User's Manual - version: 0.3

Cassio Neri

Table of Contents

1. Introduction

KeyValue is a cross-platform library for making C++ objects accessible through LibreOffice[1] Calc, Excel and other front-ends. Experience of spreadsheet users is enhanced by an object model and a handy key-value based interface.

KeyValue does more than just helping 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 later use at user's request. Additionally, KeyValue provides a set of services for 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 LibreOffice 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.”.

[1]As many LibreOffice users are aware, this office suite is a fork of OpenOffice.org started in 2010. KeyValue dates prior to this fork and used to provide an OpenOffice.org Calc add-in. However, since release 0.3, KeyValue replaced the OpenOffice.org add-in by the LibreOffice one. As of this writing, the two suites are still very similar and the LibreOffice extension will probably work for OpenOffice.org as well.

Notice also that, for the aforementioned reasons, one can find several mentions to OpenOfice.org in KeyValue's code, file names, documentation, etc. Nevertheless, we emphasize that KeyValue focuses on LibreOffice and no longer on OpenOffice.org.

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 `bridge-example/bridge-example/processor/LowTime.cpp`) would be similar to:

```
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

  const ::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 deeply into 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 the key *Dates* including the label "`Dates`" that appears on the spreadsheet. Its type is an instantiation of `key::MonotoneBoundedVector` for type parameters `ptime` and `key::StrictlyIncreasing`. This means that the expected type of value is a `::std::vector<ptime>`[2] whose elements are in increasing order.

A few generic keys like `key::MonotoneBoundedVector` are implemented. We can implement application specific keys when no generic key fits our needs or whenever this proves to be convenient. For instance, implementing a class named `Dates` can be useful if key *Dates* is used very often. In such case, `Dates` would hold all the above information and line *B* would be replaced by

```
const Dates keyDates;
```

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

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 the key *Prices*: the label "`Prices`" and its expected type of value, that is, a `::std::vector<double>` of size `dates.size()` with elements greater than or equal to zero.

E:
This line of code retrieves the `::std::vector<double>` containing the prices and, if execution gets to next line, we can be sure that `prices` and `dates` have the same size and all `prices`' elements are positive. Otherwise an exception is thrown.

F - G:
This bit of code could be part of the library which KeyValue helps to make accessible through LibreOffice Calc or Excel. We placed it here for illustration purposes only.

H:
While the type returned by `Calculator<LowTime>::getValue()` is `value::Value` the code above returns a `ptime`. For convenience, KeyValue implements implicit conversions to `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 LibreOffice Calc and Excel. A third example is an XML parser. 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 completly independent, that is, does not `#include` any header file from other layers.

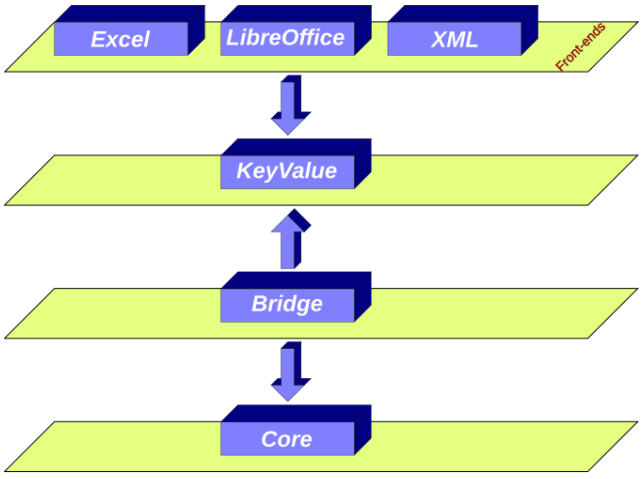

Figure 2. KeyValue's design.

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 the front-ends with KeyValue's help. This layer is also independent. As previously mentioned, the code between lines *F* and *G* in the example above would be better placed in the core library.

The **bridge** layer connects KeyValue and the core library. Bridge `#includes` files from both layers that it is connected to. The code given in the example above would be part of this layer.

In addition to KeyValue layer, the distibuted code contains the front-ends (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. Downloading and installing

KeyValue is available in standard formats at SourceForge.

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

Just download and unpack it into your hard disk.

Windows Vista users might need to perform an extra step. As we shall see below, KeyValue's build system relies on Cygwin. For an obscure reason in *some but not all* machines, Cygwin fails to copy 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 depend on the user's purpose. The following sections explain in detail the need for these tools and libraries.

2.1. Compiler

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

Most of GNU/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/visualstudio/en-us/products/2008-editions/express`

Editions differ mainly in their IDEs. However, there are also compiler differences. During KeyValue's development we came across lines of code that the Professional Edition did compile whereas the Express Edition failed. Some effort has been made to maintain compatibility with both editions.

2.2. Building tools

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

GNU/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 required. Normally, it is part of the x11 or xorg packages. To check whether you 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 these tools but, unfortunately, they are not directly available. Therefore, Cygwin (see Section 2.4, “Cygwin”) will be required to provide a GNU/Linux-compatibility layer to Windows systems.

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 and intrusive 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 into the hard disk.

Boost is available for download at its SourceForge page:

`http://sourceforge.net/projects/boost/files/boost`

2.4. Cygwin

KeyValue is a cross platform library for GNU/Linux and Windows systems. Its build system relies on tools that are very popular on GNU/Linux systems but not on Windows. For that reason, Windows users must install Cygwin to have a GNU/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 choosing the packages to install, make sure that the following items are selected:

- Archive/zip (needed only if we want to build the LibreOffice 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; and
- Utils/diffutils.

Cygwin comes with a small tool called **link.exe** 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.exe** because this is also the name of the Microsoft Visual Studio linker and, therefore, they conflict. A workaround is renaming Cygwin's **link.exe** to, say, **link-original.exe**. Open a **bash shell** by clicking on Start / Programs / Cygwin / Cygwin Bash Shell and type the command below followed by **Enter**.

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

On many occasions we need to type 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. LibreOffice SDK

KeyValue comes with a LibreOffice Calc add-in for GNU/Linux and Windows systems. To build this add-in, one must install the LibreOffice SDK.

The LibreOffice Calc add-in has been tested with some 3.x.x versions of LibreOffice and LibreOffice SDK. It probably works with all 3.x.x versions.

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

`http://www.libreoffice.org/download`

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 by new Excel releases, then the add-in should work with them as well. However, KeyValue does not work with 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. Configuring and building

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. This file contains detailed explanations on how to set up KeyValue.

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 needed to follow the tutorial given in Section 4, “Getting started with KeyValue”.

In a bash shell console, go to KeyValue's home directory. For instance, in GNU/Linux, assuming KeyValue was unpacked in `/home/cassio/keyvalue-0.3`, type

```
$ cd /home/cassio/keyvalue-0.3
```

Under Cygwin (*i.e.* Windows) one has to prefix the directory name by the drive letter. Supposing that KeyValue was unpacked in `C:\Users\cassio\Documents\keyvalue-0.3`, type

```
$ cd C:/Users/cassio/Documents/keyvalue-0.3
```

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 them:

```
$ make help
```

This also shows the list of projects to be compiled (as set by `config.mak`).

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 follow these steps:

- 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.EXE` (*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.bin` (*e.g.* `C:\Program Files\LibreOffice 3.4\program\soffice.bin`)
Command Arguments	`-nologo -calc`
Environment	`PATH=C:\Program Files\LibreOffice 3.4\program;C:\Program Files\LibreOffice 3.4\URE\bin;C:\Program Files\LibreOffice 3.4\Basis\program`

4. Getting started with KeyValue

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

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

http://www.libreoffice.org

It is worth mentioning one interface difference between LibreOffice 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, LibreOffice Calc recalculates only if it believes the cell's content has changed. In particular, **F2** followed by **Enter** recalculates a cell formula in Excel but not in LibreOffice Calc. To force LibreOffice 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 LibreOffice 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, on a spreadsheet, would fit in a single cell. For instance, the number 1.0 or the text "Foo".

Vector
: Is a collection of data that, on a spreadsheet, would fit in an one-dimensional range of cells like *A1:J1* or *A1:A10*. More precisely, when these cells are one beside another in a row we call it a **row vector** (*e.g. A1:J1*). When the cells are one above another 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, on a spreadsheet, would fit in a two dimensional range of cells like *A1:B2*. In particular, single and vector are matrices.

After building KeyValue with its core and bridge examples (see Section Section 3, “Configuring and building”), under KeyValue's home directory, we should have a ready-to-use LibreOffice 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 LibreOffice add-ins.”.

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

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

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

Notice that a console window pops up. KeyValue uses it for output, notably 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)
```

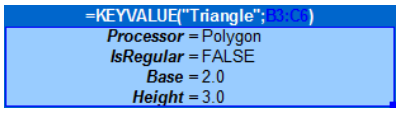

Figure 3. Data set *Triangle.*

This function call is meant to build a triangle.

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

There are no functions such as `BuildPolygon`, `CalculateArea` or anything similar. Indeed, regardless the core library, KEYVALUE is the only function exported to LibreOffice Calc.

Actually, the name of this function is defined by the bridge library. In the examplary bridge, this function is called KEYVALUE and, for the sake of concreteness, in this document we shall always assume this name.

Having just one function 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*.

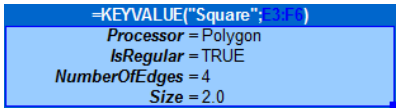

Figure 4. Data set *Square*

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 in these examples, often the data set name is the result returned from KEYVALUE to LibreOffice Calc.

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

Other KEYVALUE's parameters are optional and define key-value pairs following patterns as 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 the parameters down into 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*.

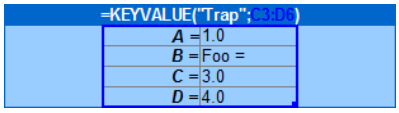

Figure 6. "Foo =" seems to define a key but it does not.

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 either a single, vector or matrix, which will be the associated value. Those three possibilities are shown on the sheet *Key-value patterns*.

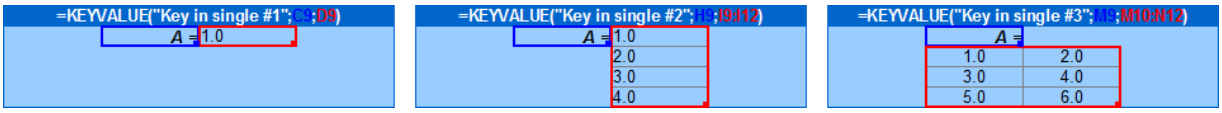

Figure 8. Key in single pattern.

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 followed by 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]).

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 first cell of second column is not a key (*i.e.* it does not contain text verifying the necessary conditions [10]).

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 more aesthetically 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.”.

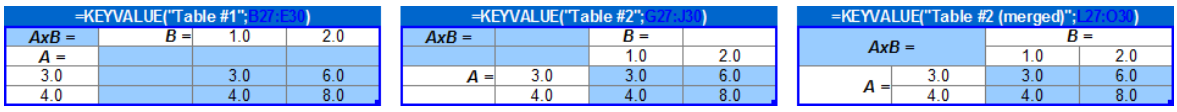

Figure 11. Table pattern. *A* is the row key, *B* is the column key and *AxB* is the table key.

7. Reserved keys

Some keys are reserved to KeyValue's use. They are explained in the sequel.

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. In any data set, the value assigned to key *Processor* (if present) names the processor which process the data set.

For instance, on sheet *Reserved keys*, the formula in *B2* creates data set *A* which selects processor *Polygon* while the one in *E2* creates an anonymous data set which selects processor *Area*. Recalculate [8] *B2* to verify on 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: Processor
  Value #3: [Single] Polygon
  Key   #4: Size
  Value #4: [Single] 1
[Debug  ] DataSet:
  Size    : 2
  Key   #1: Polygon
  Value #1: [Single] A
  Key   #2: Processor
  Value #2: [Single] Area
```

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 to be displayed on the spreadsheet are called **calculators** (*e.g. Area*).

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

7.1.1. Commands

Some processors might perform their tasks on empty data sets or, more precisely, on data sets whose unique key is *Processor*. For instance, as we see in Section 8.4, “*DeleteDataSets*”, the processor *DeleteDataSets* resets the data set repository when key *DataSets* is not present. The bridge library can declare such processors as commands.

Front-ends may provide special support for commands. For instance, the Excel add-in presents a menu from which one can call any command. The add-in creates an anonymous data set with key *Processor* and whose value is *DeleteDataSets*. Since the data set is anonymous it is immediately processed (as explained in Section 7.2, “*ProcessNow*”).

Notice that the name shown on the menu might be different of processor's name. In our example, processor *DeleteDataSets* becomes *Reset repository*.

7.2. *ProcessNow*

On sheet *Reserved keys*, the formula in *B2* actually does not build any polygon. Indeed, for non anonymous 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 laterly *might* be used to build a polygon. In this example it will happen when we request its area in *E2*.

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 the front-end. Otherwise, KeyValue just creates and stores the data set for later use and the result returned to the front-end is the data set name. Change cell *C10* to TRUE and FALSE and check the logger to see when the processor is called.

Anonymous data sets are always processed and, therefore, *ProcessNow* is ignored. 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 returned to the front-end: As a column vector, as a row vector or unchanged, *i.e.*, 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 (see Section 9, “Key resolution and the *Default* data set”) 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* are implemented by the bridge-example which comes with KeyValue. This bridge is intent to be used only as an example, and should not be linked with more serious applications (yours). Therefore, these processors will not be available. However, a few processors are implemented by KeyValue itself and not by the bridge library. 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 sent to stdout;
- "Console" - messages are displayed 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 shows a *n*-level message if *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. This processor is a command and the Excel add-in provides a menu entry to call this processor.

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. This processor is a command and the Excel add-in provides a menu entry (named Reset Repository) to call this processor.

9. Key resolution and the *Default* data set

Normally, when retrieving the value associated to a key in a given data set, KeyValue finds the value in the data set containing the key. 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.

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 name of source key and *data-set-name* is the name of source data set. You can leave either *key-name* or *data-set-name* blank to refer to the current key or data set. For instance, on sheet *Key resolution and Default data set*, key *Size* in data set *Polygon #1* has the same value as *Length* in data set *Small*.

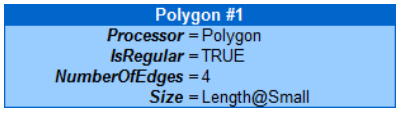

Figure 13. *Polygon #1* imports key Size from key *Length* in data set *Small*.

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

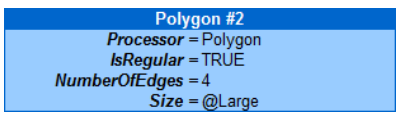

Figure 14. *Polygon #2* imports key Size from data set *Large*.

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

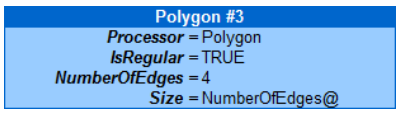

Figure 15. *Polygon #3* imports key Size from its own key *NumberOfEdges*.

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

We can import all key-value pairs from one or more data sets into 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 these 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, on 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*.

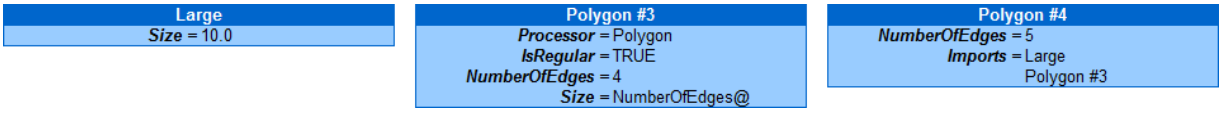

Figure 16. Use of key *Imports*.

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

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, LibreOffice Calc and Excel do not have specific representations 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, LibreOffice 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 LibreOffice and Excel add-ins. Instead of TRUE and FALSE we can use any of the following strings:

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

Additionally, LibreOffice 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.

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

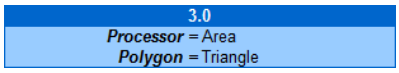

Figure 18. The value assigned to key *Polygon*, *i.e.*, "`Triangle`" is mapped to an object (the triangle, itself).

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 passed over to the processor.

More precisely, the text names a data set which is stored by the repository and defines an object. When an object is required the named data set is retrieved and passed over 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 the 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 the 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*.

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. KeyValue's 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 she needs to develop her application using KeyValue.

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

12.1. Basic types

The five so called **basic types** are:

- `bool`;
- `double`;
- `ptime`;
- `string; and`.
- `unsigned int`.

Additionally, 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 (excluding `unsigned int`) is `value::Variant`.

12.2. Values

The value assigned to a key is not necessarily a single `value::Variant`. It may be a container of `value::Variants` as well. KeyValue provides three such containers:

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

Actually, bridge and core library developers do not need to care about these containers. 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 `T` while a `value::Vector` becomes a `::std::vector<T>` and a `value::Matrix` is transformed into a `::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 value `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 often and this is very annoying. Fortunately, KeyValue implements a hierarchy tree of types that allow for multi-level implicitly conversions. Therefore, in the example above, the compiler automatically replaces the simpler statement

```
return x;
```

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 text 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. Its associated value is expected to verify certain conditions:

- The corresponding `value::Value` is made of `ptime`s rather than, say, `double`s.
- Given the plural in *Dates*, one can expect more than one `ptime` and then, `value::Value`'s content might be a `value::Vector` (of `ptime`s).
- Since each date corresponds to a stock price, these dates cannot be in the future.
- Additionally, one can expect the dates to be in increasing order.

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

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

Actually, `key::Traits` is a template class depending on a few 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` object in the input container must be mapped into an `ElementType` value. (See Section 12.3.2, “Map type”.)

12.3.1. Converter type

Conversions between KeyValue containers `value::Single`, `value::Vector` and `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` to `ElementType`.

[3]Template template parameters are one of the least known features of C++ and 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 class* while `::std::vector<int>` is a *class*.

`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 learn how to implement new container converters, the reading of the reference documentation of three container converters above it strongly advised. Moreover, their implementations can serve as samples for implementing new ones.

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 be 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` into other types. Actually, they are template classes depending on a parameter named `OutputType` which defines (but not necessarily matches) the actual output type. The actual 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 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 `string` values are accepted others not. The accepted ones are mapped into particular values of type `OutputType`. In the example of key *Month* above, `OutputType` can be `double`, `unsigned int` or an `enum` type.

`key::PartialMap<OutputType>`
Half way between `key::NoMap` and `key::FlagMap`. First, similarly to `key::NoMap` and considering front-end enabled lexical conversions, it tries to map a `value::Variant` value 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 (but 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 pointer to an object of type `OutputType`. Notice that this is the only map where `OutputType` and `OutputType_` differ.

12.3.3. Generic keys

Some properties are shared by several types of keys. For instance, *Price*, *Weight*, *Size*, etc., accept only positive values. Although one can write one class for each of them, this would imply unecessary code duplication. To avoid this, KeyValue implements a few generic keys and then, only very specific and application-dependent keys need to be written as new real keys.

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

`key::Single<OutputType>`
Key for a single object 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 objects of type `ElementType` with no constraints on them. A restriction on the size of the vector might be set on 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 objects 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 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 value of type `ElementType`. Template template parameters[3] `Bound1` and `Bound2` define the bound types and can be either `key::NoBound`, `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` or `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 `find()` to retrieve values assigned to keys. Both methods receive a real key and processes all the information about the expected value encapsulated by the key. For instance, suppose the variable `today` holds the current date and consider a key *BirthDates* which corresponds to a vector of increasing dates, supposedly, in the past or today.

An appropriate real key is then:

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

Therefore, if key *BirthDates* belongs to a `DataSet` `data`, the result of

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

is a `::boost::shared_ptr<::std::vector<ptime>>` such that the elements of the pointed vector are in increasing order and before (less than or equal to) `today`. If the input does not verify this constraint, then an exception is thrown to inform the user about the failure. Therefore, the caller does not need to check the constraints.

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 whereas the latter returns a null pointer. In practice, `getValue()` is used for compulsory keys and `find()` for optional ones. A typical use of `find()` 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 `data`, in which case, `foo` gets the given value.

12.5. Processors

All builders and calculators derive from class `Processor`. This class declares two pure virtual methods: `getName()` and `getResult()`. The former method returns the name under which the processor is recognized by key *Processor*. The second gets the result of processing a `DataSet`.

Actually, builders and calculators are specializations of template classes `Builder` and `Calculator`, resp. They depend on a parameter type named `Tag` whose primary role is distinguishing different specializations.

`Builder` and `Calculator` specializations may implement different features which affects their declarations and implementations. Therefore, different specializations of `Builders` and `Calculators` might derive from different base classes and implement different methods. Rather than declaring the specializations from scratch, providing all their base classes and declaring all necessary methods, helper files `keyvalue/mngt/DeclareBuilder.h` and `keyvalue/mngt/DeclareCalculator.h` should be used. (See `Builder` and `Calculator` for details.)

12.5.1. Commands

A `Processor` able to process an empty `DataSet` might be declared a `Command`. More precisely, if the `Processor` is `Builder<Tag>`, then its `getObject(const DataSet& data)` method might do its job ignoring `data` (e.g. *ListOfDataSets* and *NumberOfDataSets*). Another possibility is when the method does look up values in `data` but, failing to find any, can still do its job considering default values for the keys, with or without intervention of *Default data set*.

In the cases above, `Builder<Tag>` might be declared a `Command` by deriving from this class. Similar arguments hold for `Calculator<Tag>`.

When a `Processor` is a `Command`, front-ends might take advantage of this fact and provide shortcuts or menu entries to call the `Processor` without asking for additional input.

12.5.2. Building from a single value

In general, the input data required by `Builders` are so rich that must be stored in a `DataSet`. Nevertheless, in some cases, a single `value::Variant` might be enough. For instance, consider a builder that creates a function given a few points on its graph. Normally, this `Builder` needs the set of points, an interpolator and an extrapolator. A `DataSet` is necessary to hold all this information. However, when the function is known to be constant, then a single number # the constant # is enough to build the function. Rather than creating a `DataSet` to store a single `double` value, it would be more convenient if the `Builder` could accept just this value (or more generally, a `value::Variant`). This is, indeed, the case.

Any `Builder` specialization able to build from a `value::Variant` must derive from template class `BuilderFrom`. This template class depends on `ObjectType` # the type build by the builder # and on `InputType` # the basic type that the `Builder` can build from. (In the previous example, `InputType` would be `double`.)

12.6. Exceptions and Messages

`Message` is the abstract class that defines the interface for all types of messages sent to loggers. `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 by template class `exception::ExceptionImpl`. This template class has a member which is an instantiation of `MessageImpl`. The exact instantiation is provided as a template parameter of `exception::ExceptionImpl`. There are two specializations of `exception::ExceptionImpl` 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-provided 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 indirectly 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 with the core library. KeyValue comes with an example bridge which can be used as a sample for bridge developers.

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

  The global `ProcessorMngr` (accessible through template `Global`) is responsible for retrieving a processor provided its name. Therefore, every `Processor` must register itself into the global `ProcessorMngr` at KeyValue's initialization.

  The suggested registration method is the following. Bridge developers copy files `bridge-example/registerProcessors.h`, `bridge-example/registerProcessors.cpp` and `bridge-example/AllProcessors.h` to their own source directory to be compiled and linked as their own source files. The first two files are left as they are but file `AllProcessors.h` should be edited (see instructions there in) to list the `Tags` that identify the various processors.

  To register all processors, simply `#include` header file `registerProcessor.h` and call function `registerProcessors()`. The suggested place to make this call is the `Bridge` constructor (see Section 13.1, “How to implement class `Bridge`”).

  Notice that KeyValues own's processors are listed in `AllProcessors.h`. Any of them can be removed from this file if one does not want make it available at runtime.

Implementing keys:
: KeyValue comes with a few generic keys but 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 four public methods which are left to the bridge developer. (See example in `bridge-example/bridge-example/Bridge.cpp`):

```
Bridge();
```

The default constructor is declared by KeyValue (not by the compiler) and, therefore, it must be implemented. One can do whatever initialization it needs for the bridge and core libraries. For instance, `Processors` registration is suggested to be launched from here by calling function `registerProcessors()` as explained above.

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

Returns the name of the core library. The result also names the function called in LibreOffice 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, for instance, by the 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

`Builder` and `Calculator` specializations (the two flavors of `Processor`) are implemented in similar ways. Firstly, let us see how to implement the latter and then cover the differences for the former.

13.2.1. Implementing a Calculator specialization

As previously said, to get the proper declaration of a `Calculator`, the helper file `keyvalue/mngt/DeclareCalculator.h` should be used.

Some specific header files must be included at the begining of the source code, notably `keyvalue/mngt/Calculator.h`. However, prior to include this file, we include the header file containing the information on the type of smart pointer used by the bridge and core libraries: (The content of this file is explained in details in Section 14, “Using custom smart pointers”.)

```
// First #include the header file containing smart pointer information:
// (This is just an example. Each bridge library #includes its onw file.)
#include "bridge-example/PtrTraits.h"

// Now #include other required header files:
// ...
#include "keyvalue/mngt/Calculator.h"
// ...
```

Now namespace `::keyvalue` is open:

```
namespace keyvalue {
```

Then the macro `TAG` is set to a word that uniquely identifies the specialization. For sake of concreteness, let us assume that this word is *Foo*:

```
#define TAG Foo
```

Now, provided the specialization is a `Command`, we `#define` the macro `COMMAND`:

```
#define COMMAND // Must be defined if, and only if, the calculator is a Command
```

Then the helper file is included:

```
#include "keyvalue/mngt/DeclareCalculator.h"
```

The steps above provide the correct declaration of the specialization (`Calculator<tag::Foo>` in this example). It is worth mentionning that `keyvalue/mngt/DeclareCalculator.h` will declare a type *Foo* in namespace `tag`.

Now we implement a few methods. The first one is

```
const char*
Calculator<tag::Foo>::getName() const;
```

which returns the name to be assigned to key *Processor* when the user wants to call this specialization.

If the macro `COMMAND` is defined, then the following method is implemented:

```
const char*
Calculator<tag::Foo>::getCommandName() const;
```

It returns an alternative name which front-ends might use when calling this specialization as a `Command`. For example, when an empty `DataSet` is given to processor *DeleteDataSets* the whole repository is cleared out. Hence, this is exactly what happens when processor *DeleteDataSets* is called as a command. For this reason, the name `"Reset repository"` seems more appropriate to appear in a menu.

The last method is

```
value::Value
Calculator<tag::Foo>::getValue(const DataSet& data) const;
```

which processes `DataSet data` and returns a `value::Value`. Recall that `value::Value` belongs to the hierarchy of types which allows for *multi-level implicit conversions*. Therefore, any type below `value::Value` in the hierarchy might be returned without further ado.

KeyValue implements a memoization system to prevent needless recalculations when the input key-value pairs in `data` have not changed since last call. To use this feature, after having retrieve all values by calling `data.getValue()`or `data.find()`, `Calculator<tag::Foo>::getValue()` must call `data.mustUpdate()` which returns `true` if the value must be recalculated or `false`, otherwise. If the value does need to be recalculated, then `Calculator<tag::Foo>::getValue()` computes the value and returns it. Otherwise, `Calculator<tag::Foo>::getValue()` must return a default constructed `value::Value`.

If appropriate, the `Calculator` can bypass this memoization system by simply not calling `data.mustUpdate()`. For instance, if a `Calculator` returns the current time, then it should rather avoid the memoization system otherwise it will always return the same time.

Finally we close namespace `::keyvalue`:

```
} // namespace keyvalue
```

13.2.2. Implementing a Builder specialization

Similarly to `Calculators`, we `#include` required header files (notably `keyvalue/mngt/Builder.h`) and open namespace `::keyvalue`:

```
// First #include the header file containing smart pointer information:
// (This is just an example. Each bridge library #includes its onw file.)
#include "bridge-example/PtrTraits.h"

// Now #include other required header files:
// ...
#include "keyvalue/mngt/Builder.h"
```

```
// ...

namespace keyvalue {
```

Macros TAG and COMMAND are used in the same way as for Calculators. For the sake of concreteness, consider the Builder specialization for logger::Logger. (See keyvalue/bridge/processor/Logger.cpp.) This is not a Command and then, we only have:

```
#define TAG Logger
```

Additionally the macro OBJECT_TYPE is #defined to be the type of object built:

```
#define OBJECT_TYPE ::keyvalue::logger::Logger
```

Notice that we provide the fully qualified name of the object type.

Then the helper file keyvalue/mngt/DeclareBuilder.h is #included:

```
#include "keyvalue/mngt/DeclareBuilder.h"
```

Methods that return the processor- and (possibly) command- names are implemented as per Calculators. In our example, the Builder is not a Command then, only

```
const char*
Builder<tag::Logger>::getName() const;
```

is needed. Otherwise, the following method would also be required:

```
const char*
Builder<tag::Logger>::getCommandName() const;
```

The method that builds a logger::Logger from the input DataSet and return a pointer to it is:

```
value::PtrTraits<::keyvalue::logger::Logger>::Type_
Builder<tag::Logger>::getObject(const DataSet& data) const;
```

Notice the returned type. This is simpler than it appears. Indeed, in this example the returned type is just an alias to ::boost::shared_ptr<::keyvalue::logger::Logger> and using this simpler form would equaly works. The more complicated form was used for the sake of generality. As we shall see (Section 14, “Using custom smart pointers”) KeyValue can work with different types of smart pointers and the class value::PtrTraits is a helper that sets the correct smart pointer depending on the ObjectType.

The memoization works for Builder<tag::Logger>::getObject() similarly as per Calculator<tag::Foo>::getValue(): If data.mustUpdate()returns true, then the method should return a pointer to the built object. Otherwise, it should return a default constructed pointer.

In addition to TAG, COMMAND and OBJECT_TYPE, the macro BUILDS_FROM might be #defined if the Builder is able to build from a basic type. The macro should be set to be the basic type that the Builder can build from. In this case another method must be implemented. It takes one input parameter (by const reference) of type BUILDS_FROM and returns the pointer to the object built. For instance, if BUILD_FROM were set to double, then we would have provided the implementation of

```
value::PtrTraits<::keyvalue::logger::Logger>::Type_
Builder<tag::Logger>::getObject(const double& data) const;
```

It is worth remembering that all macros must be defined before keyvalue/mngt/DeclareBuilder.h is #included.

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> {
  // ...
};

} // namespace key
} // namespace keyvalue
```

is the prototype for a key accepting a single `double` value which is not mapped. (Actually, the second and third parameters of `key::Traits` above match 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: mapping- and checking- methods.

13.3.1. Mapping methods

The 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.

In many circumstances, `MapType` does not need to be explicitly provided by the user since the compiler can automatically deduce it. The choice follows a simple rule: if `key::Traits`' parameter `ElementType` is a basic type, then `key::NoMap` will be selected; else if `ElementType` is an enum then `key::FlagMap` will be chosen. Otherwise, `key::ObjectMap` will be selected because the KeyValue assumes `ElementType` is a type defined by the core library and for which a `Builder` specialization is implemented.

Map types `key::NoMap` and `key::ObjectMap` do not impose any constraint on `key::Traits` derived classes. Luckily, the constraint imposed on `key::Traits` derived classes when either `key::FlagMap` or `key::PartialMap` is selected is a matter of implementing just one method with the following signature:

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

Here, `OutputType` matches parameter `ElementType` used to instantiate `key::Traits`. This method receives a `string` object and maps it to the correct value of type `OutputType` or throws a `RuntimeError` to report failure.

13.3.2. Checking methods

The checks performed on the output of `DataSet::getValue()`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) but they can be overridden when a proper check is required. To indicate invalid values, `RuntimeError` exception must be 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` is `key::ObjectMap`. In this case, the `ConverterType` is instantiated for `value::PtrTraits<ElementType>::Type_`.

The `ConverterType` also defines the output type, `OutputType_`, returned by `DataSet::getValue()`.

`key::StdSingle<ElementType>`
: This is the default choice and can be omitted when `MapType` is so.

For this choice `OutputType_` matches `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 the one of `key::StdSingle` seen above.

There is a method to check the vector size. It is 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 avoiding to process the third element onwards. This method's 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 under construction.

`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 the one of `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 the one of `key::StdVector` but, naturally, here `ConverterType<ElementType>` is `key::StdMatrix<ElementType>`.

14. Using custom smart pointers

Each builder gives to KeyValue a pointer to an object that is then, stored in KeyValue's repository. Normally, the memory occupied by any of these object is allocated on the heap (through operator `new`) and to prevent memory leaks, smart pointers must be used. The most popular smart pointer types are, probably, `::boost::shared_ptr` and `::boost::intrusive_ptr` and KeyValue provides good support for some of them. Some projetcs, however, have a genuine need for other types of smart pointers and KeyValue is flexible with respect to this point.

Obviously, KeyValue needs to know about the smart pointer type used by the bridge and core libraries. The next sections explain how to provide the required information.

14.1. The pointer traits header file

This file provides the information on the smart pointer used by bridge and core libraries. The name of this file is flexible but, for sake of concreteness, in this manual it will be called `PtrTraits.h`.

`PtrTraits.h` must be included at the very beginning of all builder and calculator source files. (For instance, every processor implemented by the bridge-example library `#includes` the file `bridge-example/PtrTraits.h`.)

In the sequel, we shall see the four tasks that `PtrTraits.h` must acomplish. Section 14.1.5, “Examples of pointer traits header files” presents examples of pointer traits files that come with KeyValue and can be used by bridge developers as samples for writing their onw.

14.1.1. Implementing the template struct `value::PtrTraits`

This struct depends on a parameter `ObjectType` and provides a `typedef`, namely `Type_`, to the smart pointer class that points to objects of type `ObjectType`.

For instance, in `bridge-example/PtrTraits.h` we have:

```
namespace keyvalue {
namespace value {

template <typename ObjectType>
struct PtrTraits {
  typedef ::boost::shared_ptr<ObjectType> Type_;
};

} // namespace value
} // namespace keyvalue
```

This tells KeyValue that, as far as the bridge-example library is concerned, a pointer to `ObjectType` is a `::boost::shared_ptr<ObjectType>`.

14.1.2. Implementing the specialization of `value::PtrTraits` for void

The general implementation of `value::PtrTraits` defines the smart pointer for each specific type of object. However, KeyValue's repository is a container for uniform storage, that is, stored pointers must have the same type. This situation is analogous to the classic example of polymorphism found in many C++ text books where a `::std::vector<Shape*>` stores pointers to several types of shapes like `Square`, `Circle`, etc.

The specialization of `value::PtrTraits` for `void` tells KeyValue what is the type of pointer that the repository stores.

For instance, in `bridge-example/PtrTraits.h` we have:

```
namespace keyvalue {
namespace value {

template <>
struct PtrTraits<void> {
  typedef ::boost::shared_ptr< ::core::Polygon> Type_;
};

} // namespace value
} // namespace keyvalue
```

This means that KeyValue uniformly stores `::boost::shared_ptr<::core::Polygon>`s. Hence, when a `::boost::shared_ptr<::core::Triangle>` is given to KeyValue, it is cast to a `::boost::shared_ptr<::core::Polygon>` before being stored. Later, when the bridge-example library asks for this object back, KeyValue does the oposite cast.

14.1.3. Implementing the function `dynamic_pointer_cast`

As we have seen, KeyValue casts pointers to specific types to pointers to the generic type and vice versa. The cast *specific-to-generic* is performed by a constructor and this point is explained in more details in Section 14.3, “Constraints on custom smart pointers”. The *generic-to-specific* cast is performed by a template function which, essentially, has the following signature:

```
template <typename ObjectType>
value::PtrTraits<ObjectType>::Type_
dynamic_pointer_cast(const value::PtrTraits<void>::Type_& genericPointer);
```

The implementation must obey the semantics of the C++ built-in cast operator `dynamic_cast`. Specifically, if `genericPointer` does point to an `ObjectType`, then `dynamic_pointer_cast` returns a `value::PtrTraits<ObjectType>::Type_` pointing to the same object. Otherwise, a *NULL* `value::PtrTraits<ObjectType>::Type_` is returned.

This function is named after `::boost::dynamic_pointer_cast` which satisfies the requirement describe above. Hence, for some boost smart pointers, `::boost::dynamic_pointer_cast` is exactly what is needed. More precisely, `PtrTraits.h` should not implement `dynamic_pointer_cast` when the bridge library uses either `::boost::shared_ptr` or `::boost::intrusive_ptr`. In this case, KeyValue calls `::boost::dynamic_pointer_cast`.

For example, `bridge-example/PtrTraits.h` sets the specific pointer to be `::boost::shared_ptr<ObjectType>` and the generic pointer to be `::boost::shared_ptr<::core::Polygon>`. Hence, it does not implement `dynamic_pointer_cast`. Notice that, in this case, the signature above reads

```
template <typename ObjectType>
::boost::shared_ptr<ObjectType>
dynamic_pointer_cast(const ::boost::shared_ptr<::core::Polygon>& genericPointer);
```

and matches the one in namespace `::boost` whose implementation is `#included` into `bridge-example/PtrTraits.h` through `boost/shared_ptr.hpp`.

If you do need to implement `dynamic_pointer_cast`, then you can place it in one of the following namespaces:

- The global namespace.

  Do not do this! It does work but is considered bad practice (namespace pollution).

- Namespace `value`.

  The call to `dynamic_pointer_cast` comes from this namespace and then, if this function is there, then the compiler will find it.

- The namespace that contains the smart pointer.

  For instance, when using `::boost::shared_ptr`, the function can be (and it is) in namespace `::boost`. Analogously, if your curstom smart pointer belongs to namespace `::foo::bar`, then `dynamic_pointer_cast` can be placed in `::foo::bar` as well.

14.1.4. Defining the macro KEYVALUE_PTR_TRAITS_FILE

Briefly, this macro can be uniformly set in this way:

```
#ifndef KEYVALUE_PTR_TRAITS_FILE
#define KEYVALUE_PTR_TRAITS_FILE __FILE__
#endif
```

Section 14.2, “The macro KEYVALUE_PTR_TRAITS_FILE” provides detailed information about this macro.

14.1.5. Examples of pointer traits header files

Given the popularity of boost libraries, KeyValue provides three examples of pointer traits files based on some boost smart pointers. Bridge developers can use any of them almost out of the box:

`keyvalue/value/ptr-traits/shared_ptr.h`
: This implementation is based on `::boost::shared_ptr` and can be used when all core library types managed by KeyValue derive from the same base polymorphic class. The original file must be adapted to reflect the correct base class.

  Notice that `bridge-example/PtrTraits.h` is basically a copy of this file with the base polymorphic class set to `::boost::shared_ptr<::core::Polygon>`.

`keyvalue/value/ptr-traits/intrusive_ptr.h`
: Similarly to the previous file but based on `::boost::intrusive_ptr`, you can use this file when you have a common polymorphic base class for all core library types managed by KeyValue. Here again, you must edit the file to use the correct base class. Other constraints on the base class, regarding the referencing counting, are explained in boost's documentation and are outside the scope of this manual.

`keyvalue/value/ptr-traits/AnyPtr.h`
: This implemenation is based on `util::AnyPtr`. Opposite to the previous implementations, this one does not require a commom polymorphic base class and this file can be used as it is with no need for adaptations.

  The implementation of `util::AnyPtr` is very similar to the one presented in [4].

None of the files above (or their derivatives) needs to provide the implementation of `dynamic_pointer_cast`. Indeed, for the first two the implementation is provided by boost and for the third one the implementation is provided by KeyValue.

14.2. The macro `KEYVALUE_PTR_TRAITS_FILE`

The macro `KEYVALUE_PTR_TRAITS_FILE` must be set to the name of the pointer traits header file. Unfortunately, there are two places where this macro is set and both definitions must agree, otherwise weird errors can occur.

The configuration file `config/config.mak`:
: You should set `KEYVALUE_PTR_TRAITS_FILE` to either an absolute or a relative (with respect to `keyvalue` directory) path.

  For instance, to use `bridge-example/bridge-example/PtrTraits.h` and assuming that KeyValue was unpacked in `/home/cassio/keyvalue-0.3`, you set

  ```
  KEYVALUE_PTR_TRAITS_FILE := /home/cassio/keyvalue-0.3/bridge-example/bridge-exampl
  ```

  Analogously, if KeyValue was unpacked in `C:\Users\cassio\Documents\keyvalue-0.3`, then use

  ```
  KEYVALUE_PTR_TRAITS_FILE := C:/Users/cassio/Documents/keyvalue-0.3/bridge-example/
  ```

  Alternatively, in both cases above, you can also use

  ```
  KEYVALUE_PTR_TRAITS_FILE := ../bridge-example/bridge-example/PtrTraits.h
  ```

The pointer traits header file:
: This is the very same file that `KEYVALUE_PTR_TRAITS_FILE` points to. Therefore, the simplest way of defining this macro is using the standard predefined macro `__FILE__`:

  ```
  #ifndef KEYVALUE_PTR_TRAITS_FILE
  #define KEYVALUE_PTR_TRAITS_FILE __FILE__
  #endif
  ```

14.3. Constraints on custom smart pointers

KeyValue has some expectations on the smart pointers received from builders. This section covers the conditions that smart pointer implentations must verify.

[4] Neri, C., *Twisting the RTTI System for Safe Dynamic Casts of `void*` in C++*, Dr.Dobbs, April 2011 (http://drdobbs.com/cpp/229401004).

Most smart pointers are implemented as template classes parametrised on the type of object pointed to. Although this is **not** a KeyValue's requirement, to simplify the presentation, we will assume this is the case and we shall call this template `MinimalPointer`.

The interface that KeyValue requires `MinimalPointer` to implement is given by the skeleton below.

```
template <typename ObjectType>
class MinimalPointer {

public:

  MinimalPointer(const MinimalPtr& orig);

  ~MinimalPointer();

};
```

As we can see, `MinimalPointer` must have a public copy-constructor and a public destructor (virtual or not). We emphasize that this interface is the **minimum** requirement and smart pointers will certainly extend it. Having said that, we notice the omission of constructors (apart from the copy-constructor). Surely, there is no class without a constructor but, because KeyValue does not create these pointers (it only copies those created by the bridge), it does not require any other particular constructor to be implemented. Additionally, KeyValue does not dereference the pointer and, thus, does not require the implementation of `operator ->()`. The same holds for other methods usually implemented by smart pointers.

Recall that KeyValue receives pointers to different types of objects but, for uniform storage, casts them to a unique smart pointer type. For the sake of this presentation, we will assume that this smart pointer is a specialization of `MinimalPointer` when `ObjectType` is `void`. Its minimal interface is given below.

```
template <>
class MinimalPointer<void> {

public:

  MinimalPointer();

  ~MinimalPointer();

  MinimalPointer&
  operator =(const MinimalPointer& orig);

  bool
  operator ==(const MinimalPointer& rhs) const;

  template <typename ObjectType>
  MinimalPointer(const MinimalPtr<ObjectType>& orig);

};
```

The following public methods must be implemented: the default constructor, the destructor (`virtual` or not), the assignment operator, a comparison operator and a template constructor taking a generic smart pointer as argument. The first three methods do not need any further comments. Now, we shall consider how KeyValue uses the last two.

Usually smart pointers implement `operator bool()` returning `false`, if the pointer is *NULL*, or `true`, if it is not. KeyValue does not require that. Instead, it considers a pointer to be *NULL* if, and only if, the result of comparing the pointer (through `operator ==()`) with a default-constructed one gives `true`.

When KeyValue needs converting a `MinimalPointer<ObjectType>` into a `MinimalPointer<void>` it uses the template constructor above which can be `explicit` or not.

15. Linking with KeyValue

The instructions in `config/config.mak` advise you to leave variables `FELIBS_debug` and `FELIBS_release` as they are in order to link KeyValue with the examples of core and bridge libraries. However, you must change these variables to point to your own core and bridge libraries' paths in order to link KeyValue with them. You can use either absolute or relative paths. Relative paths are taken from `excel-addin` and/or `openoffice-addin` directories.

If you are a Microsoft Visual Studio user, then you must use multi-threaded runtime libraries to compile your core and bridge libraries. More precisely you have two options depending on your build system:

Microsoft Visual Studio build system
: Follow the instructions below if you use Microsoft Visual Studio build system to build your core and bridge libraries. (This is the default method choosen when you set up your libraries using Microsoft Visual Studio's IDE.)

1. Open Microsoft Visual Studio and your solution file.
2. Open the solution explorer (**Ctrl+Alt+L**).
3. Right-click on you core library project.
4. Select: Properties -> Configuration Properties -> C/C++ -> Code Generation.
5. On the right panel on the properties page, select the correct Runtime Library depending on configuration as below:

   a. For Debug configuration choose "Multi-threaded debug (/MTd)";

   b. For Release configuration choose "Multi-threaded (/MT)".

   Remark: (a) and (b) above do **not** say to choose the "DLL" libraries.

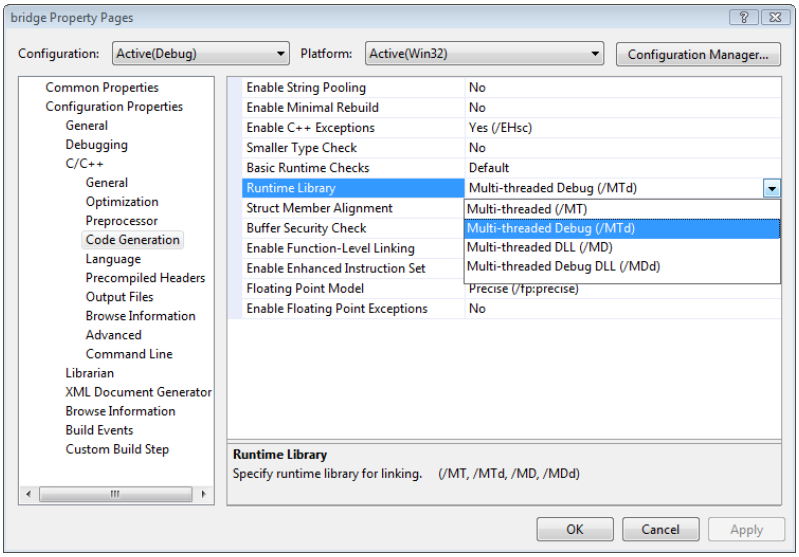

Figure 20. Choosing the right runtime libraries.

Repeat steps (3) - (5) for your bridge library and for any other library you want to link with KeyValue.

Another build system
: Follow the instruction below if you build your core and bridge libraries using another build system (e.g., makefiles, bjam, etc).

1. Make sure you pass to MSVC compiler (cl.exe) the apropriate option regarding the runtime libraries:

   a. Use `/MTd` for debug build.

   b. Use `/MT` for release build.

If for some reason you are not happy to compile your libraries using the options above, then you can change KeyValues' compilation options. However, the LibreOffice add-in will not build anymore; only the Excel add-in will. To change KeyValue's compiling options open the file `config/windows-msvc.mak` in any text editor and edit the lines below

```
debug   : OBJ_FLAGS += -D_DEBUG -Od -Gm -RTC1 -MTd -ZI
release : OBJ_FLAGS += -DNDEBUG -O2 -Oi -GL -FD -MT -Gy -Zi
```

replacing the `-MTd` and `-MT` according to your preferences. You might need to rebuild KeyValue (clean and build again).

16. The Excel add-in

The Excel add-in has two particular features describred in the sequel.

16.1. The help file

A help file in compressed HTML format can be associated to the Excel add-in. This file must be named `manual.chm` and be located in the directory containing the add-in. Then under the Excel function wizard for KEYVALUE function (or whatever is the name provided by the bridge), one can click on "Help on this function" to open `manual.chm`.

The file will be open by the program associated with extendion `.chm` at the position mapped to ID number 1000. For instance, the `.chm` of this user manual is called `manual.chm` and maps the ID number 1000 to Section 5, “The KEYVALUE function”.

Instructions on how to create `.chm` files and how to map ID numbers to anchors is outside of the scope of this document.

16.2. The menu of commands

As explained in Section 7.1.1, “Commands”, Excel add-in provides special support for commands. Under the add-in menu on the menu bar, it presents a menu named after the core library from which one can call any command. The result of the command can be seen on the global logger.