KeyValue

User's Manual - version: 0.1

Cassio Neri

Table of Contents

1. Introduction

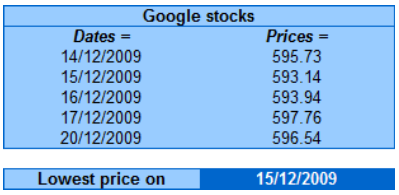
KeyValue is a cross-platform library for making C++ objects accessible through OpenOffice Calc, Excel and other front-ends. Experience of spreadsheet users is enhanced by an object model and a handy key-value based interface.

Actually, KeyValue does more than just help creating spreadsheet functions. The object model allows end-users to build C++ objects through the front-ends. These objects are stored in a repository for latter use at user's request. Additionally, the KeyValue provides a set of services to an effective use of these objects.

The library is named ater one of its main features: The key-value based interface. Parameters are passed to functions through key-value pairs in contrast to the standard positional interfaces of OpenOffice Calc, Excel, C/C++, etc.

For instance, consider a function which requires stock prices at different dates. Two vectors have to be passed: A vector of dates and a vector of prices. In a positional interface these two vectors would be provided in a specific order, say, first the vector of dates followed by the vector of prices. In contrast, KeyValue allows a label (or **key**) to be attached to each vector (the **value** associated to the key) in order to distinguish their meanings. In the example, the keys could be *Dates* and *Prices* while the values would be the vectors of dates and prices themselves.

To give a taste of KeyValue, let us develop this example a bit further. Suppose we want to write a C++ function that, given a set of dates and corresponding stock prices, returns to the spreadsheet the earliest date where the stock has reached its lowest level. In the termsheet we would see something like in Figure 1, "Data organized in the spreadsheet.".



Google stocks	
Dates =	*Prices =*
14/12/2009	595.73
15/12/2009	593.14
16/12/2009	593.94
17/12/2009	597.76
20/12/2009	596.54

Lowest price on	15/12/2009

Figure 1. Data organized in the spreadsheet.

The C++ code (see `keyvalue/bridge-example/bridge-example/processor/LowTime.cpp`) could be:

```
template <>
value::Value
Calculator<LowTime>::getValue(const DataSet& data) const {             // A

  const key::MonotoneBoundedVector<ptime, key::StrictlyIncreasing>
    keyDates("Dates");                                                 // B

  const std::vector<ptime>& dates(*data.getValue(keyDates));           // C
  const key::MonotoneBoundedVector<double, key::NonMonotone, key::Geq>
    keyPrices("Prices", 0.0, dates.size());                          // D

  std::vector<double>& prices(*data.getValue(keyPrices));            // E

  double lowPrice = prices[0];                                       // F
  ptime lowDate = dates[0];

  for (size_t i=1; i<prices.size(); ++i)
    if (prices[i] < lowPrice) {
            lowPrice = prices[i];
            lowDate  = dates[i];
    }                                                                // G

  return lowDate;                                                    // H
}
```

Without getting too deep in the details, we shall comment some important points of this example:

A:

Functions returning values to the spreadsheet are specializations of template class `Calculator` of which `getValue()` is the main method. The template type parameter `LowTime` is just a tag identifier to distinguish between different functions.

B:

Variable `keyDates` holds information about key *Dates* including the label "`Dates`" seen on the spreadsheet. Being an instantiation of `key::MonotoneBoundedVector`, it also knows that the expected type of value is a `std::vector<ptime>`[1]whose elements are in increasing order.

Many other generic keys like `key::MonotoneBoundedVector` are implemented. We can implement application specific keys when no generic key fits our needs or if this proves to be convenient. For instance, implementing a class named `Dates` can be convenient if key *Dates* is to be used very often. This class would hold all the information cited above. Then, line *B* could be replaced by

```
const Dates keyDates;
```

C:

The method `DataSet::getValue()` retrieves the `std::vector<ptime>` containing the dates. At this time, all the information contained in `keyDates` is used. In particular, the constraints on the input are verified and an exception is thrown if the check fails. Therefore, if execution gets to next line, we can safely assume that dates are in increasing order.

D:

Variable `keyPrices` holds information about key *Prices*: the label "`Prices`" and the expected type of value, that is, a `std::vector<double>` of size `dates.size()` and positive elements.

E:

Retrieves the `std::vector<double>` and, if execution gets to next line, we can be sure that `prices` and `dates` have the same size and all `price` elements are positive. Otherwise an exception will be thrown.

F - G:

This bit of code could be part of the library which KeyValue helps to make accessible through OpenOffice Calc or Excel instead of being here.

H:

While the type returned by `Calculator<LowTime>::getValue()` is `value::Value` the code above returns a `ptime`. For convenience, KeyValue implements a collection of implicit conversions to

[1]KeyValue uses time and date class `ptime` from Boost's Date_Time library.

`value::Value` from several types including bool, double, `string`, `ptime`, `std::vector<double>`, etc.

More than just a nice interface, KeyValue provides memory management, dependency control, exception handling, caching (memoization) and other services.

The two main examples of front-ends (both provided with KeyValue) are OpenOffice Calc and Excel. A third example is a XML parser which can be useful, for instance, for regression tests. Other front-ends may be easily implemented thanks to KeyValue's modular design represented in Figure 2, “KeyValue's design.”.

There are four layers. The main layer is occupied by KeyValue alone and is independent: It does not `#include` any header file from other layers.

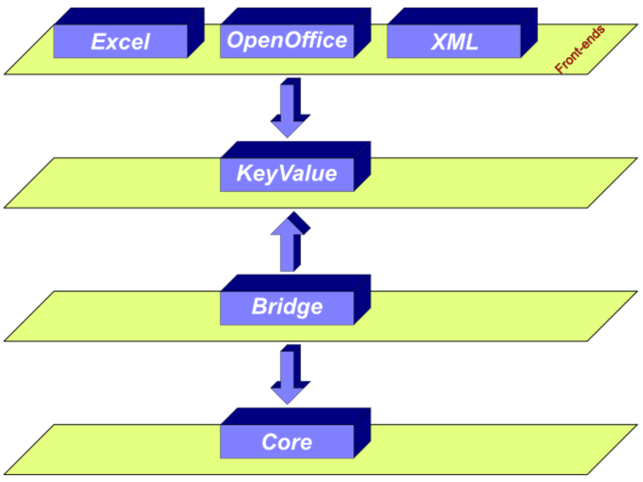


Figure 2. KeyValue's design.

The top layer is populated by front-ends. Components of this layer only `#include` header files from KeyValue. (Fact indicated by the down arrow.)

The bottom layer hosts the **core library**, that is, the C++ library which we want to use through front-ends with KeyValue's help. This layer is also independent.

The **bridge** layer connects KeyValue and core library. Bridge `#includes` files from both layers it is connected to.

Additionally to KeyValue layer, the library provides the front-end (excluding the XLM parser which will be available in a future release). KeyValue users, have to implement the bridge and core library. If they wish, they can also easily implement other front-ends.

2. Download and install

KeyValue is available in standard formats in SourceForge.

http://sourceforge.net/projects/keyvalue/files

Just download and unpack it in your hard disk.

Windows Vista users must perform an extra step. As we shall see below, KeyValue build system relies on Cygwin. For some reason, Cygwin fails to copy some files. To prevent this from happening, turn KeyValue's home directory and all its descendants into shared folders. Right click on KeyValue's home directory and select Properties. Then, click on Sharing / Share ... / Share / Done / Close. Then the directory gets a new icon with a two-people picture.

KeyValue depends on a few libraries and tools. Some of them are compulsory while others depends on the user's purpose. The following sections list those tools and libraries.

2.1. Compiler

Two C++ compilers are supported: Microsoft Visual C++ 2008 (for Windows) and GCC (for Linux).

Most of Linux distributions come with GCC already installed. KeyValue has been tested with version 4.x.x but other versions should work as well.

Microsoft provides different editions of Visual Studio C++ 2008. The Express Edition is available, free of charge, at

http://www.microsoft.com/express/download

Editions differ mainly in their IDEs. However, there are a few differences on compilers as well. During KeyValue's development we came across lines of code that the Professional Edition could compile while Express Edition failed. Some effort has been made to maintain compatibility with both editions.

2.2. Build tools

We need additional build tools, notably, GNU make and the bash shell.

Linux users do not have to worry about most of these tools since they are probably installed by default. However, a less popular tool called **makedepend** is needed as well. Normally, it is part of the x11 or xorg packages. To check whether we have it or not, on a console window type:

```
$ makedepend
```

If not found, use your distribution's package system to install it or, alternatively, download and install from source code:

http://xorg.freedesktop.org/releases/individual/util

Windows users will also need those tools but, unfortunately, they are not directly available. Therefore, Cygwin (see Section 2.4, “Cygwin”) will be needed to have a Linux-compatibility layer.

2.3. Boost

Boost is a high quality set of C++ libraries for general purposes.

KeyValue depends on a few of Boost libraries notably Smart Ptr (for shared pointers) and Date Time (for date and time classes). All Boost libraries that KeyValue depends on are header-only. Therefore, all we need is to download and unpack Boost in the hard disk.

As of this writing, the latest Boost release is 1.42.0. KeyValue have been tested with version 1.38.0. Any newer version should work as well.

Boost is available for download at its SourceForge page:

http://www.boost.org/users/download/

2.4. Cygwin

KeyValue is a cross platform library for Linux and Windows systems. Its build system relies on tools that are very popular on Linux systems but not on Windows. For that reason, Windows users must install Cygwin to have a Linux-like compatibility layer. Cygwin is available at

http://www.cygwin.com

During installation we have to make a few choices. Normally, default answers are fine. However, when selecting the packages to install, make sure the following items are selected:

- Archive/zip (needed to build the OpenOffice Calc add-in);
- Devel/make; and
- Devel/makedepend.

Although installation procedures for KeyValue developers is not in the scope of this document, we anticipate here the list of extra Cygwin packages that developers must install:

- Archive/zip; and
- Doc/libxslt.

Cygwin comes with a small tool called **link** to create file links (shortcuts). This tool is, probably, useless since there is a Windows native alternative and Cygwin also provides **ln** for the same purpose. Unfortunately, we must bother with **link** because it has the same name as the Microsoft linker, which raises a conflict. A workaround is renaming **link** to, say, **link-original**. Open a **Bash Shell** by clicking on Start / Programs / Cygwin Gygwin Bash Shell and type the following command followed by **Enter**.

```
$ mv /usr/bin/link.exe /usr/bin/link-original.exe
```

In many occasions we need Bash Shell commands. Therefore, remember how to get a Bash shell console window and consider keeping it constantly open while working with KeyValue.

2.5. OpenOffice SDK

KeyValue comes with an OpenOffice Calc add-in for Linux and Windows systems. To build this add-in, one must install the OpenOffice SDK.

The OpenOffice Calc add-in has been tested for versions 3.1.0 of OpenOffice and OpenOffice SDK. However, it probably works for all 3.x.x versions. Users of versions 2.4.x are advised to upgrade their systems.

Download and install a SDK version compatible with your installed OpenOffice version:

http://download.openoffice.org/sdk/index.html

2.6. Excel SDK

KeyValue comes with an Excel add-in. To build this add-in, one must install the Excel SDK.

Only the Excel 2007 API is supported. If compatibility with this API is kept on new Excel releases, then the add-in should work for them as well. However, it does not work for Excel 2003.

Download Excel 2007 SDK from its website

http://www.microsoft.com/downloads/details.aspx?FamilyId=5272E1D1-93AB-4BD4-AF18-CB6BB487E1C4&displaylang=en

3. Configure and build

Locate the file `config/config.mak-example` in KeyValue's home directory. Make a copy named `config.mak` and edit it with a text editor. The file contains detailed explanations.

We emphasize one particular instruction presented in the file. If you are not yet familiar with KeyValue, then leave the variables `FELIBS_debug` and `FELIBS_release` as they are. This allows for the building of the add-in used in Section 4, “Getting started with KeyValue”.

On Bash Shell console, go to KeyValue's home directory. For instance, assuming KeyValue was unpacked in `/home/cassio/keyvalue`, type

```
$ cd /home/cassio/keyvalue
```

Under Cygwin one has to prefix the directory name by the drive letter. Supposing that KeyValue was unpacked in `C:\Users\cassio\Documents\keyvalue`, type

```
$ cd C:/Users/cassio/Documents/keyvalue
```

To build the debug version (the default):

```
$ make
```

To build the release version:

```
$ make release
```

Additionally, **make** accepts other targets. To get a list of those targets:

```
$ make help
```

It also shows the list of projects to be compiled as chosen in `config.mak`.

For users, the most important targets are `clean`, `debug` and `release`. Other targets are more relevant for KeyValue developers and may not work for regular users. They are covered in KeyValue Developer's Manual.

3.1. Using Microsoft Visual Studio 2008 IDE

Microsoft Visual Studio 2008 users will find target `sln` very helpful. The command

```
$ make sln
```

creates a Microsoft Visual Studio 2008 solution (`keyvalue.sln`) and project files which allows for using Microsoft Visual Studio IDE, liberating users from direct calling **make** on Cygwin. Two configurations, debug and release, are set in `keyvalue.sln`.

Open `keyvalue.sln`. On the solution explorer (**Ctrl+Alt+L**), we see the projects. Initially, the start up project will be the first on alphabetic order. Change it to either *excel-addin* or *openoffice-addin*: Right click on the project name and then on Set as StartUp Project.

To configure *excel-addin* and *openoffice-addin* projects to call the appropriate applications under the debugger:

- Right click on *excel-addin* project, select Properties and then Debugging. Edit the fields following the example shown in Table 1, “Configuring MSVC debugger for *excel-addin.*”.

Table 1. Configuring MSVC debugger for *excel-addin.*

Field	Content
Command	Full path of Excel executable (*e.g.* `C:\Program Files\Microsoft Office\Office12\EXCEL.EXE`)
Command Arguments	`out\windows-msvc-debug\keyvalue.xll`

- Right click on *openoffice-addin* project, select Properties and then Debugging. Edit the fields following the example shown in Table 2, “Configuring MSVC debugger for *openoffice-addin.*”.

Table 2. Configuring MSVC debugger for *openoffice-addin.*

Field	Content
Command	Full path of soffice executable (*e.g.* `C:\Program Files\OpenOffice.org 3\program\soffice.bin`)

Field	Content
Command Arguments	`-nologo -calc`
Environment	`PATH=C:\Program Files\OpenOffice.org 3\program;C:\Program Files\OpenOffice.org 3\URE\bin;C:\Program Files\OpenOffice.org 3\Basis\program`

4. Getting started with KeyValue

The easiest way to get familiar with some of KeyValue's features is by using any OpenOffice or Excel add-in based on it. KeyValue comes with examples of core and bridge libraries allowing for the build of an OpenOffice and an Excel add-in. This section introduces some of these features using these add-ins.

We assume you are familiar with the basics of OpenOffice Calc or Excel. These two applications have very similar user interfaces. For this reason, we address instructions to OpenOffice Calc users only. Excel users should not have trouble in adapting them. Moreover, remember that OpenOffice is free software available at

http://www.openoffice.org

It is worth mentioning one interface difference between OpenOffice Calc and Excel. In both, either double-clicking or pressing **F2** on a cell start its editing. Pressing **Enter** finishes the edition. If the new content is a formula, while Excel immediately calculates the result, OpenOffice Calc does it only if the cell's content has changed. In particular, **F2** followed by **Enter** recalculates a cell formula in Excel but not in OpenOffice Calc. To force OpenOffice Calc to recalculate the cell, we have to fake a change. Therefore, keep in mind the following:

To recalculate a cell formula double click on the cell (or press **F2** if the cell is the current one), then press **Left Arrow** followed by **Enter**. To recalculate a formula range, in OpenOffice one must select the whole range (select any cell in the range and then press **Ctrl+/**) before pressing **F2**.

From spreadsheet applications, KeyValue derives some terminology regarding data containers:

Single
: Is a piece of data that, in a spreadsheet application, would fit in a single cell. For instance, the number 1.0 or the text "`Foo`".

Vector
: Is a collection of data that, in a spreadsheet, would fit in a one-dimensional range of cells like *A1:J1* or *A1:A10*. More precisely, when those cells are one beside the other in a row we call it a **row vector** (*e.g. A1:J1*). When the cells are one above the other in a column (*e.g. A1:A10*) we call it a **column vector**. In particular, a single is both a row and a column vector.

Matrix
: Is a collection of data that, in a spreadsheet, would fit in a two dimensional range of cells like *A1:B2*. In particular, single and vector are matrices.

Under KeyValue's home directory, we should have a ready-to-use OpenOffice add-in named `keyvalue.oxt` (or an Excel add-in named `keyvalue.xll`). The exact location is shown in Table 3, “Location of Excel and OpenOffice add-ins.”.

Table 3. Location of Excel and OpenOffice add-ins.

Build	OpenOffice (Linux)	OpenOffice (Windows)	Excel
Debug	`openoffice-addin/out/linux-gcc-debug`	`openoffice-addin\out\windows-msvc-debug`	excel-addin\out\windows-msvc-debug

Build	OpenOffice (Linux)	OpenOffice (Windows)	Excel
Release	`openoffice-addin/out/linux-gcc-release`	`openoffice-addin\out\windows-msvc-release`	excel-addin\out\windows-msvc-release

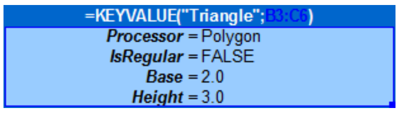
Launch OpenOffice Calc, open the debug add-in and the example workbook `keyvalue.ods` (or `keyvalue.xlsx` in Excel) located in `doc/workbooks`.

Notice that a console window pops up. KeyValue uses it to give some output including error messages.

5. The KEYVALUE function

Cell *B2* on *The KEYVALUE function* sheet of the example workbook contains a formula calling the function KEYVALUE:

```
=KEYVALUE("Triangle";B3:C6)
```



=KEYVALUE("Triangle";B3:C6)

Processor =	Polygon
IsRegular =	FALSE
Base =	2.0
Height =	3.0

Figure 3. Data set *Triangle.*

This function call is meant to build a triangle.

We can see that cells with dark blue background in the sheet contain formulas calling KEYVALUE to build polygons and to calculate their areas.

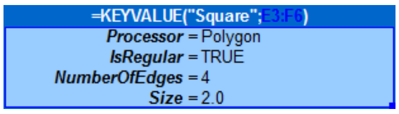
There are no functions such as `BuildPolygon`, `CalculateArea` or anything similar. Indeed, independently on the core-library, KEYVALUE[2] is the only function exported to OpenOffice Calc.

This is not as odd as it might seem (one could expect to call different functions for different tasks). Even when calling a specific function for a precise task, the function might change its behaviour depending on the data it receives. For instance, a function `CreatePolygon` would create a triangle or a square (or whatever) depending on the number of sides given. KeyValue goes one step further and considers the choice of the task as part of the input data as well.

Alternatively, we can think that KEYVALUE does have one single task: It creates **data sets** . A data set is a collection of data organized in key-value pairs (recall the stock prices example given in Section 1, “Introduction”). The example above creates a data set called *Triangle* containing key-value pairs defined by the array *B3:C6* (more details to follow). Analogously, the formula in cell *E2*

```
=KEYVALUE("Square";E3:F6)
```

creates a data set called *Square* containing key-value pairs defined by array *E3:F6*.



=KEYVALUE("Square";E3:F6)

Processor =	Polygon
IsRegular =	TRUE
NumberOfEdges =	4
Size =	2.0

Figure 4. Data set *Square*

More generally, KEYVALUE's first parameter is the name of the data set to be created. This is a compulsory parameter of text type (which might be left empty `""` for **anonymous data sets**). Moreover, as for those examples, often the data set name is the result returned from KEYVALUE to OpenOffice Calc.

[2]The function name might change depending on the bridge.

Once created, a named data set is stored in a **repository** and might be retrieved latter through its name.

Other KEYVALUE's parameters are optional and define key-value pairs following patterns discussed in next section.

6. Key-value patterns

KEYVALUE's parameters, from second onwards, define the data set. Although there is some flexibility on how they are organized, they must follow certain **patterns**. It allows the library to break down the parameters in key-value pairs. Recalculate [8] cell *B2* of sheet *The KEYVALUE function* and take a look at the console logger to see the key-value pairs of data set *Triangle*.

```
[Debug  ] DataSet: Triangle
Size    : 4
Key   #1: Base
Value #1: [Single] 2
Key   #2: Height
Value #2: [Single] 3
Key   #3: IsRegular
Value #3: [Single] 0
Key   #4: Processor
Value #4: [Single] Polygon
```

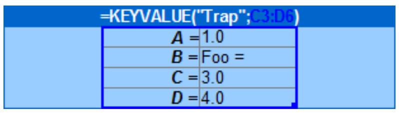
Figure 5. Console logger shows key-value pairs in data set *Triangle*.

For a text to define a key, it is *necessary but not sufficient* that:

- excluding trailing spaces it ends with " =" (space + equal sign); and
- excluding the ending "=", it contains a non space character.

KeyValue replaces the last "=" (equal sign) by " " (space) and, from the result, removes leading and trailing spaces. What remains is the key. For instance, all data sets in sheet *The KEYVALUE function* contain a key called *Processor* defined by the text "` Processor = `".

The conditions above are not sufficient to define a key since the patterns mentioned earlier also play a role in this matter. For instance, in data set *Trap* of sheet *Key-value patterns* , "`Foo  =`" does not define a key *Foo*.


=KEYVALUE("Trap";C3:D6)

A =	1.0
B =	Foo =
C =	3.0
D =	4.0

Figure 6. "Foo =" seems to define a key but it does not.

Actually, it assigns the value "`Foo  =`" to key *B* as you can verify in the console after recalculating [8] *B2*.

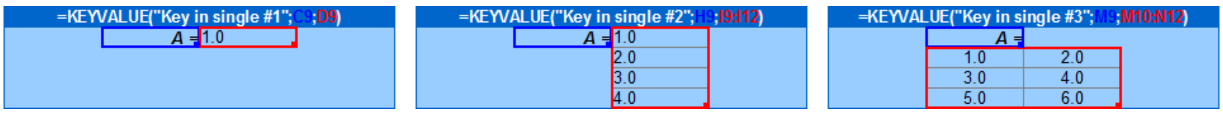
```
[Debug  ] DataSet: Trap
Size    : 4
Key   #1: A
Value #1: [Single] 1
Key   #2: B
Value #2: [Single] Foo =
Key   #3: C
Value #3: [Single] 3
Key   #4: D
Value #4: [Single] 4
```

Figure 7. Console shows that "Foo =" is the value assigned to key *B*.

The following sections explain the patterns and clarify this point.

6.1. Key in single

This pattern is composed by two parts: A single containing a text defining a key (*i.e.* verifying the necessary conditions [10]) followed by a single or a vector or a matrix, which will be the associated value. Those three possibilities are shown in the sheet *Key-value patterns* .



=KEYVALUE("Key in single #1";C9;D9)

A =	1.0

=KEYVALUE("Key in single #2";H9;I9:I12)

A =	1.0
	2.0
	3.0
	4.0

=KEYVALUE("Key in single #3";M9;M10:N12)

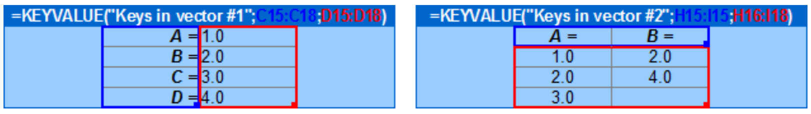
A =	
1.0	2.0
3.0	4.0
5.0	6.0

Figure 8. Key in single pattern.

6.2. Keys in vector

There are two cases of this pattern. The first (the transpose of the second) is composed by a column vector and a matrix such that

- they have the same number of rows; and
- the vector contains only keys (*i.e.* all cells contain text verifying the necessary conditions [10]).



=KEYVALUE("Keys in vector #1";C15:C18;D15:D18)

A =	1.0
B =	2.0
C =	3.0
D =	4.0

=KEYVALUE("Keys in vector #2";H15:I15;H16:I18)

A =	B =
1.0	2.0
2.0	4.0
3.0	

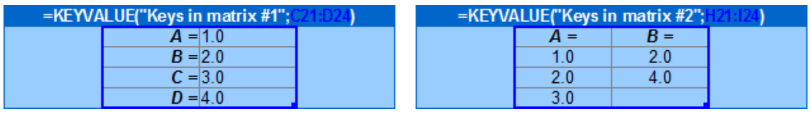
Figure 9. Keys in vector pattern.

Furthermore, for each key in the vector, the corresponding row in the matrix defines a vector which is the value associated to the key.

6.3. Keys in matrix

There are two cases of this pattern. The first (the transpose of the second) is composed by a matrix such that

- it has at least two columns;
- the first column contains only keys (*i.e.* all cells contain text verifying the necessary conditions [10]); and
- the second cell of first row is not a key (*i.e.* it does not contain text verifying the necessary conditions [10]).



=KEYVALUE("Keys in matrix #1";C21:D24)

A =	1.0
B =	2.0
C =	3.0
D =	4.0

=KEYVALUE("Keys in matrix #2";H21:I24)

A =	B =
1.0	2.0
2.0	4.0
3.0	

Figure 10. Keys in matrix pattern.

Furthermore, for each key in the first column, the remaining cells on the same row define a vector which is the value associated to the key.

6.4. Table

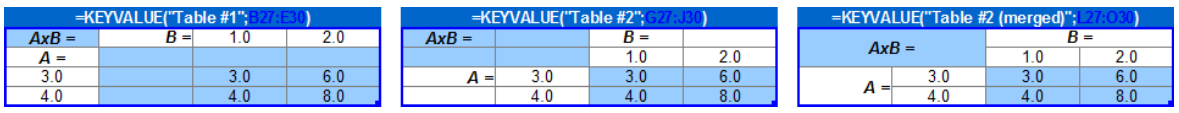
Useful for tables, this pattern is made by one matrix $M = M(i, j)$, for $i = 0, ..., m-1$ and $j = 0, ..., n-1$ (with $m>2$ and $n>2$). In M we find three key-value pairs: row, column and table. There are two variants of this pattern:

Format 1:

The row key is in *M(1, 0)* and its value is the column vector *M(i, 0)* for *i = 2, ..., m-1*. The column key is in *M(0, 1)* and its value is the row vector *M(0, j)* for *j = 2, ..., n* -1. Finally, the table key is in *M(0,0)* and its value is the sub-matrix *M(i, j)* for *i = 2, ..., m-1* and *j = 2, ..., n-1*.

Format 2:

The row key is in *M(2, 0)* and its value is the column vector *M(i, 1)* for *i = 2, ..., m-1*. The column key is in *M(0, 2)* and its value is the row vector *M(1, j)* for *j = 2, ..., n* -1. Table key and value are as in **Format 1**. This variant is aesthetic pleasant when some cells are merged together as show in data set *Table #2 (merged)* in Figure 11, “Table pattern. *A* is the row key, *B* is the column key and *AxB* is the table key.”.



=KEYVALUE("Table #1";B27:E30)

AxB =	B =	1.0	2.0
A =			
3.0		3.0	6.0
4.0		4.0	8.0

=KEYVALUE("Table #2";G27:J30)

AxB =		B =	
		1.0	2.0
A =	3.0	3.0	6.0
	4.0	4.0	8.0

=KEYVALUE("Table #2 (merged)";L27:O30)

AxB =		B =	
		1.0	2.0
A =	3.0	3.0	6.0
	4.0	4.0	8.0

Figure 11. Table pattern. *A* is the row key, *B* is the column key and *AxB* is the table key.

7. Reserved keys

Some special keys are reserved for KeyValue use. They are explained in the following sections.

7.1. *Processor*

The task performed on a data set is defined exclusively by its content. Indeed, excluding the *Default* data set (see Section 9, “Key resolution and the *Default* data set”), the value assigned to key *Processor* informs the action to be performed. More precisely, the bridge library implements a number of **processors** which perform different tasks on data sets. The value assigned to key *Processor* gives the name of the processor for a data set.

For instance, in sheet *Reserved keys*, the formula in *B2* selects processor *Polygon* while the one in *E2* selects processor *Area*. Recalculate [8] *B2* to verify in the logger the called processors:

```
[Debug  ] DataSet: A
Size     : 4
Key   #1: IsRegular
Value #1: [Single] 1
Key   #2: NumberOfEdges
Value #2: [Single] 4
Key   #3: Size
Value #3: [Single] 1
Key   #4: Processor
Value #4: [Single] Polygon
[Debug  ] DataSet:
Size     : 2
Key   #1: Polygon
Value #1: [Single] A
Key   #2: Processor
Value #2: [Single] Area
[Info   ] Processor 'Area' called on anonymous data set.
[Info   ] Processor 'Polygon' called on data set 'A'.
```

Figure 12. In each data set, its key *Processor* selects the processor for this data set.

Processors that create objects are called **builders** (*e.g. Polygon*). Those that compute results are called **calculators** (*e.g. Area*).

Key *Processor* is optional. A data set which does not have such key is called **data-only**.

7.2. *ProcessNow*

In sheet *Reserved keys*, the formula in *B2* actually does not build any polygon. Indeed, for named data sets, by default KeyValue implements a lazy initialization strategy: It avoids to call processors until this is really necessary. In this case, all `KEYVALUE` does is creating the data set *A* which latterly *might* be used to build a polygon. In this example it will happen when we request its area in *E2*.

For named data sets, key *ProcessNow* is used to change this behaviour. If *ProcessNow* is `TRUE`, then the data set is immediately processed and the result is returned to OpenOffice Calc. Otherwise, KeyValue just creates and stores the data set for latter use and the result returned to OpenOffice Calc is the data set name. Change cell *C10* to `TRUE` and `FALSE` and check in the logger when the processor is called.

Anonymous data sets are always processed and *ProcessNow* has no effect. Change *F10* and check the logger.

This key is optional and when it cannot be resolved (see Section 9, “Key resolution and the *Default* data set”) assumes the value `FALSE`.

7.3. *VectorOutput*

When the result of `KEYVALUE` is a vector the user may choose how this vector should be presented on the spreadsheet: As a column vector, as a row vector or as it is returned by the processor. For this purpose, the key *VectorOutput* might be assigned to "`Row`", "`Column`" or "`AsIs`".

This key is optional and when it cannot be resolved (see Section 9, “Key resolution and the *Default* data set”) assumes the value "`AsIs`".

7.4. *Imports*

Key *Imports* is optional. Its value is a vector of data set names whose keys and values are imported to the current data set. For more details see Section 9.2, “Importing all key-value pairs from other data sets”.

7.5. *Export*

Key *Export* is reserved only in *Default* data set where it defines whether key-value pairs in *Default* participate in key resolution or not. (See Section 9, “Key resolution and the *Default* data set”.)

8. Reserved processors

The processors *Polygon* and *Area* were implemented by the *bridge-example* which comes with KeyValue. This bridge is intent to be used only as an example, and will not be linked with more serious applications (yours). Hence these processors will not be present. However, a few processors are implemented by KeyValue (not by the bridge). See the *Reserved Processors* sheet of the example workbook for examples of reserved processors.

8.1. *Logger*

This processor builds a logger where KeyValue sends messages to. The input data set should contain the following keys:

Device

Compulsory key that defines the type of logger. Possible values are:

- "`Standard`" - messages are shown in the `stdout`;

- "`Console`" - messages are shown in a console window; and
- "`File`" - messages are saved in a file.

Level
: Compulsory key that defines the logger's verbosity level. Any non negative integer number is an allowed value.

  Loggers receive messages with verbosity levels. A *m*-level logger process a *n*-level message if either *m>n* or the message is an error. Otherwise the message is ignored. Therefore, a *0*-level logger ignores all but error messages.

FileName
: This key is compulsory when *Device* is "`File`" and ignored in other cases. It specifies the output file name.

Global
: The core library can use different loggers for different purposes. Hence, users are able to build many loggers at the same time. However, all KeyValue messages are sent to the global logger. This key can assume the values `TRUE` or `FALSE` and tells KeyValue if the new logger must replace the current global logger.

8.2. *NumberOfDataSets*

This processor does not have any specific key. It returns the number of data sets currently stored by the repository.

8.3. *ListOfDataSets*

This processor does not have any specific key. It returns a vector with the names of data sets currently stored in the repository.

8.4. *DeleteDataSets*

Deletes a list of data sets from the repository. Only one key is expected:

DataSets
: This is an optional key which list the names of all data sets to be erased. If this key is ommited, then all data sets will be removed.

This processor returns the number of data sets that were effectively deleted from the repository.

9. Key resolution and the *Default* data set

Normally, when retrieving the value associated to a key in a given data set, KeyValue finds it in the same data set. However, this is not always the case. The process of finding the correct value assigned to a given key is called **key resolution** .

The most basic way to assign a value to a key is providing the key-value pair as we have seen so far. Additionally, there are three ways to import values from different keys and data sets. Those are the subjects of following sections.

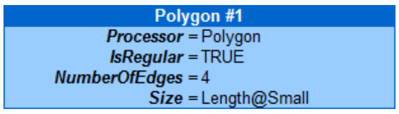
9.1. Importing a value from another key

We can import the value of a key from another key. Moreover, the source key might be in a different data set. For this purpose, instead of providing the value for the key we should put a reference in the following format:

`key-name@data-set-name`

where *key-name* is the source key and *data-set-name* is the source data set. You can leave either *key-name* or *data-set-name* blank to refer to the same key or data set.

For instance, in sheet *Key resolution and Default data set* , key *Size* in data set *Polygon #1* has the same value as *Length* in data set *Small*.

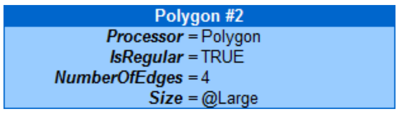


Polygon #1

Processor = Polygon
IsRegular = TRUE
NumberOfEdges = 4
Size = Length@Small

Figure 13. *Polygon #1* imports key Size from key *Length* in data set *Small*.

Data set *Polygon #2* imports key *Size* from data set *Large*.

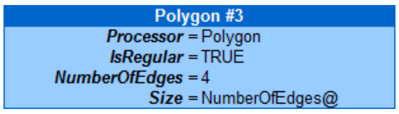


Polygon #2

Processor = Polygon
IsRegular = TRUE
NumberOfEdges = 4
Size = @Large

Figure 14. *Polygon #2* imports key Size from data set *Large*.

In data set *Polygon #3* keys *Size* and *NumberOfEdges* have the same value.



Polygon #3

Processor = Polygon
IsRegular = TRUE
NumberOfEdges = 4
Size = NumberOfEdges@

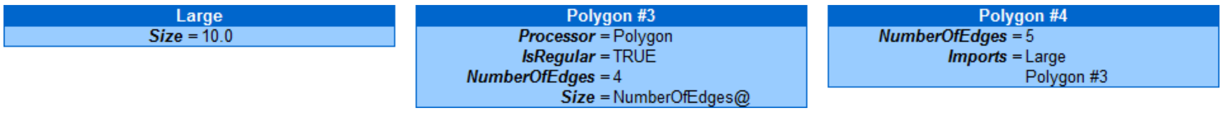
Figure 15. *Polygon #3* imports key Size from its own key *NumberOfEdges*.

9.2. Importing all key-value pairs from other data sets

We can import all key-value pairs from one or more data sets to the current one through the key *Imports*. The value associated to *Imports* must be a vector of data set names. All key-value pairs in any of those data sets are imported to the data set containing *Imports*.

Keys assigned locally, either directly or through references, take precedence over imported keys. Data sets assigned to key *Imports* are processed in the order they appear.

For instance, in sheet *Key resolution and Default data set* , *Polygon #4* imports keys first from *Large* and second from *Polygon #3* . Only keys that are not found neither in *Polygon #4* nor in *Large* will be imported from *Polygon #3*. Therefore, key *NumberOfEdges* is assigned locally, key *Size* is imported from *Large* and *isRegular* is imported from *Polygon #3* .



Large

Size = 10.0

Polygon #3

Processor = Polygon
IsRegular = TRUE
NumberOfEdges = 4
Size = NumberOfEdges@

Polygon #4

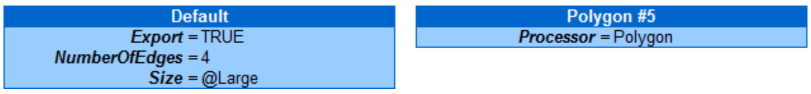
NumberOfEdges = 5
Imports = Large
Polygon #3

Figure 16. Use of key *Imports*.

9.3. Importing key-values from *Default* data set

After searching a key locally and in imported data sets, if the key is still not resolved, then KeyValue makes a last trial searching in a special data set named *Default*. To make this search effective, *Default* must have a key *Export* set to TRUE.

For instance, in sheet *Key resolution and Default data set* , *Polygon #5* imports all keys, but *Processor*, from *Default*.



Default
Export = TRUE
NumberOfEdges = 4
Size = @Large

Polygon #5
Processor = Polygon

Figure 17. *Polygon #5* imports all keys, but Processor, from *Default*.

10. Lexical conversions

Front-ends may lack representation for some of KeyValue's basic types: number, text, boolean and date. In that case lexical conversions are required. For instance, OpenOffice Calc and Excel do not have a specific representation for time. Instead, they use a double which represents the number of days since a certain epoch. Therefore, the front-end must convert from double to KeyValue's representation of time.

Moreover, lexical conversions can make user interface more friendly. For instance, OpenOffice Calc and Excel users might prefer to use "`Yes`" and "`No`" rather than the built-in boolean values (`TRUE` and `FALSE`).

Front-ends must implement all lexical converters they need. The lexical conversion cited above (from text to boolean values) is, indeed, implemented for OpenOffice and Excel add-ins. Instead of `TRUE` and `FALSE` we can give use any of the following strings:

- "`TRUE`", "`True`", "`true`", "`YES`", "`Yes`", "`yes`", "`Y`", "`y`"; or
- "`FALSE`", "`False`", "`false`", "`NO`", "`No`", "`no`", "`N`", "`n`".

Additionally, OpenOffice and Excel add-ins implement lexical conversions from text to number, that is, providing the text "`1.23`" when a number is required is the same as providing the number 1.23.

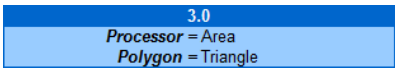
11. Key mappings

Sometimes, a text assigned to a key is mapped to some other type in a process called **key mapping**. The four types of key mappings are described in the following sections.

11.1. Object map

This is the most typical example of key mapping: An object name is mapped to the object itself.

In sheet *The KEYVALUE function* of the example workbook, the formula in cell *B8* returns the area of a certain polygon.



3.0
Processor = Area
Polygon = Triangle

Figure 18. The value assigned to key *Polygon*, *i.e.*, "`Triangle`" is mapped to an object (the triangle, itself).

Notice that value assigned to key *Polygon* is the text "`Triangle`". Rather than a text, the processor *Area* requires a polygon to computes its area. Therefore, when the processor asks for the value associated to key *Polygon*, KeyValue maps the text "`Triangle`" to a polygon which is returned to the processor.

More precisely, the text names a data set which is stored by the repository and defines an object. When the object is required the named data set is retrieved and passed to a processor (defined by key *Processor*) which creates the object. Then, the object is returned to the processor which has initiated the call.

11.2. Flag map

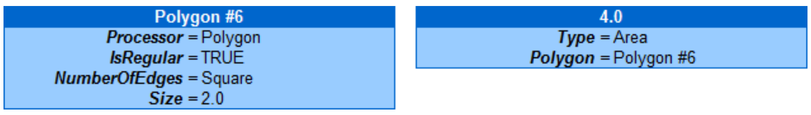
A text is mapped to some other basic type. For instance, consider a key *Month*. The user might prefer to provide text values: "`Jan`", "`Fev`", ..., "`Dec`". On the other hand, for the processor, numbers 1, 2, ..., 12 might be more convenient.

This mapping is very similar to the lexical conversion from "`Yes`" to `TRUE` as discussed in section Section 10, “Lexical conversions”. The difference is that opposite to lexical conversions, flag map depends on the key. For instance, for a key *Planet* the text "*Mar*" might be mapped to something representing the planet Mars (*e.g.* the number 4 since Mars is the forth planet of our solar system) rather than the month of March.

11.3. Partial map

Like flag map, a text is mapped into a number or date. However, the user can also provide the corresponding number or date instead of the text.

For instance, the key *NumberOfEdges* used in our example workbook implements a partial map. Its value must be an integer greater than 2. For some special values (not all) there correspond a polygon name (*e.g* "`Triangle`" for 3 or "`Square`" for 4). There is no special name for a regular polygon with 1111 edges. To see this mapping in action, go to sheet *Key mappings* and change the value of *NumberOfEdges* in data set *Polygon #6* to "`Triangle`" or "`Square`" or 1111 and see its area on *E2*.



Polygon #6
Processor = Polygon
IsRegular = TRUE
NumberOfEdges = Square
Size = 2.0

4.0
Type = Area
Polygon = Polygon #6

Figure 19. Key *NumberOfEdges* implements partial map. Assigning to it "`Square`" is the same as assign it to 4.

11.4. No map

Finally, there is the identity map (a.k.a no map): The text which is assigned to the key is retrieved by KeyValue and passed to the caller as it is.

12. Design: the basics

This section covers some basic aspects of KeyValue's design. The material is kept at the minimum just enough to give the reader all he/she needs to use KeyValue.

All KeyValue classes, functions, templates, etc. belong to namespace `::keyvalue`.

12.1. Basic types

Inspired by spreadsheet applications, KeyValue uses five **basic types**:

- bool;
- double;
- `value::Nothing`;
- `ptime`; and
- `string`.

The first two types above are C++ built-in types. The other three are library types.

KeyValue introduces `value::Nothing` to represent empty data.

To maximize portability, KeyValue uses `::std::string` and `::boost::posix_time` for strings and times, resp. These types are exported to namespace `::keyvalue` where they are called `string` and `ptime` resp.

The single-value and multi-type container for basic types is `value::Variant`.

12.2. Values

Values assigned to keys are not always given by a single `value::Variant`. They may be containers of `value::Variants`. KeyValue provides three such containers:.

- `value::Single`;
- `value::Vector`; and
- `value::Matrix`.

Actually, bridge and core library developers do not need to care neither about the containers above. Indeed, they are used exclusively inside KeyValue and at some point are converted to more standard types. More precisely, a `value::Single` is converted into an appropriate basic type a `T` while a `value::Vector` becomes a `::std::vector<T>` and a `value::Matrix` is transformed into `::std::vector<std::vector<T> >` .

Class value::Value is a single-value and multi-type container for `value::Single`, `value::Vector` or `value::Matrix`.

12.2.1. Hierarchy of types and multi-level implicit conversions

Only value::Values are returned from KeyValue to front-ends. Hence, a series of conversions must be performed when one wants to return a more basic type. For instance, suppose that a double variable x must be returned. In that case the sequence of conversions would be:

```
return value::Value(value::Single(value::Variant(x)));
```

Statements like the one above would be needed very often, which is very annoying. Fortunately, KeyValue implements a hierarchy tree of types that allow for multi-level implicitly conversions. Therefore, in the example above, the simpler statement

```
return x;
```

would be implicitly converted by the compiler into the one previously shown.

The hierarchy of types constitutes a tree where each node is defined by a specialization of template struct `Parent`.

12.3. Keys

Initially a key is just a `string` labeling a value. However, there is more inside a key that just a `string` can model. Consider the key *Dates* in the introductory example again. The value associated to it is expected to verify certain conditions:

- The corresponding `value::Value` is made of `ptimes`.
- One can expect more than one `ptime` and, then `value::Value`'s container might be a `value::Vector` (of `ptimes`).
- Since to each date corresponds a stock price, it cannot be in the future.
- Additionally, one can expect the dates to be in increasing order.

This kind of information on keys is encapsulated by a certain class. In KeyValue terminology, those classes are called **real key** and belong to namespace `::keyvalue::key`.

The class is `key::Key` is the base for all real keys. More precisely, real keys derive from `key::Traits` which derives from `key::Key`.

Actually, `key::Traits` is a template class depending on a few template parameters:

ElementType
: Type parameter which defines the type of elements in the output container. It can be bool, double, `ptime`, `string`, classes defined by the core library, etc.

ConverterType
: This template template parameter[3] defines the class responsible to convert the input value container into a more appropriate type for core library's use. (See Section 12.3.1, “Converter type”.)

MapType
: This template template parameter[3] tells how each `value::Variant` in the input container must be mapped into an ElementType. (See Section 12.3.2, “Map type”.)

12.3.1. Converter type

Conversions between KeyValue containers `value::Single`, `value::Vector` or `value::Matrix` to more standard types are responsibility of **container converter** classes.

KeyValue provides three such templates (described below) depending on a parameter ElementType.

`key::StdSingle<ElementType>`
: Converts from `value::Single` into ElementType.

`key::StdVector<ElementType>`
: Converts from `value::Vector` to `::boost::shared_ptr<std::vector<ElementType> >`.

`key::StdMatrix<ElementType>`
: Converts from `value::Matrix` to `::boost::shared_ptr<std::vector<std::vector<ElementType> > >`.

If the core library uses non-standard containers, then bridge developers have two choices. They can either use the converters above as a first step and then convert again to desired types; or they can implement new container converters that produce the desired types directly from KeyValue containers. The second option is clearly more efficient.

To implement new container converters, the reading of reference documentation of the three container converters above it strongly advisable. Moreover, their implementations can serve as samples.

12.3.2. Map type

Similarly to lexical conversions but depending on the key, sometimes, each element of the input container must be mapped to a special value. For instance, for a key *Month*, it may convenient to map `strings` "`Jan`", "`Fev`", ..., "`Dec`" into numbers 1, 2, ..., 12. This is an example of `key::FlagMap`.

Mappings are performed by classes which implement a method to convert from a `value::Variant` value into other type. Actually, they are template classes depending on a parameter named OutputType which defines but not necessarily matches the real output type. The real output type might be recovered through the member type OutputType_.

The map template classes are the following:

`key::NoMap<OutputType>`
: Through this map, a `value::Variant` holding a value *x* is mapped into an object of type OutputType which has the same lexical value as *x*. Only front-end enabled lexical conversions are considered. For

[3]Template template parameters belong to the less known features of C++ and then deserve a quick note here. Most template parameters are types. Nevertheless, sometimes a template parameter can be a template, in which case it is referred as a **template template parameter**. For instance, a template `Foo` depending on only one template template parameter might be instantiated with `Foo<std::vector>` but not with `Foo<std::vector<int> >`. Recall that `std::vector` is a *template* while `std::vector<int>` is a *type*.

instance, a `value::Variant` holding either the double 10.1 or the `string` "`10.1`" is mapped into the double (OutputType in this case) 10.1.

`key::FlagMap<OutputType>`
: Some `strings` are accepted. Each of them is mapped into a particular value of type OutputType. In the example above OutputType is double.

`key::PartialMap<OutputType>`
: A mix between `key::NoMap` and `key::FlagMap`. First, like `key::NoMap`, considering front-end enabled lexical conversions, it tries to map a `value::Variant` into an object of type OutputType which has the same lexical value as *x*. If it fails, then, like `key::FlagMap`, it tries to map a `string` into a corresponding value of type OutputType. For instance, the value for *NumberOfEdges* (of a regular polygon) must be an unsigned int greater than 2. For some special values (not all) there correspond a polygon name (*e.g*. "`Triangle`" for 3 or "`Square`" for 4). There is no special name for a regular polygon with 1111 edges.

`key::ObjectMap<OutputType>`
: This is a map where a `string` identifier is mapped into a `::boost::shared_ptr<OutputType>` pointing to an object of type OutputType. Notice that this is the only map where OutputType and OutputType_ do not match.

12.3.3. Generic keys

There are some basic properties shared by several types of keys. For instance, *Price*, *Weight*, *Size*, etc., accept only positive number for values. Although one can implement one class for each of them, this would imply in extensive code duplication. Therefore, KeyValue implements a few generic keys which can be used for keys having basic properties. Only very specific and application dependent keys need to be implied as new real keys.

All generic keys set the key label at construction time. They are:

`key::Single<ElementType>`
: Key for a single value of type ElementType. No constraints on the value are set.

  Example: A key labeled `Number` which accepts any double value is defined by

```
key::Single<double> key("Number");
```

`key::Vector<ElementType>`
: Key for a vector of values of type ElementType with no constraints on them. A restriction on the size of the vector might be set at construction.

  Example: A key labeled `Names` which expects a vector of 5 strings is defined by

```
key::Vector<string> key("Names", 5);
```

`key::Matrix<ElementType>`
: Key for a matrix of values of type ElementType with no constraints. A restriction on the matrix dimension can be set at construction.

  Example: A key labeled `Transformation` which accepts a *2x3* matrix is defined by

```
key::Matrix<double> key("Transformation", 2, 3);
```

`key::Positive`
: Key for a positive number.

  Example: A key labeled `Price` is defined by

```
key::Positive key("Price");
```

`key::StrictlyPositive`
: Key for a strictly positive number.

  Example: If in the key in the previous example could not accept the value 0, then it would be defined by

  ```
  key::StrictlyPositive key("Price");
  ```

`key::Bounded<ElementType, Bound1, Bound2>`
: Key for a single bounded values of type ElementType. Template template parameters[3] Bound1 and Bound2 define the bound types and can be either `key::Greater`, `key::Geq` (greater than or equal to), `key::Less` or `key::Leq` (less than or equal to).

  Example: A key labeled `Probability` accepting any double value from and including 0 up to and including 1 is defined by

  ```
  key::Bounded<double, key::Geq, key::Leq> key("Probability", 0.0, 1.0);
  ```

`key::MonotoneBoundedVector<ElementType, Monotone, Bound1, Bound2>`
: Key for vectors whose elements are monotonic and/or bounded. Template template parameter[3] `Monotone` defines the type of monotonicity and can be either `key::NonMonotone`, `key::Increasing`, `key::StrictlyIncreasing`, `key::Decreasing`, `key::StrictlyDecreasing`. Bound1 and Bound2 are as in `key::Bounded`. Additionally, a constraint on the vector size can be set at construction.

  Example: A key labeled `Probabilities` accepting 10 strictly increasing numbers from and excluding 0 up to and including 1 is defined by

  ```
  key::Bounded<double, key::StrictlyIncreasing, key::Greater, key::Leq>
    key("Probabilities", 0.0, 1.0, 10);
  ```

12.4. DataSet

Key-value pairs are stored in `DataSets`. This class implements methods `getValue()` and also `find()` to retrieve values assigned to keys. Both methods receive a real key and processes all the information about the expected value encapsulated in it. For instance, suppose that the variable `today` holds the current date and consider a key *BirthDates* which corresponds to a vector of increasing dates, supposedly, in the past.

An appropriate real key is then:

```
key::MonotoneBoundedVector<ptime, key::Increasing, key::Leq>
  births("BrithDates", today);
```

Therefore, if key *BirthDates* is in a `DataSet` named `data`, the result of

```
data.getValue(births);
```

is a `::boost::shared_ptr< const ::std::vector<ptime> >` such that the elements of the vector pointed by this pointer are in increasing order and before (less than or equal to) `today`. The caller does not need to check that.

Since the type returned by `getValue()` depends on the real key it receives, this method is a template function. The same is true for `find()`.

The difference between `getValue()` and `find()` concerns what happens when the key is not resolved. The former method throws an exception to indicate the failure while the latter returns a null pointer. In practice, `getValue()` is used for compulsory keys and `find()` for optional keys. A typical use of `find()` is as follows:

```
bool foo(false);
if (bool* ptr = data.find(key::Single<bool>("Foo")))
  foo = *ptr;
```

In the code above `foo` is `false` unless key *Foo* is found, in which case, `foo` gets the given value.

12.5. Processors

All builders and calculators derive from protocol class `Processor`. This class declares three pure virtual methods: `getName()` and two overloads of `getResult()`. The former method returns the name under which the processor is recognized by key *Processor*. The first overload of `getResult()` takes a `DataSet` parameter while the second one takes a `value::Variant`. The difference between them is explained below.

In general, the information that processors need to perform their duties is so rich that must be stored in a `DataSet`. Nevertheless, in some particular cases, a single `value::Variant` might be enough. For instance, consider a builder that creates a curve given a few points on it. Normally, this processor needs the set of points together with interpolator and extrapolator methods. In this general case, a `DataSet` is necessary to hold all this information. However, when the curve is known to be constant, then a single number - the constant - is enough. Rather than creating a `DataSet` to store a single number, it would be more convenient if the processor could accept just this value (or more generally, a `value::Variant`). This is, indeed, the case.

Actually, builders and calculators are specializations of template classes `Builder` and `Calculator`, resp. They depend on a parameter type named either *OutputType* (for a builder) or *Tag* (for a calculator). The primary role of this parameter is to distinguish between different specializations. Additionally, in the case of a builder, it also defines the type of object constructed.

The header files of `Builder` and `Calculator` implement the two overloads of `Processor::getResult()`. The first overload forwards the call to either `Builder<ObjectType>::getObject()` or `Calculator<Tag>::getValue()`. These methods must be implemented by bridge developers whenever the template classes are used.

As explained earlier, processing a `value::Variant` does not always make sense. Therefore, the second overload of `Processor::getResult()` throws an exception. When this processing does make sense, then template classes `XBuilder` and `XCalculator` which extend their base classes `Builder` and `Calculator`, resp., must be used. Their header files reimplement the second overload of `Processor::getResult()` forwarding the call to `XBuilder<ObjectType>::getObject()` and `XCalculator<Tag>::getValue()`. As before, these two methods must be implemented by bridge developers whenever the extended template classes are used.

12.6. Exceptions and Messages

The abstract class `Message` defines the interface for all types of messages. `MessageImpl` is a template class which implements `Message`'s pure virtual methods. There are six different specializations of `MessageImpl` with corresponding typedefs:

- `Error`;
- `Logic`;
- `Info`;
- `Warning`;
- `Report`; and
- `Debug`.

They define `operator&()` to append formated data to themselves. A typical use follows:

```
Info info(1);  // Create a level-1 Info message.
size_t i;
std::vector<double> x;
//...
info & "x[" & i & "] = " & x[i] & '\n';
```

Similarly, `exception::Exception` is an abstract class whose pure virtual methods are implemented in template class `exception::ExceptionImpl`. This template class has a member which is an instance of `MessageImpl`. The exact type of this instance is provided as a template parameter of `exception::Exception`. There are two specializations of `exception::Exception` with corresponding typedefs:

- `RuntimeError` (having an `Error` member); and
- `LogicError` (having a `Logic` member).

`RuntimeError` indicates errors that can be detected only at runtime depending on user data. `LogicError` indicates errors that should be detected at development time. In other terms, a `LogicError` means a bug and is thrown when a program invariant fails. It is mainly used through macro `KV_ASSERT` as in

```
KV_ASSERT(i < getSize(), "Out of bound!");
```

To keep compatibility with exception handlers catching standard exceptions, `RuntimeError` derives from `::std::runtime_error` while `LogicError` derives from `::std::logic_error`.

Method `exception::ExceptionImpl::operator&()` provides the same functionality of `MessageImpl::operator&()`. Example:

```
if (price <= 0.0)
  throw RuntimeError() & "Invalid price. Expecting a positive number. Got " &
    price;
```

Other more specific exception classes are implemented to indicate errors that need special treatment. They all derive from either `RuntimeError` or `LogicError`.

13. How to implement the bridge library

The bridge library connects KeyValue and core libraries. KeyValue comes with an example bridge which can be used as a model for bridge developers.

The implementation of a bridge library is composed of three tasks.

- Implementing class `Bridge`:

  This class provides information about the core library, *e.g.,* its name and greeting messages. (See Section 13.1, “How to implement class `Bridge`”.)

- Implementing and registering processors:

  The bridge implement a certain number of processors to be called by users through key *Processor*. (See Section 13.2, “How to implement a processor”.)

  The class `ProcessorMngr` is responsible for retrieving a processor provided its name. Therefore, every processor must be known by this class which is done by including the processor's address into its array member `ProcessorMngr::processors_`.

  The suggested registration method is as follows. Bridge developers copy files `bridge-example/Register.h` and `bridge-example/Register.cpp` to their own source directory to be compiled and linked as their other source files. File `Register.cpp` is left as it is while file `Register.h` is edited (see instructions there in) to list names of pointers pointing to the various processors.

- Implementing keys:

  KeyValue comes with generic keys *(e.g.* `key::Positive`, which is used for keys whose values are positive numbers). Other application specific keys can be implemented. (See Section 13.3, “How to implement a key”.)

13.1. How to implement class `Bridge`

Some methods of class `Bridge` are implemented by KeyValue itself. However there are three public methods which are left to the bridge developer. (See example in `bridge-example/bridge-example/Bridge.cpp`):

```
const char*
getCoreLibraryName() const;
```

Returns the name of the core library. The result also names the function called in OpenOffice Calc or Excel spreadsheets and, for that reason, must be a single word (no white spaces). Otherwise front-ends might get in trouble.

```
const char*
getSimpleInfo() const;
```

Returns a simple description (one line long) of the core library. This message is used by Excel add-in manager.

```
const char*
getCompleteInfo() const;
```

Returns a more detailed description of the core library. This message is presented by loggers when they are initialized.

13.2. How to implement a processor

The two types of processors, builders and calculators, are implemented in similar ways. Let us see how to implement a calculator and then the differences for builders.

13.2.1. Implementing a calculator

Basic calculators (those that cannot process `value::Variants`) are specializations of template class `Calculator`. (See example in `bridge-example/processor/Area.cpp`.)

Each specialization is uniquely identified by a type Tag which, normally, is a forward-declared-only class:

```
class Tag;
```

Given the type Tag, developers must implement two of `Calculator<Tag>`'s methods:

```
template <>
const char* Calculator<Tag>::getName() const;
```

Returns the name under which `Calculator<Tag>` is recognized by key *Processor*.

```
template <>
value::Value
Calculator<Tag>::getValue(const DataSet& data) const;
```

Processes `DataSet data` and returns a `value::Value` computed based on `data`'s key-value pairs. Recall that `value::Value` belongs to the hierarchy of types which allows for multi-level implicit conversions. (See Section 12.2.1, “Hierarchy of types and multi-level implicit conversions”.) Therefore, any type below `value::Value` in the hierarchy might be returned without further ado.

As explained earlier, the processor must be registered into the `ProcessorMngr`. The suggested method [23] needs a pointer to `Processor` in namespace `::keyvalue::bridge` pointing to the instance of `Calculator<Tag>` returned by `Calculator<Tag>::getInstance()`:

```
namespace keyvalue {
  namespace bridge {
    Processor* calculatorTag = &Calculator<Tag>::getInstance();
  }
}
```

Then, this pointer must be listed in the header file `Register.h` as explained there in.

13.2.2. Implementing an extended calculator

Extended calculators are specializations of template class `XCalculator`. Given a type Tag, `XCalculator<Tag>` derives from `Calculator<Tag>` and all the instructions above to implement `Calculator<Tag>` apply. Additionally, the following method must be implemented:

```
template <>
value::Value
XCalculator<Tag>::getValue(const value::Variant& data) const;
```

This methods processes `value::Variant data` and overriding the implementation provided by `Calculator<Tag>` (which throws an exception).

13.2.3. Implementing a builder

Basic builders are specializations of template class `Builder`. (See example in `bridge-example/processor/Polygon.cpp`.) Similarly to `Calculator`, each specialization of `Builder` is uniquely defined by a template type parameter OutputType. Additionally, for builders this type also defines the type of object built. Normally, it will be a type defined by the core library.

Given OutputType, the method which returns `Builder<ObjectType>`'s name has the following signature:

```
template <>
const char* Builder<OutputType>::getName() const;
```

The registration is similar to that of a `Calculator`: One have to define a pointer to the unique instance of `Builder<ObjectType>` and list this pointer in `Register.h`.

```
namespace keyvalue {
  namespace bridge {
    Processor* builderObjectType = &Builder<ObjectType>::getInstance();
  }
}
```

The method which builds from a `DataSet` has the following signature:

```
template <>
shared_ptr<ObjectType>
Builder<ObjectType>::getObject(const DataSet& data) const;
```

KeyValue implements a memoization system to prevent rebuilding an object when the input key-value pairs in `data` have not changed since last built. To use this feature, after having retrieve all values which define the object to be build, `Builder<ObjectType>::getObject()` must call `data.mustUpdate()` which returns `true` if the object must be rebuilt (or `false`, otherwise). If the object does need to be rebuilt, then `Builder<ObjectType>::getObject()` builds a new object and returns a shared pointer to it. Otherwise, `Builder<ObjectType>::getObject()` returns a null `shared_ptr<ObjectType>`. For instance, in `bridge-example/processor/Polygon.cpp` we have (recall that `Regular` derives from `Polygon`):

```
template <>
shared_ptr<Polygon>
Builder<ObjectType>::getObject(const DataSet& data) const {
  // ...
  if (data.mustUpdate())
```

```
    return shared_ptr<Polygon>(new Regular(nEdges, size));
  // ...
  return shared_ptr<Polygon>();
}
```

13.2.4. Implementing an extended builder

Given ObjectType the extended builder for this type is `XBuilder<ObjectType>` which derives from `Builder<ObjectType>`. All the instructions above apply and, additionally, the following method, which builds from a `value::Variant`, must be implemented:

```
template <>
shared_ptr<ObjectType>
XBuilder<ObjectType>::getObject(const value::Variant& data) const;
```

For this method the memoization explained above does not apply since there is no method `value::Variant::mustUpdate()`.

13.3. How to implement a key

Key functionalities belong to namespace `::keyvalue::key` and all keys should be in this namespace as well.

The basics for implementing keys were explained in Section 12.3, “Keys”. In particular, we have seen that all keys derive from template `key::Traits`. For instance,

```
namespace keyvalue {
  namespace key {
    class MyKey : public Traits<double, StdSingle, NoMap> {
      // ...
    };
  }
}
```

is the prototype for a key accepting a single double value which is not mapped. (Actually, the second and third of `key::Traits`' parameters above are the default choices and, then, could be omitted.)

The choices of `key::Traits` parameters impose some methods to be implemented by derived classes. Those methods are divided in two categories: checking- and mapping- methods.

13.3.1. Checking methods

The third parameter of `key::Traits`, namely MapType, is a template class which defines the type of output value, OutputType_. More precisely, this is the type that one gets when passes the key (*i.e.,* the class derived from `key::Traits`) to DataSet`::getValue()`.

The checks performed on the output value depend on its type. For instance, one can check the size of a vector but not that of a single. Regardless the ConverterType, `key::Traits` implements all required checking methods. Actually, the provided implementations accept all values (no check at all). When the developer wants a proper check, then the corresponding method can be overridden and reimplemented. To indicate invalid values `RuntimeError` exception must thrown.

KeyValue implements three templates that can be assigned to ConverterType. They depend on a type parameter ElementType which, in general, matches its homonym provided to `key::Traits`. The only exception is when the MapType parameter given to `key::Traits` is `key::ObjectMap`. In this case, the ConverterType is instantiated for `::boost::shared_ptr<ElementType>`.

`key::StdSingle<ElementType>`
: This is the default choice and can be omitted when `key::Traits`' third parameter is so.

For this choice OutputType_ is the same as ElementType and the method called to validate the output has the following signature:

```
void
checkOutput(const OutputType_& output) const;
```

`key::StdVector<ElementType>`

In this case, OutputType_ is `::boost::shared_ptr<std::vector<ElementType> >`. The method that validates the output has the same signature as for `key::StdSingle` seen above.

There is a method for checking the vector size, declared as follows:

```
void
checkSize(size_t size) const;
```

Additionally, there is a method to check the output while it is still being calculated. This is useful to indicate errors as earlier as possible. For instance, consider a vector which is expected to have a huge number of increasing elements. If the second element is not greater than the first one, the method can immediately spot the problem and avoid to process from the third element onwards. The signature is

```
void
checkSoFar(const ConverterType<ElementType>& container) const;
```

Notice that it receives a ConverterType<ElementType>, which in this case, is `key::StdVector<ElementType>`. This type provides accessor methods to the output vector being constructed.

`key::StdMatrix<ElementType>`

Here OutputType_ is `::boost::shared_ptr<std::vector<std::vector<ElementType> > >`. The method that validates the output has the same signature as for `key::StdSingle` seen above. The method for checking the matrix dimensions is

```
void
checkSize(size_t nRows, size_t nCols) const;
```

Finally, a method for checking the output as the computation runs has the same signature as for `key::StdVector` seen above. However, here ConverterType<ElementType> is `key::StdMatrix<ElementType>`.

13.3.2. Mapping methods

Third parameter of `key::Traits`, namely MapType, defines the type of mapping as introduced in Section 12.3.2, “Map type”. The value assigned to this parameter should be a template class chosen among four possibilities.

Two map types when selected do not impose any constraint on `key::Traits` derived classes. They are `key::NoMap` and `key::ObjectMap`. Actually, they do not need to be explicitly provided by the user since the compiler will automatically select one of them when MapType is omitted. The choice between those two follows a simple rule: if `key::Traits` parameter ElementType is bool, double, `string`, `ptime` or unsigned int, then `key::NoMap` will be selected; otherwise `key::ObjectMap` will because the compiler assumes that ElementType is a type defined by the core library and for which a `Builder` specialization is implemented.

Luckily the constraint imposed on `key::Traits` derived classes when either `key::FlagMap` or `key::PartialMap` is selected is just a matter of implementing one method with the following signature:

```
OutputType
get(const string& name) const = 0;
```

Here, OutputType is the same as the parameter ElementType used to instantiate `key::Traits`. This method receives a `string` object and maps it to the correct value of type OutputType or, if it fails, throws a `RuntimeError`.