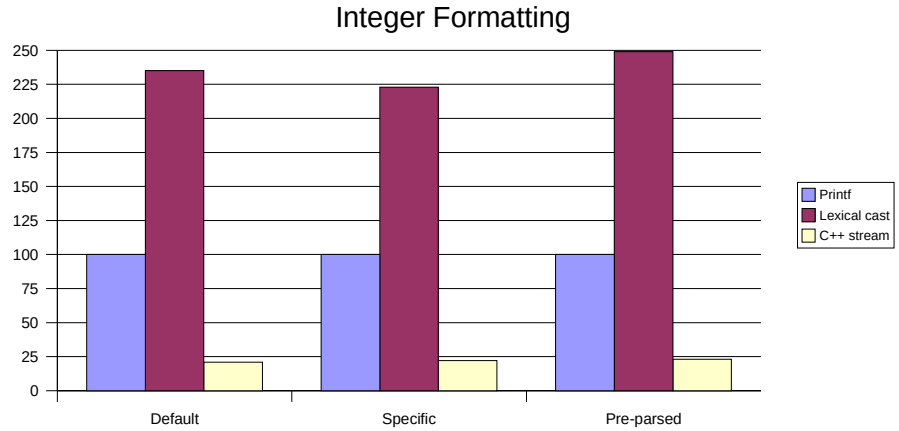
```
// printf implementation
char buffer[50];
sprintf(buffer, "%5.3d", i);
std::string s = buffer;

// lexical cast implementation
std::string s = lexical_cast<std::string>(i, "5.3d");

// C++ stream implementation
std::ostringstream oss;
oss << std::setw(5) << std::setprecision(3) << i;
std::string s = oss.str();
```

Finally, in the lexical cast implementation only, the formatter specification can be factorised among many calls:

```
// lexical cast implementation
LexicalFormatter f("5.2d");
for(int i=0; i<N; ++i)
{
  std::string s = lexical_cast<std::string>(i, f);
  ...
}
```

The performance results are:

	Default	Specific	Pre-parsed
Printf	100	100	100
Lexical cast	235	223	249
C++ stream	21	22	23



On average, Sword lexical cast implementation is 2.3 times faster than printf and still maintains the C++ advantages: type safety and memory management.

4 Coding Standards

4.1 Introduction

This chapter describes the official guidelines for writing/formatting C++ code within the sword project. These guidelines don't bring immediate advantages in code speed/memory usage, neither do they guarantee high quality code. But the benefit of following these guidelines is at least as valuable: we'll get clean, readable code that is easy to maintain and allows for extension. That is vital for a team!

As you may know, there are quite a few C++ coding styles out there. Every good team and programmer follows a coding style. There are not many objective arguments to compare different coding styles, and to decide that one style is better than another, and in any case, any style is better than no style. So the style presented here makes no claims to be better than other styles, it's a style on which Sword developers agree on; it tries to offer a set of guidelines to unify the way code 'looks' across the project, without placing a burden on the programmer.

The basic goal for \sword\ code regarding coding style is that all of the code should use the same style (as opposed to: 'this is my module, I'll just use my own style').

This chapter will try to cover most of the coding style issues, but at the same time it should be reasonably brief, so it can be easy to use, by avoiding to list hundreds of pages with all the cases and details. When in doubt, take a look at the code already written and/or ask for advice from the other team members (and if it's the case, update this doc).

4.2 General guidelines

- The code should be clear and simple to read. Avoid tricks and use special optimizations only when they are *really* needed,
- The code formatting should follow and express the logical structure of the program (e.g. group similar code together),
- Public interfaces (everything that is exposed by a class, global functions, constants) should be clearly documented (and should not require reading through the implementation to understand basic functionality),
- Try to avoid abusing C++ features to create an 'extended language' – e.g. try to avoid macros (use a better alternative when possible: inline functions, C++ constants, etc.),
- Avoid creating an unjustified number of synonymous names for the same entity,
- Coding style should be consistent - all the code should look the same (no matter who wrote it),
- Try to fit the code in 80 columns (break extra long lines),
- Others should enjoy reading your code as much as you enjoy writing it.

4.3 Identifiers naming

- Names should be suggestive (a23, bjXY, xwv) and concise (theMaximumValueForTheInterval),
- When an identifier is composed of more words, each intermediary word will be capitalized (lastEvent, ComboBox, windowList_, setFixedSize) and not using underline, or without any form of separation (last_event, setfixedsize).

4.3.1 Variables names (global, member, auto, ...)

- Variables identifiers should start with a lowercase letter (totalTime, i, cmdOk, TotalTime, CmdOK),

- All data members will end with '_' (value_, parent_, range_). Note that public data members should be avoided as much as possible and thus the '_' is also the sign of a private or protected item,
- Global data, such as singletons, should be prefixed with 'The' (TheLogMediaManager, ConfigurationSingleton).

4.3.2 Types (class, enum, typedef)

- Class names begin with uppercase letter with no prefix (Object, Time, OStorage, CObject, TTime),
- Template arguments are spelled with all letters uppercase (template<class TYPE> class MyTemplateClass).

4.3.3 Functions (global, methods, ...)

- Function (global or members) names begin with lowercase letter and no prefix (read(), isRunning(), registerMedia(), Read(), bIsRunning(), m_initInstance()),
- Methods pairs that set/get the values of an 'attribute' should be named as follows:
    - Attribute: 'TextColor',
    - Type: 'Color'
    - Set method: 'void textColor(const Color &color)',
    - Get method: 'Color textColor() const'.

4.3.4 Constants

- Preprocessor constants names should be spelled all uppercase (HAVE_UNISTD). Please note however that preprocessor constants should be avoided and C++ 'static const' or 'enum' should be used instead whenever possible,
- Typed constants (or enum members) should be spelled like normal variables.

4.3.5 Namespaces

Namespace names follow the same naming conventions as identifier names (starting with lowercase and each intermediate word is capitalized). (namespace sword).

4.4 Source code organization

4.4.1 Source file names

- Source file names follow the same naming conventions as class names (starting with uppercase and each intermediate word is capitalized). (Time.cpp, Log.cpp, time.cpp, log_media.h),
- Source files are named after the class they declare or implement,
- Source files follow an hierarchical naming style where the first part is a module name (and sub modules if necessary) followed by the class name. Name parts are separated by dots and module names are all lower case. (streams.File.h, File.h, file.cpp).
- Source files use the following suffixes (extensions):
    - C++ source file : '.cpp',
    - C++ header : '.h',
    - C++ source intended to be included/inlined : '.inl'.

4.4.2 Location (hierarchical structure)

Headers are in '/sources/include/sword', inline files are in '/sources/include/sword.inline', source files are in '/sources/src'. Additionally, samples files are in a hierarchy below '/samples' and the regression tests are in '/testsuite'.

4.4.3 Source file structure

- Lines should not be longer than 80 columns
- Header files should use the a symbol definition to avoid multiple inclusions of the same header. The symbol used must be modeled after the module and the name of the class declared in the file (and thus after the name of the file itself.
- All source files should begin with '#include "sword/sword.h"'. This is to ensure that the same data types and system include files are used everywhere, as well as to speed up compilation with compilers that can handle pre-compiled headers.

Example:

```
#ifndef _SWORD_MODULE_CLASS_
#define _SWORD_MODULE_CLASS_

... the rest of the header ...

#endif  // _SWORD_MODULE_CLASS_
```

4.5 Comments

- Comments should add extra information to the code (for a human reader) and not duplicate what the code does (and should be brief, easy to read),
- Comments should describe the logic of classes, methods, or blocks of code with the same purpose, and not each individual line,
- Comments will be indented with the code, and the comments should be on separate lines preceding the code (and not on the same line with the code),
- Each function (global or member) should be documented in a function comment block before the function using the doxygen metainfo syntax:
    - Brief description of the function,
    - Description of each parameter,
    - Description of return value,
    - List potential exceptions that the function might throw,

The template for doxygen metainfo method commend is as follow:

```
//! Brief description
/*! And this complements the brief description with longer and
 *  more detailed specs
 *  \param name1 documentation
 *  \param name2 documentation
 *  \throws MyException
 *  \see A::f, B::g
 */
void f(int name1, int name2);
```

Additional information for a function could be:

- Pre-conditions, post-conditions and invariants,
- List of related entities (functions, classes, ...)
- Description of the algorithm used (if non trivial) - including time and space complexity classes,
- Side effects (should be avoided!),
- Thread safe or not.

4.6 Formatting

4.6.1 Indentation guideline

- Indentation should use 2-spaces spacing.
- A C++ block is indented as follows:

- Enclosing pair of brackets should be on the same column and should be the only thing in that line (with the exception of do..while and switch statements, see below)
- The block contents should be indented as follows:

```
if (condition)
{
  ...
}
else if (condition2)
{
  ...
}
```

- Switch statements should be indented as follows:

```
switch(value) {
case value1:
  ...
  break;

case value2:
  ...
  break;

default:
  ...
  break;
}
```

- A C++ class is indented as follows:
  - The block bracket should be on the same line as the class keyword
  - The template specification, if any, should be on the previous line,
  - Protection keywords must be aligned with the class keyword,
  - All other content should be indented:

```
template<class T>
class MyClass {
public:
    MyClass();
    ~MyClass();
private:
    void f_();
};
```

4.6.2 Functions (methods)

- Functions and methods will be formatted as follows:
  Type foo(T1 arg1, T2 arg2, ...);
  and
  Type Class::foo (T1 arg1, T2 arg2, ...);
- Methods must never be implemented directly within the class declaration (even if they ought to be inline). The body will be implemented either in the inline file ('.inl') or in the source file ('.cpp') associated with the header,
- When declaring the prototype of a method use the names of the paramenters (although C++ allows to skip them). Also try to use the same names in the method implementation,

4.6.3 Declarations

- Always put the pointer '*' or reference '&' symbol with the variable, not with the type (const std::string &name} but MyClass& myInstance),
- Always create a type for a container instance using a 'typedef'. Use this new type instead of the container in order to access iterators, etc...

```
typedef map<string, MyClass> Classes;
```

```
Classes classes;

for(Classes::iterator i=classes.begin();
...
```

4.6.4 Expressions

- In expressions, binary operators will be separated from their operands by a space (with the exception of operator ',' which will have a space only after it (i = 10, j = 0;)),
- (), [] and unary operators will not be separated with space from their operands (t[a + b] = foo (*x), t [a+b]=foo (* x)),
- Don't use too many (). Use them only to force a different evaluation order or to enhance expression clarity,
- When declaring data members or variables, each variable will be on a separate line.