Using container objects in datatools

(Software/DatatoolsContainerTutorial – version 0.1)

F. Mauger <`mauger@lpccaen.in2p3.fr`>

2011-11-29

Abstract

This note explains how to use the container classes provided by the `datatools` program library.

Contents

1 Introduction

The `datatools` program library provides among other components a couple of container classes, namely `datatools::properties`, `datatools::multi_properties` and `datatools::things` classes.

Subversion repository:
`https://nemo.lpc-caen.in2p3.fr/svn/datatools/`
in the `doc/Memos/DatatoolsContainerTutorial` subdirectory
DocDB reference: `NemoDocDB-doc-1997`
References: see also *Serialization with the* ***`datatools`*** *program library* (`NemoDocDB-doc-2003`)

2 The `datatools::utils::properties` class

2.1 Introduction

The `datatools::utils::properties` class is designed to be an useful container of arbitrary *properties*. In the `datatools`' terminology, a *property* is a variable of a simple type which has :

- a *name* which can be used to uniquely identify and thus access to it,
- a *value* of a certain *type* which is supposed to be the material used by an user or a client application.

As an example, let's build the virtual representation of a simple geometry 3D-object, namely a cube, using *properties*. We may choose the following list of *properties* :

Name	Value	Type
`"dimension"`	28.7 cm	real
`"colour"`	`"blue"`	string
`"material"`	`"copper"`	string
`"price"`	2.36 €	real
`"available"`	true	boolean
`"nb_in_stock"`	65	integer
`"manufacturer"`	`"The ACME International Company"`	string
`"reference_number"`	`"234/12.456"`	string

As it can be seen in the example above, this list of *properties* makes possible to figure out non-ambiguously the main characteristics of an object. We should have writen, its main *static* characteristics, because the table above does not contain informations about the *behavior* and *dynamic* characteristics of this cube.

2.2 Header file and instantiation

In order to use `datatools::utils::properties` objects, one must use the following include directive:

```
#include <datatools/utils/properties.h>
```

The declaration of a `datatools::utils::properties` instance can be simply done with:

```
datatools::utils::properties my_properties_container;
```

Note the use of the nested `datatools::utils` namespace. Alternatively one can use :

```
using namespace datatools::utils;
properties my_properties_container;
```

or:

```
namespace du = datatools::utils;
du::properties my_properties_container;
```

2.3 Design

2.3.1 Naming scheme

The `datatools::utils::properties` class is implemented through a `std::map` container as a dictionnary (or lookup table) of *records*. A record contains all informations to uniquely identify a given property : name, type, value and a few other attributes. On the point of view of the user, implementation details are not important.

First of all, when one wants to store a property within a `datatools::utils::properties` container object, we must choose it a *name* (or *key*). This name must be unique in this container, so it must not be already used by another property record.

The name must fulfill some rules:

- it must not be the empty string (`""`),
- it must contains only digits (0...9), alphabetical characters (`'a'` to `'z'` and `'A'` to `'Z'`) and the underscore (`'_'`) or dot (`'.'`) characters,
- it must not start with a digit,
- it must not end with a dot.

Table 1 shows some valid and invalid choices for names to be used as property keys in a `datatools::utils::properties` object.

Key	Validity
`"a"`	yes
`"debug_level"`	yes
`"_hello_"`	yes
`"FirstName"`	yes
`"__status"`	yes
`"logging.filename"`	yes
`"b."`	no
`"${DOC}"`	no
`"007"`	no

Table 1: Example of names (keys) supported or not supported by the `datatools::utils::properties` class.

This default behavior can be changed by attaching another *key validator* object to the container. However, this functionnality is out of the scope of this tutorial.

2.3.2 Supported types

Now we have chosen the name (key) for a property, we must indicate of which type it is. Only four types are supported by the `datatools::utils::properties` class, both in a *scalar* version (one single value) or in a *array* version (implemented as a `std::vector`). These types are given in table 2.

Label	Implementation	Range
boolean (flags)	`bool`	false or true (0 or 1)
integer	32 bits signed integers	from -2^{31} to $+2^{31}-1$
real	`double`	IEEE-754 64 bit encoded double precision floating point numbers
string	`std::string`	any character string not containing the double quote character(`'"'`)

Table 2: Property types supported by the `datatools::utils::properties` class.

2.3.3 Additional property's traits

When one stores a new property in a *properties* container, it is possible to provide an arbitrary string that describes it.

More it is also possible to mark it as *non-mutable*. It means that it will not be possible to further modify its value. However, the property still can be erased.

It is also possible to use *private* properties. Conventionnaly, private properties have a name/key starting with two underscores (example: `"__secret)"`. The *private* trait can trigger different I/O behaviors (see below).

2.3.4 Vector properties

As mentionned above, a property can store a single (scalar) typed value as well as an array of values of the same type. The size of such an array can be zero or whatever number of elements. It is only limited by the available memory and possible limitations of the implementation (here we use `std::vector` templatized containers).

2.3.5 Meta informations

Aside some internal data structures used to implement the dictionnary of property records, a `datatools::utils::properties` object has a *debug* flag which is not supposed to be activated in some production program and a *description* string which can be used to store some arbitrary text formated meta-information.

2.3.6 Constructors

The `datatools::utils::properties` class provides a few useful constructors.

The default constructor initializes an empty container of properties and fulfills most needs in realistic use cases. Sample 1 shows the syntax used to declare a properties container object.

2.3.7 Interface methods

There are many methods available to manipulate `datatools::utils::properties` container objects.
Non mutable methods :

```
#include <datatools::utils::properties.h>
...
{
  ...
  datatools::utils::properties config;
  ...
  datatools::utils::properties documented_config ("my documented configuration");
  ...
}
```

Sample 1: The declaration of a default empty *properties* object and a documented (with an embedded description) empty *properties* object.

- `get_description` returns the embedded description character string (if any),
- `keys` builds a list of the keys of all objects stored in the container,
- `is_locked` informs if the container is locked (its contents cannot be changed),
- `size` returns the number of properties stored in the container,
- `has_key` informs if the container containes a property with a given key/name,
- `is_boolean`, `is_integer`, `is_real`, `is_string` inform if a stored property is of some given type,
- `is_scalar` informs if a stored property is scalar (a single value is stored) or not,
- `is_vector` informs if a stored property is vector (an array of values of the same type is stored) or not,
- `has_flag` informs if a stored boolean property exists and has the "`true`" value,
- `fetch_boolean` returns the value of an existing boolean property (flag) stored in the container.
- `fetch_integer` returns the value of an existing integer property stored in the container.
- `fetch_real` returns the value of an existing real property stored in the container.
- `fetch_string` returns the value of an existing string property stored in the container.
- various `fetch` and `fetch_XXX_YYY` methods return the value of an existing property of some given type `XXX` (`boolean`, `integer`, `real` or `string`) and given size `YYY` (`scalar`, `vector`) stored in the container. In case of a vector property, the index of value in the internal array can be passed.
- `tree_dump` prints the content of the container in an ASCII readable format.

A large set of mutable methods are available :

- store_flag and set_flag adds a new boolean property in the container with a given key (the key must not already exist), the value is set at "true" by default
- store_boolean adds a new boolean property in the container with a given key (the key must not already exist), the value is set on user choice,
- store_integer adds a new integer property in the container with a given key (the key must not already exist), the value is set on user choice,
- store_real adds a new real property in the container with a given key (the key must not already exist), the value is set on user choice,
- store_string adds a new string property in the container with a given key (the key must not already exist), the value is set on user choice,
- various overloaded store methods add a new property of a given type (scalar, vector) in the container with a given key (the key must not already exist), the value is set at user choice,
- various change_XXX methods modify the value of an existing property of type XXX (boolean, integer, real or string). In case of a vector property, the index of the value to be changed in the internal array can be passed.
- various overloaded change methods modify the value of an existing property of a given type and size (scalar, vector) in the container. The value is set at user choice. In case of a vector property, the index of the value to be changed in the internal array can be passed.
- update_flag adds a new boolean true value using a given key or update it at its true value is it already exists.
- update_XXX methods update the value or add it if it does not exist yet. The value is set at user choice.
- various overloaded update methods behave like the one described above.
- erase erase the property with a given key
- erase_all erase all stored properties,
- erase_all_starting_with erase all stored properties with a key that starts with a given prefix,
- erase_all_not_starting_with erase all stored properties with a key that does not start with a given prefix, ,
- erase_all_ending_with erase all stored properties with a key that ends with a given prefix,
- erase_all_not_ending_with erase all stored properties with a key that does not end with a given prefix,

- `lock` locks the container for all mutable methods but the `unlock` method,
- `unlock` unlocks the container if it was locked previously.
- `reset` and `clear` completely blank the contents of the container (its description and its dictionnary of properties).

2.3.8 Exporting features

The `datatools::utils::properties` class is copyable. It means that it is possible to use the affectation operator (`operator=`) to affect the value of one container property object to another one. A copy constructor is also available. The program source 1 shows how two containers can be initialized by copy of an original one. The output printed is shown on sample 2.

```
#include <iostream>

#include <datatools/utils/properties.h>

int main (void)
{
  datatools::utils::properties config ("A test properties container");
  config.store_flag ("test");      // store a boolean/flag property (true)
  config.store ("name", "Monty");  // store a string property
  config.tree_dump (std::cout, "Original container :"); // ASCII print
  std::cout << endl;

  datatools::utils::properties  config2;
  config2 = config;
  config2.tree_dump (std::cout, "Properties container copied through affectation :");
  std::cout << endl;

  datatools::utils::properties config3 (config);
  config3.tree_dump (std::cout, "Properties container copied at construction :");

  return 0;
}
```

Program 1: A program that uses the assignment operator and copy constructor for `datatools::utils::properties` objects.

These plain copying functionalities are obviously extremely useful and are implemented by default in the library. However, it is sometimes desirable to export only a subset of properties from an original properties container and *install* these subset inside another properties container : this is called *exporting properties*. It realize some kind of partial copy of some original container (source) to another one (target).

Several export methods are implemented, all need to be passed a reference to the target container and some string that is used as a prefix for properties' keys:

- `export_starting_with` install, in the target container, a copy of each property, of which the key starts with a given prefix,

```
Original container :
|-- Description  : A test properties container
|-- Name : "name"
|   |-- Type  : string (scalar)
|   `-- Value : "Monty"
`-- Name : "test"
    |-- Type  : boolean (scalar)
    `-- Value : 1

Properties container copied through affectation :
|-- Description  : A test properties container
|-- Name : "name"
|   |-- Type  : string (scalar)
|   `-- Value : "Monty"
`-- Name : "test"
    |-- Type  : boolean (scalar)
    `-- Value : 1

Properties container copied at construction :
|-- Description  : A test properties container
|-- Name : "name"
|   |-- Type  : string (scalar)
|   `-- Value : "Monty"
`-- Name : "test"
    |-- Type  : boolean (scalar)
    `-- Value : 1
```

Sample 2: The output of the program 1.

- `export_not_starting_with` install, in the target container, a copy of each property, of which the key does not start with a given prefix,
- `export_and_rename_starting_with` install, in the target container, a copy of each property, of which the key starts with a given prefix; copied properties are renamed with a new prefix,
- `export_if` (template) install, in the target container, a copy of each property, of which the key fulfills a given predicate,
- `export_not_if` (template) install, in the target container, a copy of each property, of which the key does not fulfill a given predicate.

The program source 2 shows how properties containers can be initialized by exporting original properties from an original one. The output printed is shown on sample 3.

2.3.9 Serialization and I/O features

The `datatools::utils::properties` container class is equipped with some I/O functionnalities. There are two techniques that can be used to *serialize* a properties container object :

- ASCII formatting in standard I/O streams: this technique makes possible to store and load properties objects from simple text files. It uses an ASCII human readable format. The full dictionnary of properties is thus recorder in an non-ambiguous way. Such approach is extremely useful to implement configuration files.
- High level serialization using the Boost/Serialization library. Some dedicated template methods are implemented to fulfill the Boost/Serialization API. Boost text, XML and binary I/O archives are supported. The `datatools::utils::properties` class itself fulfill some special serialization interface that makes it usable with some advanced serialization mechanism[1]. The description of this feature is out of the scope of this document.

Let's concentrate on the ASCII I/O functionnalities:
the `datatools::utils::properties::config` class is responsible to write/read to/from ASCII formated standard streams. Utility static wrapper methods are provided to perform such I/O operations with files. By default only *non-private* properties are saved or loaded.

The program source 3 shows how to use this class to dump the ASCII format on the terminal or a file. The output printed is shown on sample 5, the contents of the saved file is shown on sample 5.

Of course, it is perfectly possible to create a configuration file that respect this format using your favorite text editor or some external script. It is then possible to ask a given program to initialize a properties container using this input file. The program source 4 shows such an example with a hand-edited configuration file (sample 6). The output of the program is shown in sample 7.

[1] The `datatools::serialization::i_serializable` class interface

```
#include <iostream>

#include <datatools/utils/properties.h>

// A predicate on 'string' that searches for a given substring :
class contains_string_predicate : public std::unary_function<const string &, bool>
{
  string _token_; // the substring to be found in the string argument
public:
  contains_string_predicate (const string & token_) : _token_ (token_)
  {
  }
  bool operator () (const string & key_)const
  {
    return key_.find (_token_) != key_.npos;
  }
};

int main (void)
{
  datatools::utils::properties config ("A test properties container");
  config.store_flag ("test");
  config.store_flag ("program_name", "prog");
  config.store ("debug.level", 1);
  config.store ("debug.filename", "debug.log");
  config.tree_dump (std::cout, "Original container :");
  std::cout << endl;

  datatools::utils::properties debug_config;
  config.export_starting_with (debug_config, "debug.");
  debug_config.tree_dump (std::cout,
    "Exported properties container with the 'debug.' key prefix :");
  std::cout << endl;

  datatools::utils::properties non_debug_config;
  config.export_not_starting_with (non_debug_config, "debug.");
  non_debug_config.tree_dump (std::cout,
    "Exported properties container without the 'debug.' key prefix :");
  std::cout << endl;

  datatools::utils::properties with_name_config;
  config.export_if (with_name_config, contains_string_predicate ("name"));
  with_name_config.tree_dump (std::cout,
    "Exported properties container with the 'name' string in key :");
  std::cout << endl;

  datatools::utils::properties without_name_config;
  config.export_not_if (without_name_config, contains_string_predicate ("name"));
  without_name_config.tree_dump (std::cout,
    "Exported properties container without the 'name' string in key :");
  std::cout << endl;

  return 0;
}
```

Program 2: A program that uses `datatools::utils::properties` class' exporting methods.

```
Original container :
|-- Description  : A test properties container
|-- Name : "debug.filename"
|   |-- Type  : string (scalar)
|   `-- Value : "debug.log"
|-- Name : "debug.level"
|   |-- Type  : integer (scalar)
|   `-- Value : 1
|-- Name : "program_name"
|   |-- Description  : prog
|   |-- Type  : boolean (scalar)
|   `-- Value : 1
`-- Name : "test"
    |-- Type  : boolean (scalar)
    `-- Value : 1

Exported properties container with the 'debug.' key prefix :
|-- Name : "debug.filename"
|   |-- Type  : string (scalar)
|   `-- Value : "debug.log"
`-- Name : "debug.level"
    |-- Type  : integer (scalar)
    `-- Value : 1

Exported properties container without the 'debug.' key prefix :
|-- Name : "program_name"
|   |-- Description  : prog
|   |-- Type  : boolean (scalar)
|   `-- Value : 1
`-- Name : "test"
    |-- Type  : boolean (scalar)
    `-- Value : 1

Exported properties container with the 'name' string in key :
|-- Name : "debug.filename"
|   |-- Type  : string (scalar)
|   `-- Value : "debug.log"
`-- Name : "program_name"
    |-- Description  : prog
    |-- Type  : boolean (scalar)
    `-- Value : 1

Exported properties container without the 'name' string in key :
|-- Name : "debug.level"
|   |-- Type  : integer (scalar)
|   `-- Value : 1
`-- Name : "test"
    |-- Type  : boolean (scalar)
    `-- Value : 1
```

Sample 3: The output of the program 2.

```
#include <iostream>

#include <datatools/utils/properties.h>

int main (void)
{
  {
    datatools::utils::properties config ("A test properties container");
    config.store_flag ("test", "A boolean value");
    config.store ("debug.level", 1, "The debug level");
    config.store ("pi", 3.14159, "The approximated value of pi");
    config.store ("name", "Monty");

    // ASCII formatting in the terminal :
    datatools::utils::properties::config writer;
    std::cout << "ASCII formatted printing of the properties container : "<< endl;
    std::cout << "-----------------------------------------------------" << endl;
    writer.write (std::cout, config);
    std::cout << "-----------------------------------------------------" << endl;
    std::cout << endl;

    // save in an ASCII configuration file :
    datatools::utils::properties::write_config ("properties_4.conf",
                                                config);
  }

  {
    datatools::utils::properties restored_config;
    // read (restore) from an ASCII configuration file :
    datatools::utils::properties::read_config ("properties_4.conf",
                                               restored_config);
    restored_config.tree_dump (std::cout,
                               "The properties container restored from the file :");
  }

  return 0;
}
```

Program 3: Save and load functionnalities with `datatools::utils::properties` objects. Note that the description of the container itself is printed through a meta-comment (line starting with the "`#@config`" prefix). Also, some properties have been originaly stored with an associated transcient description string. This information is also saved for convenience in the ASCII file using special meta-comments (lines starting with "`#@description`" prefix).

```
ASCII formatted printing of the properties container :
---------------------------------------------------
# List of properties (datatools::utils::properties):

#@config A test properties container

#@description The debug level
debug.level : integer = 1

name : string = "Monty"

#@description The approximated value of pi
pi : real = 3.14159

#@description A boolean value
test : boolean = 1

# End of list of properties.
---------------------------------------------------

The properties container restored from the file :
|-- Description  : A test properties container
|-- Name : "debug.level"
|   |-- Description  : The debug level
|   |-- Type  : integer (scalar)
|   `-- Value : 1
|-- Name : "name"
|   |-- Type  : string (scalar)
|   `-- Value : "Monty"
|-- Name : "pi"
|   |-- Description  : The approximated value of pi
|   |-- Type  : real (scalar)
|   `-- Value : 3.14159
`-- Name : "test"
    |-- Description  : A boolean value
    |-- Type  : boolean (scalar)
    `-- Value : 1
```

Sample 4: The output of the program 3.

```
# List of properties (datatools::utils::properties):

#@config A test properties container

#@description The debug level
debug.level : integer = 1

name : string = "Monty"

#@description The approximated value of pi
pi : real = 3.14159

#@description A boolean value
test : boolean = 1

# End of list of properties.
```

Sample 5: The contents of the file generated by the program 3. Note the string property `"name"` has no description.

```
# List of properties (datatools::utils::properties):

# Some arbitrary comments (not parsed) :
#   Author: F.Mauger
#   Date:   2011-11-27
#   Place:  Paris-Annecy TGV
# Meta-comments start with '#@'

#@config A set of configuration parameters for an application

#@description The name of the application
app.name : string = "Monty"

#@description The system on which the application must be used
app.system : string = "Linux"

#@description The major/minor/patch level version numbers of the application
app.version : integer[3] = 3 1 4 # an array of 3 integers

#@description The debug level
app.debug.level : integer = 1

#@description A test flag
test : boolean = 0

#@description The value of pi used by the application
app.math.pi : real = 3.14159

#@description The value of pi used by the application
app.io.output_file : string = "${TMPDIR}/data/app.out"

#@description The value of pi used by the application
app.log.level : integer = 4

#@description The list of email addresses to send alarm messages
app.alarm.emails : string[2] = \
  "staff@acme.com" \
  "me@physics.lab.eu" # an array of 2 strings

# End of list of properties.
```

Sample 6: A configuration file created *by hand* to be used by the program 4. Note that if a backslash, immediately followed by a newline character, is added at the end of a given line, the parser assumes the line continues on the next line in the file.

```
#include <iostream>

#include <datatools/utils/properties.h>

int main (void)
{
  datatools::utils::properties config;

  // read the container from an ASCII configuration file :
  datatools::utils::properties::read_config ("properties_5.conf",
                                             config);

  config.tree_dump (std::cout,
                      "The transcient properties container built from a file :");
  return 0;
}
```

.

Program 4: Initialization of a `datatools::utils::properties` container through the reading of the ASCII configuration file (sample 6)

```
The transcient properties container built from a file :
|-- Description  : A set of configuration parameters for an application
|-- Name : "app.alarm.emails"
|   |-- Description  : The list of email addresses to send alarm messages
|   |-- Type  : string[2] (vector)
|   |-- Value[0] : "staff@acme.com"
|   `-- Value[1] : "me@physics.lab.eu"
|-- Name : "app.debug.level"
|   |-- Description  : The debug level
|   |-- Type  : integer (scalar)
|   `-- Value : 1
|-- Name : "app.io.output_file"
|   |-- Description  : The value of pi used by the application
|   |-- Type  : string (scalar)
|   `-- Value : "${TMPDIR}/data/app.out"
|-- Name : "app.log.level"
|   |-- Description  : The value of pi used by the application
|   |-- Type  : integer (scalar)
|   `-- Value : 4
|-- Name : "app.math.pi"
|   |-- Description  : The value of pi used by the application
|   |-- Type  : real (scalar)
|   `-- Value : 3.14159
|-- Name : "app.name"
|   |-- Description  : The name of the application
|   |-- Type  : string (scalar)
|   `-- Value : "Monty"
|-- Name : "app.system"
|   |-- Description  : The system on which the application must be used
|   |-- Type  : string (scalar)
|   `-- Value : "Linux"
|-- Name : "app.version"
|   |-- Description  : The major/minor/patch level version numbers of the application
|   |-- Type  : integer[3] (vector)
|   |-- Value[0] : 3
|   |-- Value[1] : 1
|   `-- Value[2] : 4
`-- Name : "test"
    |-- Description  : A test flag
    |-- Type  : boolean (scalar)
    `-- Value : 0
```

Sample 7: The output of the program 4.

2.4 Examples

2.4.1 Store and fetch boolean properties

Sample code for adding boolean properties and/or flags in a properties container (by convention a flag is an existing boolean property with value *true*) :

```
datatools::utils::properties config;

// Default value is set at 'true' for this boolean property :
config.store_flag ("debug", "Activation of debugging feature");

// Set explicitely the values of some boolean properties :
config.store ("test", false);
config.store_boolean ("another", true);
```

Sample code for checking the existence of some boolean properties (flags) in a properties container and fetch the associated value :

```
if (config.is_boolean ("test"))
  {
    std::cout << "Boolean 'test' exists !" << std::endl;
  }

if (config.has_flag ("debug"))
  {
    std::cout << "Flag 'debug' is set !" << std::endl;
  }

if (config.has_flag ("another"))
  {
    std::cout << "Flag 'another' is also set !" << std::endl;
  }

if (! config.has_flag ("test"))
  {
    std::cout << "Flag 'test' is not set !" << std::endl;
  }

if (config.has_key ("debug"))
  {
    bool debug = config.fetch_boolean ("debug");
    std::cout << "Fetched boolean value for 'debug' is : " << debug << std::endl;
  }
```

Sample code to store a vector of boolean values :

```
  std::vector<bool> bits;
  bits.assign (4, true);
  bits[1] = false;
  config.store ("bits", bits, "An array of bits");
```

Sample code to fetch a vector of boolean values :

```
  std::vector<bool> bits;
  if (config.has_key ("bits") && config.is_vector ("bits"))
    {
      config.fetch ("bits", bits);
    }
```

2.4.2 Store and fetch integer properties

Sample code for adding integer properties in a properties container :

```
datatools::utils::properties config;

// Set explicitely the values of some integer properties :
config.store ("number", 5, "Some number of something");
config.store_integer ("another_number", 8);
```

Sample code for checking the existence of some integer properties in a properties container and fetch the associated value :

```
if (config.has_key ("number") &&
    config.is_integer ("number") &&
    config.is_scalar ("number")  )
  {
    std::cout << "Integer scalar property 'number' exists !" << std::endl;
  }

if (config.is_integer ("another_number"))
  {
    bool another_number = config.fetch_integer ("another_number");
    std::cout << "Fetched integer value for 'another_number' is : "
              << another_number << std::endl;
  }
```

Sample code to store a vector of integer values :

```
  std::vector<int32_t> some_numbers;
  some_numbers.assign (3, 0);
  some_numbers[0] = 3;
  some_numbers[1] = 1;
  some_numbers[2] = 4;
  config.store ("some_numbers", some_numbers, "An array of integers");
```

Sample code to fetch a vector of integer values :

```
  std::vector<int32_t> some_numbers;
  if (config.has_key ("some_numbers") &&
      config.is_integer ("some_numbers") &&
      config.is_vector ("some_numbers"))
    {
      config.fetch ("some_numbers", some_numbers);
    }
```

Sample code to get the size of a vector of integer values :

```
    std::cout << "Size of the array : "
              << config.size ("some_numbers") << std::endl;
```

Sample code to fetch a value at given index of a vector of integer values :

```
    std::cout << "Value at index 1 in the array : "
              << config.fetch_integer ("some_numbers", 1) << std::endl;
```

Sample code to change a scalar integer value. The `change` methods request that the property exists and has the right type :

```
    config.change ("number", 666);
```

Sample code to change a value at given index from a vector of integers :

```
    config.change ("some_numbers", 7, 1);
```

Sample code to update some integer scalar values. The `update` methods request that the property has the right type is it already exists, otherwise it is created :

```
    config.update ("dummy", 9);
    config.update ("another_number", -7);
```

2.4.3 Store and fetch real properties

The real properties are handled is the same way the integer properties are. The `store`, `store_real`, `is_real`, `change`, `change_real`, `update`, `update_real`, `fetch`, `fetch_real`, `fetch_real_scalar`, `fetch_real_vector` methods are implemented.

2.4.4 Store and fetch string properties

The string properties are handled is the same way the integer properties are. The `store`, `store_string`, `is_string`, `change`, `change_string`, `update`, `update_string`, `fetch`, `fetch_string`, `fetch_string_scalar`, `fetch_string_vector` methods are implemented.
Reminder: a string value cannot contents the double quote character, because it is used as the string delimiter character.

3 The datatools::utils::multi_properties class

3.1 Introduction

We have seen in the previous section that the `datatools::utils::properties` class can be used to store simple arbitrary parameters of simple types. It is particularly useful to represent and manipulate some list of configuration parameters to setup an application or the behaviour of an algorithm.
However, in a complex software environment, there are many coexisting components that may need such configuration interface. Using an unique properties container to manage all the settings for all components of a large scale application is not very practical or desirable. For example, several individual components may implement a configuration interface based on a properties container and associated configuration ASCII files. Integrating such components in a larger software framework should not enforce developpers to rewrite the global configuration interface and the users to change their configuration file formatting. A better approach is likely to reuse the individual configuration interfaces from each component, eventually adding some extension to it to support a higher level of functionnality at the scale of the full application.
The `datatools::utils::multi_properties` class is designed to address this use case, among others. It is a meta-container that contains several sections, each section being implemented as a `datatools::utils::properties` container. Practically, a multi-properties container is a dictionnary of which each element (properties container) are stored with an unique mandatory access key (or name). The naming policy is the same than the one used for property records in a properties container.

3.2 Header file and instantiation

In order to use `datatools::utils::multi_properties` objects, one must use the following include directive:

```
#include <datatools/utils/multi_properties.h>
```

The declaration of a `datatools::utils::multi_properties` instance can be simply done with:

```
datatools::utils::multi_properties my_multi_container;
```

Note the use of the nested `datatools::utils` namespace. Alternatively one can use :

```
using namespace datatools::utils;
multi_properties my_multi_container;
```

or:

```
namespace du = datatools::utils;
du::multi_properties my_multi_container;
```

3.3 Design

3.3.1 Names and metas

As mentionned above, a multi-properties container behaves like a dictionary of properties containers. A given stored properties container is addressed through its unique key (or *name*). It is also optionally associated to a *meta* data string that gives some additional information about the properties container stored in a given section of a multi-properties container.

3.3.2 Basics

When instantiated, a multi-properties is given:

- an optional description string (methods:`set_description`, `get_description`),
- a mandatory string to label the *key* in human readable format (default is `"name"`, methods: `set_key_label`, `get_key_label`),
- a optional string to label the *meta* data in human readable format (default is `"type"`, methods: `set_meta_label`, `get_meta_label`). This meat label can be forced to the empty string. In this case, no meta data is used by the container.

The program source sample 5 shows how to declare and initialize a multi-properties object.

```
#include <iostream>

#include <datatools/utils/multi_properties.h>

int main (void)
{
  datatools::utils::multi_properties mconfig;

  mconfig.set_description ("A test multi-properties container");
  mconfig.set_key_label ("name");
  mconfig.set_meta_label ("type");

  return 0;
}
```

Program 5: Declare and setup a `datatools::utils::multi_properties` object.

3.3.3 Container interface

The container interface of a multi-properties object enables to :

- `has_key`, `has_section` : check if a section with a given key exists,
- `get` : provide access to a section entry with a given name,

- `add` : add a new section with given name and optional meta strings,
- `remove` : remove an existing new section with given name,
- `size` : return the number of stored properties,
- `clear` : remove all stored properties container items.

The program source sample 6 shows how to add some empty sections in a multi-properties object. It prints the contents of the container in a human readable formaty (sample 8.

```
#include <iostream>

#include <datatools/utils/multi_properties.h>

int main (void)
{
  datatools::utils::multi_properties mconfig;

  mconfig.set_description ("A test multi-properties container");
  mconfig.set_key_label ("name");
  mconfig.set_meta_label ("type");

  // Add new sections with unique key and associated meta strings :
  mconfig.add ("test", "foo");
  mconfig.add ("debug", "gnus");
  mconfig.add ("log", "gnats");

  mconfig.tree_dump (std::cout,
                     "A multi-properties container with 3 empty sections :");
  return 0;
}
```

Program 6: Adding section in a `datatools::utils::multi_properties` object.

It is also possible to store a copy of an independant properties container object within a multi-properties container as shown in sample 9. This technique can be inefficient if the source properties container hosts a large number of properties.
Accessing individual properties containers is their respective section is illustrated with program sample . Here the `has_key("<KEY>")` and chained `get("<KEY>").get_properties()` methods are the basic tools.
A more efficient technique consists in the manipulation of the stored properties container through a reference (mutable or const) as shown on sample .

```
A multi-properties container with 3 empty sections :
|-- Description  : A test multi-properties container
|-- Key label    : "name"
|-- Meta label   : "type"
|-- Entries      : [3]
|   |-- Entry : "debug"
|   |   |-- Key         : "debug"
|   |   |-- Meta        : "gnus"
|   |   `-- Properties : <empty>
|   |       `-- <no property>
|   |-- Entry : "log"
|   |   |-- Key         : "log"
|   |   |-- Meta        : "gnats"
|   |   `-- Properties : <empty>
|   |       `-- <no property>
|   `-- Entry : "test"
|       |-- Key         : "test"
|       |-- Meta        : "foo"
|       `-- Properties : <empty>
|           `-- <no property>
`-- Ordered entries      : [3]
    |-- Entry [rank=0] : "test"
    |-- Entry [rank=1] : "debug"
    `-- Entry [rank=2] : "log"
```

Sample 8: The output of the program 6. We can check that the three sections have been stored. They correspond to still empty properties containers because no property have been stored so far.

```
  datatools::utils::properties config;
  config.store_flag ("dummy");
  config.store ("tmpdir", "/tmp");
  mconfig.add ("config", "blah", config);
```

Sample 9: Copy a standalone properties container object within a multi-properties container.

```
// Add a new section with unique key and associated meta strings :
mconfig.add ("test", "foo");

if (mconfig.has_key ("test"))
  {
    std::cout << "The multi-properties container has a section "
              << "with key 'test'." << std::endl;
    // Update a string property in properties section 'test' :
    mconfig.get ("test").get_properties ().update_string ("bird",
                                                          "African swallow");
    if (! mconfig.get ("test").get_properties ().has_key ("fruit"))
      {
        // Store a new string property in properties section 'test' :
        mconfig.get ("test").get_properties ().store_string ("fruit",
                                                             "coconut");
      }
    // Finally, clear all properties in this section :
    mconfig.get ("test").get_properties ().clear ();
  }
```

Sample 10: Manipulating a properties container object stored in the section of a `datatools::utils::multi_properties` object.

```
if (mconfig.has_key ("test"))
  {
    datatools::utils::properties & test_config
      = mconfig.get ("test").get_properties ();

    test_config.store_flag ("using_excalibur");
    test_config.store ("number_of_witches", 1);
    test_config.update_string ("color", "blue");
    test_config.update_string ("quest", "The search for the Holly Grail");

    const datatools::utils::properties & const_test_config
      = mconfig.get ("test").get_properties ();

    std::cout << "Number of properties in section 'test' : "
              << const_test_config.size () << std::endl;
  }
```

Sample 11: Manipulating a properties container object stored in the section of a `datatools::utils::multi_properties` object through mutable and const references.

3.3.4 Serialization and I/O features

Like the `datatools::utils::properties` class, the `datatools::utils::multi_properties` class has I/O functionnalities :

- ASCII formatting in standard I/O streams.
- High level serialization based on the Boost/Serialization library (not considered here).

The program source 7 shows how to use this class to save a multi-properties object in an ASCII formated file (method: `write`) and to reload it (method: `read`) . The contents of the saved file is shown on sample 12.

3.3.5 Advanced features

Having a look on the program output 8, one should notice that some informations about the order in which the different sections have been stored is recorded inside the multi-properties container.
Such a feature enables not only to benefit of the dictionnary interface of the `datatools::utils::multi_properties` class, but also to access the items with respect to the order used to store them. The `ordered_entries()` and `entries()` methods in the class implements some non mutable access respectively to the ordered collection of sections (at insertion in the multi-properties container) and the non-ordered collection of sections (using the default order provided by the standard `std::map` class).

```
#include <iostream>

#include <datatools/utils/multi_properties.h>

int main (void)
{
  {
    datatools::utils::multi_properties mconfig;

    mconfig.set_description ("A test multi-properties container");
    mconfig.set_key_label ("name");
    mconfig.set_meta_label ("type");

    // fill the multi-properties container :
    mconfig.add ("test", "test::config");
    datatools::utils::properties & test_config
      = mconfig.get ("test").get_properties ();
    test_config.store_flag ("using_excalibur");
    test_config.store ("number_of_witches", 1);
    test_config.update_string ("bird", "African swallow");
    test_config.update_string ("color", "blue");
    test_config.update_string ("quest", "The search for the Holly Grail");

    mconfig.add ("debug", "debug::config");
    datatools::utils::properties & debug_config
      = mconfig.get ("debug").get_properties ();
    debug_config.store_flag ("active");
    debug_config.store ("level",3);
    debug_config.update_string ("filename", "/tmp/debug.out");

    mconfig.add ("math", "math::config");
    datatools::utils::properties & math_config
      = mconfig.get ("math").get_properties ();
    math_config.store_real ("pi", 3.14159);
    math_config.store_real ("big", 1.e300);
    math_config.store_integer ("digit", 12);

    mconfig.write ("multi_properties_4.conf");
  }

  {
    datatools::utils::multi_properties mconfig;
    mconfig.read ("multi_properties_4.conf");
    mconfig.tree_dump (std::cout,
                       "A multi-properties container restored from a file :");
  }
  return 0;
}
```

Program 7: Save and load functionnalities with `datatools::utils::multi_properties` objects. Note that the optional description of the multi-properties container is recorded through a meta-comment (line starting with the "`#@description`" prefix). The mandatory key label and meta label strings are also printed using the `#@key_label` and `#@meta_label` meta-comments. These three informations must be printed before any sections. Each section has a square bracketed header which indicates the key (`name="XXX"`) and the meta string (with `type="YYY"`). The internal of a section conforms the ASCII formatting syntax for `datatools::utils::properties` objects.

```
# List of multi-properties (datatools::utils::multi_properties):

#@description A test multi-properties container
#@key_label   "name"
#@meta_label  "type"

[name="test" type="test::config"]

bird : string = "African swallow"

color : string = "blue"

number_of_witches : integer = 1

quest : string = "The search for the Holly Grail"

using_excalibur : boolean = 1


[name="debug" type="debug::config"]

active : boolean = 1

filename : string = "/tmp/debug.out"

level : integer = 3


[name="math" type="math::config"]

big : real = 1e+300

digit : integer = 12

pi : real = 3.14159


# End of list of multi-properties.
```

Sample 12: The ASCII file generated by program 7.

4 The datatools::utils::things class

4.1 Introduction

The `datatools::utils::things` class is a general container. It is able to store various types of objects, depending on the needs. It is serializable and implements a dictionary interface. Stored objects must fulfill a special interface.

4.2 Header file and instantiation

In order to use `datatools::utils::things` objects, one must use the following include directive:

```
#include <datatools/utils/things.h>
```

The declaration of a `datatools::utils::things` instance can be simply done with:

```
datatools::utils::things my_bag;
```

Note the use of the nested `datatools::utils` namespace. Alternatively one can use :

```
using namespace datatools::utils;
multi_properties my_bag;
```

or:

```
namespace du = datatools::utils;
du::multi_properties my_bag;
```

4.3 Design

4.3.1 Basics concept

A *things* container allows to store an arbitrary number of objects of any type, provided they inherit from the `datatools::serialization::i_serializable` interface. This design choice has been done to enable the *things* container to be serializable itself, using the Boost/Serialization mechanism as the native I/O system. Also a *things* container can store another *things* container. Note that the implementation of the `datatools::utils::things` class relies on runtime type identification (RTTI) functionnalities of the C++ language.

4.3.2 Banks

A *things* container behaves like a dictionnary and stored objects – we also speak about *banks* – must be accessed through a unique *key* (or name). As it is not possible to automatically *guess* the type of objects once they have been stored in the container, one has to use templatized methods to check and manipulate them. However, a special *serialization tag* associated to the stored object is also stored in order to enable some further type identification or introspection functionnalities.

Finally the *things* container stores objects through pointers, and is responsible of the corresponding memory allocation and deallocation of its contents. The consequences of this implementation are:

- the stored objects can only be manipulated by const or mutable *references*,
- the `datatools::utils::things` class is *non-copyable*.

Despite these features (or limitations), the *things* container is a very flexible container that is adapted for many applications. It provides some powerfull management tools :

- add/remove arbitrary *banks* of data (with the serializable interface),
- internal dynamic memory management,
- manipulation of the data stored in banks through references.
- a dictionnary interface to check and access to *banks* of data,
- fully serializable (Boost/Serialization).

4.3.3 Interface

List of public `datatools::utils::things` class methods :

- `size` : return the number of stored objects/banks,
- `empty` : check if the container is empty,
- `reset`, `clear` : remove all stored objects/banks,
- `has`: check if an object/bank with a given name is stored,
- `has_serial_tag`: check if an object/bank with a given name and given *serialization tag* is stored,
- `set_constant` : mark an object/bank with a given name as non-mutable,
- `is_constant` : check if an object/bank with a given name is marked as non-mutable,
- `is_mutable` : check if an object/bank with a given name is mutable,
- `set_description` : set the description string associated to an object/bank with a given name,
- `get_description` : get the description string associated to an object/bank with a given name,
- `get_names` : get the list of names (keys) associated to all object/bank stored in the things container,

- `remove`, `erase` : remove an object/bank stored with a given name,
- `tree_dump` : prints the container in a human friendly format.

List of public template methods for the manipulation of stored objects/banks :

- `add<T>` : add a new object/bank of type `T` with a given name, return a mutable reference to the new allocated instance,
- `is_a<T>` : check if an existing object/bank with a given name is of type `T`,
- `get<T>` : return a non-mutable reference to an existing object/bank with a given name of type `T`,
- `grab<T>` : return a mutable reference to an existing object/bank with a given name of type `T`.

4.4 Examples

4.4.1 Declare a *things* container and add objects in it

The program source sample 8 shows how to declare a *things* container object and add two *properties* objects in it. It prints the contents of the container in a human readable formaty (sample 13.

```
#include <iostream>

#include <datatools/utils/things.h>
#include <datatools/utils/properties.h>

int main (void)
{
  datatools::utils::things bag;

  // Give it a description string :
  bag.set_description ("A test things container");

  // Add two properties objects (serializable) :
  bag.add<datatools::utils::properties> ("foo");
  bag.add<datatools::utils::properties> ("bar");

  bag.tree_dump (std::cout,
                 "A things container with 2 banks :");
  return 0;
}
```

Program 8: Adding objects in a `datatools::utils::things` container. Note that all low-level memory allocation operation is performed internally. The user does not have to care about it.

```
A things container with 2 banks :
|-- Description  : A test things container
|-- Name : "bar"
|   |-- Const  : 0
|   `-- Handle  : 0x816c870 (serial tag: 'datatools:utils::properties')
`-- Name : "foo"
    |-- Const  : 0
    `-- Handle  : 0x816c7f0 (serial tag: 'datatools:utils::properties')
```

Sample 13: The output of the program 8. We can check that the two banks of data have been stored. Both are `datatools::utils::properties` objects, which are empty here.

4.4.2 Instant manipulation of added objects through references

The program source sample 14 shows how to declare a *things* container object, add two *properties* objects in it and make use of the mutable reference returned by the template `add` methods.

```
  // Add two properties objects (serializable) and
  // use the mutable reference returned by the 'add' method :

  // 'On the fly' usage of the returned reference :
  bag.add<datatools::utils::properties> ("foo").store_flag ("debug");

  // 'Off-line' usage of the returned reference :
  datatools::utils::properties & bar_ref =
    bag.add<datatools::utils::properties> ("bar");
  // Now we are free to use the 'properties' class interface :
  bar_ref.store_flag ("test");
  bar_ref.store ("number_of_gate", 9);
  bar_ref.store ("pi", 3.14159);
  bar_ref.store ("name", "John Doe");
```

Sample 14: We use the references returned while adding objects in a `datatools::utils::things` container in order to manipulate the stored object.

4.4.3 Get references to manipulate stored objects

The program source sample 16 shows how to obtain non-mutable and mutable references to an object stored in a *things* container object with a given name. The `get<T>` and `grab<T>` template methods respectively return non-mutable and mutable references (here class `T` corresponds to `datatools::utils::properties`). The references are then used to manipulate the stored object through its own interface.

4.4.4 Create a new class to be stored in a *things* container

The program source samples 9 and 10 shows how to declare new serializable class `storable_type` that can be stored in a *things* container object (output is shown on

```
// Check the availability of a 'properties' object
// in the 'things' container and get :
//   a) a non-mutable reference to it
//   b) a mutable reference to it
if (bag.has ("foo") && bag.is_a<datatools::utils::properties> ("foo"))
  {
    const datatools::utils::properties & const_foo_ref =
      bag.get<datatools::utils::properties> ("foo");
    if (! const_foo_ref.has_key ("Devil"))
      {
        datatools::utils::properties & foo_ref =
          bag.grab<datatools::utils::properties> ("foo");
        foo_ref.store ("Devil", 666);
      }
 }
```

Sample 15: We explicitely initialize references to an object stored in the `datatools::utils::things` container in order to manipulate the stored object.

sample 16).

```
#include <iostream>
#include <datatools/utils/things.h>
#include <datatools/utils/properties.h>
#include <boost/cstdint.hpp>
#include <datatools/serialization/i_serializable.h>

/*** Interface of the class ***/
class storable_type : DATATOOLS_SERIALIZABLE_CLASS
{
public:
  storable_type ();
  void set_value (int32_t);
  int32_t get_value () const;
private:
  int32_t _value_;
  /* interface i_serializable */
  DATATOOLS_SERIALIZATION_DECLARATION();
};
// interface for pointer serialization :
#include <boost/serialization/export.hpp>
BOOST_CLASS_EXPORT_KEY2(storable_type, "storable_type")

/*** Implementation of the class ***/
// serial serialization tag :
DATATOOLS_SERIALIZATION_SERIAL_TAG_IMPLEMENTATION(storable_type, "storable_type")
storable_type::storable_type () : _value_ (0)
{
}
void storable_type::set_value (int32_t value_)
{
  _value_ = value_;
  return;
}
int32_t storable_type::get_value () const
{
  return _value_;
}

/*** Serialization implementation ***/
#include <boost/serialization/nvp.hpp>
// template Boost serialization method :
template<class Archive>
void storable_type::serialize (Archive & ar_, const unsigned int version_)
{
  ar_ & DATATOOLS_SERIALIZATION_I_SERIALIZABLE_BASE_OBJECT_NVP;
  ar_ & boost::serialization::make_nvp ("value", _value_);
  return;
}
// automated pre-instantiation of serialization template code :
#include <datatools/serialization/archives_instantiation.h>
DATATOOLS_SERIALIZATION_CLASS_SERIALIZE_INSTANTIATE_ALL(storable_type)
// export class for pointer serialization :
BOOST_CLASS_EXPORT_IMPLEMENT(storable_type)
```

Program 9: Creation of a new storable class for `datatools::utils::things` container.

```
int main (void)
{
  datatools::utils::things bag;
  bag.set_description ("A test things container");

  bag.add<datatools::utils::properties> ("foo").store_flag ("test");
  bag.add<storable_type> ("dummy").set_value (12345);

  bag.tree_dump (std::cout, "The bag: ");
  std::cout << std::endl;

  bag.get<datatools::utils::properties> ("foo").tree_dump (std::cout,
                                                           "foo : ");
  const storable_type & dummy_ref = bag.get<storable_type> ("dummy");
  std::cout << "dummy's value = " << dummy_ref.get_value () << std::endl;
  std::cout << std::endl;

  bag.remove ("foo");
  bag.tree_dump (std::cout, "The bag: ");
  return 0;
}
```

Program 10: Program to store a new storable class (see program 9) in a `datatools::utils::things` container.

```
The bag:
|-- Description  : A test things container
|-- Name : "dummy"
|   |-- Const  : 0
|   `-- Handle  : 0x985abd8 (serial tag: 'storable_type')
`-- Name : "foo"
    |-- Const  : 0
    `-- Handle  : 0x985ab70 (serial tag: 'datatools:utils::properties')

foo :
`-- Name : "test"
    |-- Type  : boolean (scalar)
    `-- Value : 1
dummy's value = 12345

The bag:
|-- Description  : A test things container
`-- Name : "dummy"
    |-- Const  : 0
    `-- Handle  : 0x985abd8 (serial tag: 'storable_type')
```

Sample 16: The output of the program 9-10.

5 Conclusion

Three foundation container classes have been presented in this note :

- the `datatools::utils::properties` class can be used as a container for configuration parameters for complex objects, algorithms or application programs; it can also be used within classes by aggregation to serve as a versatile and extensible store for arbitrary data; it provides ASCII human friendly formatted I/O functionality,
- `datatools::utils::multi_properties` can be used to handle a complex set of *configurations*, each configuration implements a `datatools::utils::properties` object and is stored within a named *section*; it provides ASCII human friendly formatted I/O functionality,
- `datatools::utils::things` is a versatile container for arbitrary serializable objects; it can be used as the core container for some complex data model.

All three classes are serializable with the native Boost/Serialization system.