C++ API Guide

Version 1.2 30 September 2021

Ciaran McHale

www.config4star.org

Availability and Copyright

Availability

The Config4* software and its documentation (including this manual) are available from www.config4star.org. The manuals are available in several formats:

- HTML, for online browsing.
- PDF (with hyper links) formatted for A5 paper, for on-screen reading.
- PDF (without hyper links) formatted 2-up for A4 paper, for printing.

Copyright

AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Contents

Chapter 1

Introduction

1.1 Purpose of this Manual

This manual provides a reference for the application programming interface (API) of Config4Cpp. This manual does *not* provide a beginner's tutorial on Config4Cpp. You can find such a tutorial in the *Config4* Getting Started Guide*.

The rest of this chapter discusses the principles that underpin the API of Config4Cpp. Knowledge of these principles makes it easier to understand the API. The chapters that follow discuss the API of individual classes.

1.2 Namespace

All the classes of Config4Cpp are defined in the `config4cpp` namespace. For conciseness, the `config4cpp::` namespace prefix is not explicitly stated in the discussion of classes and operations in this manual.

1.3 Memory management

The API is designed so that ownership of heap-allocated memory is *not* transferred from Config4Cpp to application code, or vice versa. For example, if an application calls `lookupString()`, the application should *not* `delete` the returned string when it is finished processing it. There are two motivations behind this memory management policy.

The first motivation is to simplify the API and, in doing so, minimize the chances of memory leaks in applications that use Config4Cpp.

The second motivation comes from the Microsoft Visual C++ compiler. This compiler does something unusual: it implements the `new` and `delete` operators with one algorithm if you compile in *debug* mode, but implements those operators with an incompatible, algorithm if you compile in *release* (that is, non-debug) mode. If an application contains some files that were compiled in debug mode and other files that were compiled in release mode, then the application might crash if memory allocated with the debug version of `new` is freed using the release version of `delete`, or vice versa. Typically, this problem occurs when application code is compiled in, say, debug mode, but is linked against a third-party library that was compiled in release mode. Many vendors of third-party libraries work around this problem by supplying both debug and release versions of their libraries. The Config4Cpp library takes a different approach to avoiding mismatched versions of `new` and `delete`. Put simply, any memory allocated *inside* Config4Cpp is later freed from *inside* Config4Cpp. By doing this, it is not necessary to compile both debug and release versions of the Config4Cpp library.

1.4 Portability

One of the design goals of Config4Cpp is portability. Not only should Config4Cpp compile on many operating systems; it should also compile with both new and older C++ compilers. Some older C++ compilers do not fully support the standard C++ library. Because of this, the implementation of Config4Cpp does not use anything in the standard C++ library. This has some knock-on effects, as I now discuss.

First, the implementation of Config4Cpp does not use the IO streams library to read a configuration file. This is because using IO streams means using *either* classic IO streams (that is, `<iostream.h>`), *or* standard IO streams (that is, `<iostream>`). In many compilers, these are not link compatible so if your application uses classic IO streams, then you cannot link your application with a third-party library that uses standard IO streams. It would have been possible for Config4Cpp to use conditional compilation to let a person compiling Config4Cpp specify if it should use classic or standard IO streams. However, it turns out to be just as easy for Config4Cpp to use `<stdio.h>` in the standard C library to read configuration files, and so bypass unpleasant issues associated

with the choice between classic or standard IO streams.

Second, the implementation of Config4Cpp does not use any types in the standard template library (STL). Nor does Config4Cpp define any template types itself. One reason for this is to avoid the portability headache of dealing with platform-specific build issues when using template types. Another reason is that some projects wishing to use Config4Cpp might be using a relatively old C++ compiler that does not support template types. The amount of code in Config4Cpp could have been reduced by making use of `std::string`, `std::vector` and `std::map`. However, for the sake of portability, the Config4Cpp library implements its own utility classes that implement similar functionality.

Most of the external APIs used by the implementation of Config4Cpp are functions in the standard C library, which is much more portable than the C++ library. However, it has not been possible to write Config4Cpp entirely using only portable APIs: a few platform-specific APIs have been required too; these are discussed in the *Config4* Maintenance Guide*.

1.5 Error Reporting

The Config4Cpp parser does not make any attempt at error recovery. Instead, it stops at the first error it encounters, and reports the error by throwing an exception. The lack of error recovery helps to keep the implementation of the parser simple. It also simplifies the public API because the ability to report multiple parsing errors would have required a more complex API.

1.6 Specifying Scoped Names

Many of the operations in Config4Cpp work with scoped names, for example, `foo_srv.log.dir`. Typically, the first part, `foo_srv`, is a scope for a particular application, and the remainder, `log.dir`, is a configuration variable used by that application. It can be useful to change the application name (`foo_srv`) *without* needing to change lots of code. It would not be possible to do this if `foo_srv.log.dir` appeared in application code. Instead, it is best for the two parts of a name to be specified separately, and then merged to form the fully-scoped name.

It would be tedious for developers to do such merging manually. To avoid this, the Config4Cpp operations that work on scoped names take

two string parameters. Internally, the operations merge the strings to obtain a fully-scoped name. For example, you can access the value of `foo_srv.log.dir` with the following statement:

```
logDir = cfg->lookupString("foo_srv", "log.dir");
```

The intention is that an application can declare a variable called, say, `scope` and initialize its value from a command-line argument. Then the application can access configuration information from within that scope by using code like that shown below:

```
logDir = cfg.lookupString(scope, "log.dir");
```

By rerunning an application with a different command-line argument, you can change the scope used to configure the application. This provides a lot of flexibility. For example, you might have one configuration scope for running an application *without* debugging diagnostics, and another scope that *enables* debugging diagnostics. Alternatively, you might have a separate scope for each instance of a replicated server application.

1.7 Support Classes

I explained in Section 1.4 on page 2 that, to increase its portability, Config4Cpp avoids use of template types, including those in the standard C++ library. Instead, Config4Cpp implements two support classes that provide functionality somewhat similar to that provided by some classes in the standard C++ library.

1.7.1 The `StringBuffer` Class

Instead of using the `std::string` and `std::stringstream` classes, Config4Cpp defines its own `StringBuffer` class that provides similar-ish functionality. The public API of this class is shown in Figure 1.1.

The default constructor initialises the `StringBuffer` to maintain an empty C-style string (`""`). The constructor taking a `const char*` parameter takes a deep copy of that parameter. Likewise, the copy constructor makes a deep copy of the string contained inside its parameter.

You can grow the C-style string inside a `StringBuffer` by calling the overloaded `append()` operation or by using the overloaded `<<` operator.

You can replace the C-style string contained in a `StringBuffer` by using the overloaded assignment operator (`"="`).

Figure 1.1: The `StringBuffer` class

```
// Access with #include <config4cpp/StringBuffer.h>
// or          #include <config4cpp/Configuration.h>

class StringBuffer
{
public:
    StringBuffer();
    StringBuffer(const char * str);     // deep copy
    StringBuffer(const StringBuffer &); // deep copy
    ~StringBuffer();

    const char *   c_str() const;
    char           lastChar() const;
    int            length() const;
    void           empty();
    void           deleteLastChar();
    StringBuffer & append(const StringBuffer & other); // deep copy
    StringBuffer & append(const char * str);           // deep copy
    StringBuffer & append(int val);
    StringBuffer & append(float val);
    StringBuffer & append(char ch);

    StringBuffer & operator=(const char * str);           // deep copy
    StringBuffer & operator=(const StringBuffer & other); // deep copy
    char           operator[](int index) const;
    char &         operator[](int index);
    StringBuffer & operator<<(const StringBuffer & other);// deep copy
    StringBuffer & operator<<(const char * str);          // deep copy
    StringBuffer & operator<<(int val);
    StringBuffer & operator<<(float val);
    StringBuffer & operator<<(char ch);
};
```

The `c_str()` operation returns a pointer to the C-style string contained inside the `StringBuffer`.

The `empty()` operation resets the `StringBuffer` to having an empty C-style string.

The `length()` operation returns the length of the C-style string contained inside the `StringBuffer`.

The `lastChar()` operation returns the final (non-null) character of the C-style string contained in the `StringBuffer` if the string is not empty;

otherwise, it returns `'\0'`.

The `deleteLastChar()` operation removes the last character from the C-style string contained in the `StringBuffer`. It is an error to call this operation on an empty `StringBuffer`.

The `StringBuffer` class is used mainly by the internals of Config4Cpp. However, `StringBuffer` *is* exposed to application programmers via parameters to a small number of public operations. In such cases, it is always used as an "out" or "in-out" parameter, so that application code does not have to explicitly delete a heap-allocated string allocated by the internals of Config4Cpp.

1.7.2 The `StringVector` Class

Config4Cpp defines its own `StringVector` class that provides functionality similar-ish to that provided by `std::vector<std::string>`. The public API of `StringVector` is shown in Figure 1.2.

The `StringVector` class provides a simplification wrapper around a null-terminated array of C-style strings. The default constructor initialises the `StringVector` to maintain an empty, null-terminated array. The copy constructor makes a deep copy of all the C-style strings in the parameter.

Strings are added to a `StringVector` by calling the overloaded `add()` operation, which grows the internal array of the `StringVector` if necessary. If you know in advance how many strings you will add, then you can invoke `ensureCapacity()` to prevent repeated re-allocations of the internal array.

The `c_array()` operation provides access to the internal array of C-style strings. The returned array is always null-terminated. This operation is overloaded to provide access to the null-terminated array of strings with and without a count of the number of strings in the array.

The `length()` operation returns the number of C-style strings currently in the array. You can use `operator[]` to get read-only access to individual items of the array, and call `replace()` to replace an individual item.

The `removeLast()` operation removes the string at the end of the array, while `empty()` resets the `StringVector` back to being an empty, null-terminated array.

The `sort()` operation uses `strcmp()` as its comparison function to sort the strings in the array.

The `bSearchContains()` operation performs a binary search (using

Figure 1.2: The `StringVector` class

```
// Access with #include <config4cpp/StringVector.h>
// or          #include <config4cpp/Configuration.h>

class StringVector
{
public:
    StringVector(int initialCapacity = 10);
    StringVector(const StringVector &);    // deep copy
    ~StringVector();

    void           ensureCapacity(int size);
    void           add(const char * str);
    void           add(const StringBuffer & strBuf);
    void           add(const StringVector & other); // adds all items

    void           c_array(const char**& array, int& arraySize) const;
    const char **  c_array() const;

    int            length() const;
    const char *   operator[](int index) const;
    StringVector & operator=(const StringVector & other); // deep copy

    void           replace(int index, const char * str); // deep copy
    void           removeLast();
    void           empty();

    void           sort();
    bool           bSearchContains(const char * str) const;
};
```

`strcmp()` as its comparison function) on the array of strings, which are assumed to be sorted. If the target string is found, then this operation returns `true`; otherwise, it returns `false`.

The `StringVector` class is used mainly by the internals of Config4Cpp. However, `StringVector` *is* exposed to application programmers via parameters to some public operations, such as `lookupList()`, `insertList()`, `listFullyScopedNames()` and `listLocallyScopedNames()`.

Chapter 2

The **Configuration** Class

2.1 The **ConfigurationException** Class

An exception of type `ConfigurationException` is thrown if any Config4Cpp operation fails. The public API of this class is shown in Figure 2.1.

Figure 2.1: The `ConfigurationException` class

```
// Access with #include <config4cpp/ConfigurationException.h>
// or          #include <config4cpp/Configuration.h>

class ConfigurationException
{
public:
    ConfigurationException(const char * str);
    ConfigurationException(const ConfigurationException & other);
    ~ConfigurationException();
    const char * c_str() const;
};
```

Application code can access a string description of the exception by invoking the `c_str()` operation, as shown below:

```
try { ... } catch(const ConfigurationException & ex) {
    cerr << ex.c_str() << endl;
}
```

As explained in Section 1.4 on page 2, to avoid having a dependency on either classic or standard IO streams, Config4Cpp is *not* implemented with the IO Stream library. Because of this, Config4Cpp does not define an operator for streaming a `ConfigurationException` to an output stream. There is nothing preventing you from defining such a streaming operator in the global scope in your own applications. Alternatively, you can stream an exception to an output stream by explicitly invoking the `c_str()` operation, as shown in the previous example.

2.2 The `create()` and `destroy()` Operations

Figure 2.2 shows the operations that are used to create and destroy a `Configuration` object.

Figure 2.2: Initialization and destruction APIs for `Configuration`

```
// Access with #include <config4cpp/Configuration.h>

class Configuration {
public:
    static Configuration * create();
    virtual void destroy();
    ...
};
```

You use the static `create()` operation to create a `Configuration` object. A newly created `Configuration` object is empty initially. You can then populate it and access its contents, as I will discuss in the following sections of this chapter. Finally, you should call `destroy()` to reclaim the memory of the `Configuration` object.

The correct behaviour of Config4Cpp depends on the locale being set correctly. Because of this, it is advisable to call `setlocale()` *before* invoking any Config4Cpp APIs. If you do this, then Config4Cpp will be able to handle characters defined in your locale, such as European accented characters or Japanese ideographs. If you neglect to call `setlocale()`, then Config4Cpp is likely to correctly process *only* characters in the 7-bit US ASCII character set. Figure 2.3 illustrates how to call `setlocale()`, `create()` and `destroy()`.

Most of the operations defined in the `Configuration` class can throw `ConfigurationException` exceptions, so those operations should be called

Figure 2.3: Example of creating and destroying a `Configuration` object

```
#include <locale.h>
#include <config4cpp/Configuration.h>
using namespace config4cpp;
...
setlocale(LC_ALL, "");
Configuration * cfg = Configuration::create();
try {
  ... // invoke operations on cfg
} catch(const ConfigurationException & ex) {
  cout << ex.c_str() << endl;
}
cfg->destroy();
```

from inside a `try-catch` clause. However, the `create()` and `destroy()` operations do *not* throw that exception, so, as shown in Figure 2.3, they can be called from outside a `try-catch` clause.

2.3 Utility Operations

Config4Cpp provides several utility operations, shown in Figure 2.4, that you may need to use from time to time.

Figure 2.4: Utility operations

```
class Configuration {
public:
    static void mergeNames(const char *    scope,
                           const char *    localName,
                           StringBuffer &  fullyScopedName);
    static bool patternMatch(const char * str,
                             const char * pattern);
    static int mbstrlen(const char * str);
    ...
};
```

As I discussed in Section 1.6 on page 3, many Config4* operations take a pair of parameters, `scope` and `localName`, that, when merged, specify the fully-scoped name of an entry in a `Configuration` object. The `mergeNames()` operation performs that merging, and it puts the result into the `fullyScopedName` parameter. The fully-scoped name is usually

of the form *scope.localName*, but if either `scope` or `localName` is an empty string, then the dot (`"."`) is omitted when performing the merge.

The `patternMatch()` operation compares a string against a pattern, and returns `true` if they match. Within the pattern, the `"*"` character acts as a wildcard that matches zero or more characters. For example:

```
Configuration::patternMatch("Hello, world", "Hello*") → true
Configuration::patternMatch("Hello, world", "123*89") → false
```

Your locale setting specifies, amongst other things, the character set being used. If the locale specifies an 8-bit character set, such as ASCII or ISO-Latin-1, then each character is fully encoded in a single byte. However, if the locale specifies a multi-byte character set, such as UTF-8, then *some* characters may be encoded in a single byte but other characters will be encoded in a multi-byte sequence. Because of this possibility, you cannot rely on using the `strlen()` function to return the number of *characters* (instead of *bytes*) in a string. The `mbstrlen()` operation returns the number of characters in a string, regardless of the character set specified by the locale setting; it returns `-1` if the string contains invalid multi-byte characters.

2.4 The `parse()`, `fileName()` and `empty()` Operations

Figure 2.5 shows the signatures of the `fileName()`, `parse()` and `empty()` operations.

The `fileName()` operation returns the name of the most recently parsed file. If `parse()` has not previously been called, then `fileName()` returns an empty string.

I defer discussion of the one-parameter version of `parse()` until Section 2.4.4 on page 15 because it is just a simplified version of the three-parameter version of `parse()`. In the three-parameter version of `parse()`, the value of the first parameter determines the meaning of the other parameters.

- If the first parameter is `INPUT_FILE`, then the second parameter is the name of the file to be parsed, and the third parameter is ignored.

  The second parameter will be the value returned from future calls to `fileName()`.

Figure 2.5: The parse(), fileName() and empty() operations

```
class Configuration {
public:
    enum SourceType {INPUT_FILE, INPUT_STRING, INPUT_EXEC};

    const char * fileName() const;

    void parse(Configuration::SourceType    sourceType,
               const char *                 source,
               const char *                 sourceDescription = "")
                                       throw(ConfigurationException);

    void parse(const char * sourceTypeAndSource)
                                       throw(ConfigurationException);
    void empty();
    ...
};
```

- If the first parameter is INPUT_STRING, then the second parameter is a string to be parsed.

  If the third parameter is *not* an empty string, then it will be the value returned from future calls to fileName(); otherwise the string "<string-based configuration>" will be the value returned from future calls to fileName().

- If the first parameter is INPUT_EXEC, then the second parameter is an external command to be executed and whose standard output is to be parsed.

  If the third parameter is *not* an empty string, then it will be the value returned from future calls to fileName(); otherwise the string resulting from appending the second parameter to "exec#" will be the value returned from future calls to fileName().

The string returned from fileName() is used at the start of text messages inside exceptions. For example, many components of Config4Cpp (including the parser, schema validator and lookup operations) format exception messages as shown below:

```
if (...) {
    StringBuffer    msg;
    msg << cfg->fileName() << ": something went wrong";
    throw ConfigurationException(msg.c_str());
```

```
}
```

For this reason, if you call `parse()` with `INPUT_STRING` for the first parameter, then you should ensure that the value of the third parameter acts as a descriptive "file name".

2.4.1 Parsing a File

The code segment in Figure 2.6 shows an example use of `parse()`. The `create()` operation creates a `Configuration` object that is empty initially. Then the `parse()` operation is used to populate the `Configuration` object. A `try-catch` clause is used to print any exception that might be thrown. Once the `Configuration` object has been populated, lookup operations (which I will discuss in Section 2.6) can be used to access information in it. Finally, `destroy()` is called to reclaim the memory of the `Configuration` object when it is no longer required.

Figure 2.6: An example of using `parse()`

```
Configuration * cfg = Configuration::create();
try {
    cfg->parse(Configuration::INPUT_FILE, "myFile.cfg");
    ... // invoke lookup operations
} catch(const ConfigurationException & ex) {
    cerr << ex.c_str() << endl;
}
cfg->destroy();
```

2.4.2 Parsing the Output of a Command

The example in Figure 2.6 used the following to parse a file:

```
cfg->parse(Configuration::INPUT_FILE, "myFile.cfg");
```

If, instead of parsing a file, you want to execute a command and parse its standard output, then you can do so as follows:

```
cfg->parse(Configuration::INPUT_EXEC, "curl -sS http://host/file.cfg");
```

Using `INPUT_EXEC` for the first parameter tells `parse()` to to interpret the second parameter as the name of a command to be executed.

2.4.3 Parsing a String

The example in Figure 2.6 used the following to parse a file:

```
cfg->parse(Configuration::INPUT_FILE, "myFile.cfg");
```

If, instead of parsing a file, you want to parse a string, then you can do so as follows:

```
const char * cfgStr = ...;
cfg->parse(Configuration::INPUT_STRING, cfgStr, "embedded configuration");
```

Using `INPUT_STRING` for the first parameter tells `parse()` to interpret the second parameter as configuration data that should be parsed directly, and the third parameter is the "file name" that will be used when reporting errors. You can initialise the second parameter in a variety of ways, for example:

- You could use the `config2cpp` utility to convert a configuration file into (a class wrapper around) a string that is embedded into the application. In this case, you might use `"embedded configuration"` or `"fallback configuration"` as the third parameter.

- Perhaps you are developing a client-server application that serialises messages into Config4* syntax and then transmits them across a socket connection. In the receiving application, the second parameter to `parse()` would be a message that was read from a socket connection. In this case, you might use `"incoming message"` as the third parameter.

2.4.4 The Simplified Version of `parse()`

The one-parameter version of `parse()` is a simplification wrapper around the three-parameter version. Its implementation is shown in Figure 2.7.

The following examples show how to use this simplified version:

```
cfg->parse("exec#curl -sS http://host/file.cfg");
cfg->parse("file#file.cfg");
cfg->parse("file.cfg");
```

In practice, the parameter to this operation is unlikely to be hard-coded into an application, but rather will come from, say, a command-line option or an environment variable.

Figure 2.7: Simplified version of `parse()`

```
void
Configuration::parse(const char * str) throw(ConfigurationException)
{
    if (strncmp(str, "exec#", 5) == 0) {
        parse(Configuration::INPUT_EXEC, &(str[5]));
    } else if (strncmp(str, "file#", 5) == 0) {
        parse(Configuration::INPUT_FILE, &(str[5]));
    } else {
        parse(Configuration::INPUT_FILE, str);
    }
}
```

2.4.5 Parsing Multiple Files and the `empty()` Operation

If you want to parse multiple configuration files, then you can use multiple `Configuration` objects. Alternatively, you can reuse the same object multiple times. If you do this, then you will probably want to call `empty()` between successive calls of `parse()`, as shown in Figure 2.8. The `empty()` operation has the effect of removing all variables and scopes from the `Configuration` object.

Figure 2.8: Calling `parse()` and `empty()` multiple times

```
Configuration * cfg = Configuration::create();
try {
    cfg->parse("file1.cfg");
    ... // Access the configuration information
    cfg->empty();
    cfg->parse("file2.cfg");
    ... // Access the configuration information
    cfg->empty();
    cfg->parse("file3.cfg");
    ... // Access the configuration information
    cfg->empty();
    cfg->parse("file4.cfg");
    ... // Access the configuration information
} catch(const ConfigurationException & ex) {
    cerr << ex.c_str() << endl;
}
cfg->destroy();
```

It is legal to call `parse()` multiple times *without* calling `empty()` between successive calls. If you do this, then each call to `parse()` merges its information with information already in the `Configuration` object. The Config4* parser implements the `@include` statement by (recursively) calling `parse()`, so you can think of multiple calls to `parse()` without calls to `empty()` as being similar to multiple `@include` statements.

It is difficult to think of a compelling reason why you might want to use a single `Configuration` object to parse multiple configuration files *without* calling `empty()` between successive calls of `parse()`. However, it is useful to know what the semantics of doing so are, because it can help you understand what is happening if you forget to call `empty()` between calls to `parse()` on the same `Configuration` object.

2.5 Insertion and Removal Operations

Most applications will populate a `Configuration` object by parsing a configuration file. However, it is possible to populate a `Configuration` object by using the operations shown in Figure 2.9.

The `insertString()` operation inserts into the `Configuration` object an entry using the fully-scoped name (obtained by merging the `scope` and `localName` parameters) and the specified `strValue`. If a variable of the same name already exists in the `Configuration` object, then it is replaced with the new value.

The `insertList()` operation inserts into the `Configuration` object an entry using the fully-scoped name (obtained by merging the `scope` and `localName` parameters) and the specified list. If a variable of the same name already exists in the `Configuration` object then it is replaced with the new value. The `insertList()` operation is overloaded so you can specify the list in one of three different ways: as an array of strings plus the size of the array, as an array of strings terminated by a null pointer, or as a `StringVector`.

The `insertString()` and `insertList()` operations make a deep copy of the value when inserting it into the `Configuration` object.

The `ensureScopeExists()` operation merges the `scope` and `localName` parameters to obtain a fully-scoped name. It ensures that a scope with this fully-scoped name exists. If any ancestors of the specified scopes are missing, then they are also created. Internally, the `insertString()` and `insertList()` operations call `ensureScopeExists()`. Because of this, applications rarely need to call `ensureScopeExists()` directly.

Figure 2.9: Insertion and removal operations

```
class Configuration {
public:
    void insertString(
                    const char *            scope,
                    const char *            localName,
                    const char *            strValue)
                                        throw(ConfigurationException);
    void insertList(
                    const char *            scope,
                    const char *            localName,
                    const char **           array,
                    int                     arraySize)
                                        throw(ConfigurationException);
    void insertList(
                    const char *            scope,
                    const char *            localName,
                    const char **           nullTerminatedArray)
                                        throw(ConfigurationException);
    void insertList(
                    const char *            scope,
                    const char *            localName,
                    const StringVector &    vec)
                                        throw(ConfigurationException);
    void ensureScopeExists(
                    const char *        scope,
                    const char *        localName)
                                        throw(ConfigurationException);

    void remove(const char * scope, const char * localName)
                                        throw(ConfigurationException);
    ...
};
```

The `remove()` operation merges `scope` and `localName` to form a fully-scoped name. It then removes the entry with the specified name.

If you are making use of identifiers that have `"uid-"` prefixes, then it is your duty to expand such identifiers before invoking any of the operations listed in Figure 2.9. Section 2.12 on page 36 explains how to expand identifiers that have `"uid-"` prefixes.

2.6 The lookup<Type>() Operations

Figure 2.10 lists lookup-style operations that you can use to access the values of configuration variables. There are a *lot* of lookup operations, for the following reasons.

- Syntactically, variables in a configuration file are either strings or lists. However, strings are often used to encode other types, such as integers, floats, booleans, durations and so on. Because of this, there are type-safe lookup operations that convert a string value to another format. For example, `lookupInt()` converts a string value to an `int`, and `lookupBoolean()` converts a string value to a `bool`.
- The lookup operations are overloaded so that you can optionally specify a default value that should be returned if the specified configuration variable is not present.
- The `lookupList()` operation is overloaded so you can access a list as an array of strings or a `StringVector`.

The lookup operations perform error checking. For example, if the `lookupInt()` operation cannot convert the string value into an integer, then it throws an exception that contains an easy-to-understand error message. Likewise, if you do not specify a default value to a lookup operation and the specified configuration variable is missing (from both the main configuration object and fallback configuration), then an exception is thrown.

2.6.1 Lookup Operations for Enumerated Types

Among other parameters, the `lookupEnum()` operation takes an array of `EnumNameAndValue` structures and the size of that array. This operation calls `lookupString()` and then uses information in the array to convert the string value into an integer. For example:

```
EnumNameAndValue colourInfo[] = {
    { "red",   0 },
    { "green", 1 },
    { "blue",  2 },
};
colour = cfg->lookupEnum(scope, "font_colour", "colour", colourInfo, 3);
```

The `typeName` parameter (`"colour"` in the above example) specifies the "type name" of the enum names, and is used to construct an informative error message if an exception is thrown.

Figure 2.10: The lookup<Type>() operations

```
struct EnumNameAndValue {
    const char  *    name;
    int              value;
};

class Configuration
{
public:
    const char * lookupString(
                    const char *            scope,
                    const char *            localName,
                    const char *            defaultVal) const
                                        throw(ConfigurationException);

    const char * lookupString(
                    const char *            scope,
                    const char *            localName) const
                                        throw(ConfigurationException);

    void lookupList(
                    const char *            scope,
                    const char *            localName,
                    const char **&          array,
                    int &                   arraySize,
                    const char **           defaultArray,
                    int                     defaultArraySize) const
                                        throw(ConfigurationException);
    void lookupList(
                    const char *            scope,
                    const char *            localName,
                    const char **&          array,
                    int &                   arraySize) const
                                        throw(ConfigurationException);
    void lookupList(
                    const char *            scope,
                    const char *            localName,
                    StringVector &          list,
                    const StringVector &    defaultList) const
                                        throw(ConfigurationException);
... continued on the next page
```

Figure 2.10 (continued): The lookup<Type>() operations

```
... continued from the previous page
    void lookupList(
                    const char *            scope,
                    const char *            localName,
                    StringVector &          list) const
                                        throw(ConfigurationException);
    int lookupInt(
                    const char *            scope,
                    const char *            localName,
                    int                     defaultVal) const
                                        throw(ConfigurationException);
    int lookupInt(
                    const char *            scope,
                    const char *            localName) const
                                        throw(ConfigurationException);
    float lookupFloat(
                    const char *            scope,
                    const char *            localName,
                    float                   defaultVal) const
                                        throw(ConfigurationException);
    float lookupFloat(
                    const char *            scope,
                    const char *            localName) const
                                        throw(ConfigurationException);
    int lookupEnum(
                    const char *                scope,
                    const char *                localName,
                    const char *                typeName,
                    const EnumNameAndValue *    enumInfo,
                    int                         numEnums,
                    const char *                defaultVal) const
                                        throw(ConfigurationException);
    int lookupEnum(
                    const char *                scope,
                    const char *                localName,
                    const char *                typeName,
                    const EnumNameAndValue *    enumInfo,
                    int                         numEnums,
                    int                         defaultVal) const
                                        throw(ConfigurationException);
... continued on the next page
```

Figure 2.10 (continued): The `lookup<Type>()` operations

```
... continued from the previous page
    int lookupEnum(
                    const char *                scope,
                    const char *                localName,
                    const char *                typeName,
                    const EnumNameAndValue *    enumInfo,
                    int                         numEnums) const
                                        throw(ConfigurationException);

    bool lookupBoolean(
                    const char *            scope,
                    const char *            localName,
                    bool                    defaultVal) const
                                        throw(ConfigurationException);

    bool lookupBoolean(
                    const char *            scope,
                    const char *            localName) const
                                        throw(ConfigurationException);

    void lookupFloatWithUnits(
                    const char *            scope,
                    const char *            localName,
                    const char *            typeName,
                    const char **           allowedUnits,
                    int                     allowedUnitsSize,
                    float &                 floatResult,
                    const char *&           unitsResult) const
                                        throw(ConfigurationException);

    void lookupFloatWithUnits(
                    const char *            scope,
                    const char *            localName,
                    const char *            typeName,
                    const char **           allowedUnits,
                    int                     allowedUnitsSize,
                    float &                 floatResult,
                    const char *&           unitsResult,
                    float                   defaultFloat,
                    const char *            defaultUnits) const
                                        throw(ConfigurationException);
... continued on the next page
```

Figure 2.10 (continued): The lookup<Type>() operations

```
... continued from the previous page
    void lookupUnitsWithFloat(
                    const char *            scope,
                    const char *            localName,
                    const char *            typeName,
                    const char **           allowedUnits,
                    int                     allowedUnitsSize,
                    float &                 floatResult,
                    const char *&           unitsResult) const
                                        throw(ConfigurationException);
    void lookupUnitsWithFloat(
                    const char *            scope,
                    const char *            localName,
                    const char *            typeName,
                    const char **           allowedUnits,
                    int                     allowedUnitsSize,
                    float &                 floatResult,
                    const char *&           unitsResult,
                    float                   defaultFloat,
                    const char *            defaultUnits) const
                                        throw(ConfigurationException);
    void lookupIntWithUnits(
                    const char *            scope,
                    const char *            localName,
                    const char *            typeName,
                    const char **           allowedUnits,
                    int                     allowedUnitsSize,
                    int &                   intResult,
                    const char *&           unitsResult) const
                                        throw(ConfigurationException);
    void lookupIntWithUnits(
                    const char *            scope,
                    const char *            localName,
                    const char *            typeName,
                    const char **           allowedUnits,
                    int                     allowedUnitsSize,
                    int &                   intResult,
                    const char *&           unitsResult,
                    int                     defaultInt,
                    const char *            defaultUnits) const
                                        throw(ConfigurationException);
... continued on the next page
```

Figure 2.10 (continued): The lookup<Type>() operations

```
... continued from the previous page
    void lookupUnitsWithInt(
                    const char *            scope,
                    const char *            localName,
                    const char *            typeName,
                    const char **           allowedUnits,
                    int                     allowedUnitsSize,
                    int &                   intResult,
                    const char *&           unitsResult) const
                                        throw(ConfigurationException);
    void lookupUnitsWithInt(
                    const char *            scope,
                    const char *            localName,
                    const char *            typeName,
                    const char **           allowedUnits,
                    int                     allowedUnitsSize,
                    int &                   intResult,
                    const char *&           unitsResult,
                    int                     defaultInt,
                    const char *            defaultUnits) const
                                        throw(ConfigurationException);
    int lookupDurationMicroseconds(
                    const char *            scope,
                    const char *            localName,
                    int                     defaultVal) const
                                        throw(ConfigurationException);
    int lookupDurationMicroseconds(
                    const char *            scope,
                    const char *            localName) const
                                        throw(ConfigurationException);
    int lookupDurationMilliseconds(
                    const char *            scope,
                    const char *            localName,
                    int                     defaultVal) const
                                        throw(ConfigurationException);
    int lookupDurationMilliseconds(
                    const char *            scope,
                    const char *            localName) const
                                        throw(ConfigurationException);

... continued on the next page
```

Figure 2.10 (continued): The lookup<Type>() operations

```
... continued from the previous page
    int lookupDurationSeconds(
                    const char *            scope,
                    const char *            localName,
                    int                     defaultVal) const
                                        throw(ConfigurationException);
    int lookupDurationSeconds(
                    const char *            scope,
                    const char *            localName) const
                                        throw(ConfigurationException);
    int lookupMemorySizeBytes(
                    const char *            scope,
                    const char *            localName,
                    int                     defaultVal) const
                                        throw(ConfigurationException);
    int lookupMemorySizeBytes(
                    const char *            scope,
                    const char *            localName) const
                                        throw(ConfigurationException);
    int lookupMemorySizeKB(
                    const char *            scope,
                    const char *            localName,
                    int                     defaultVal) const
                                        throw(ConfigurationException);
    int lookupMemorySizeKB(
                    const char *            scope,
                    const char *            localName) const
                                        throw(ConfigurationException);
    int lookupMemorySizeMB(
                    const char *            scope,
                    const char *            localName,
                    int                     defaultVal) const
                                        throw(ConfigurationException);
    int lookupMemorySizeMB(
                    const char *            scope,
                    const char *            localName) const
                                        throw(ConfigurationException);
    void lookupScope(
                    const char *            scope,
                    const char *            localName) const
                                        throw(ConfigurationException);
};
```

2.6.2 Lookup Operations for Unit-based Types

Lookup operations that have `Units` in their name take, among other parameters, an array of strings (specifying the allowed units) and the size of that array. An example can be seen in Figure 2.11.

Figure 2.11: Example invocation of `lookupUnitsWithFloat`

```
float         amount;
const char *  currency;
const char *  currencies[] = {"£", "$", "€"};
cfg->lookupUnitsWithFloat(scope, "discount_price", "price",
                          currencies, 3, amount, currency);
```

The `typeName` parameter (`"price"` in the example) specifies the "type name" of correctly-formatted strings, and is used to construct an informative error message if an exception is thrown.

The output parameters, `amount` and `currency`, contain the quantity and units that were parsed from the value of the configuration variable.

2.7 The `type()` and `is<Type>()` Operations

Figure 2.12 shows the operations you can use to query type information.

The `type()` operation merges the `scope` and `localName` parameters to form the fully-scoped name of an entry in the `Configuration` object, and then returns the type of that entry. The return value of this operation will be one of the following:

Return value	Meaning
`Configuration::CFG_NO_VALUE`	The entry does not exist
`Configuration::CFG_STRING`	The entry is a string variable
`Configuration::CFG_LIST`	The entry is a list variable
`Configuration::CFG_SCOPE`	The entry is a scope

Operations with names of the form `is<Type>()` return `true` if the `str` parameter is of the specified type. For example:

```
cfg->isBoolean("true")                    → true
cfg->isBoolean("Fred")                    → false
cfg->isDurationSeconds("2.5 minutes")     → true
cfg->isDurationSeconds("100 milliseconds") → false
```

Figure 2.12: The type() and is<Type>() operations

```
struct EnumNameAndValue { const char * name;  int value; };
class Configuration {
public:
    enum Type {CFG_NO_VALUE       = 0, // bit masks
               CFG_STRING         = 1, // 0001
               CFG_LIST           = 2, // 0010
               CFG_SCOPE          = 4, // 0100
               CFG_VARIABLES      = 3, // 0011 = STRING | LIST
               CFG_SCOPE_AND_VARS = 7  // 0111 = STRING | LIST | SCOPE
    };
    Type type(const char * scope, const char * localName) const;
    bool isBoolean(const char * str) const;
    bool isInt(const char * str) const;
    bool isFloat(const char * str) const;
    bool isDurationMicroseconds(const char * str) const;
    bool isDurationMilliseconds(const char * str) const;
    bool isDurationSeconds(const char * str) const;
    bool isMemorySizeBytes(const char * str) const;
    bool isMemorySizeKB(const char * str) const;
    bool isMemorySizeMB(const char * str) const;
    bool isEnum(const char *                str,
                const EnumNameAndValue *    enumInfo,
                int                         numEnums) const;
    bool isFloatWithUnits(
                    const char *        str,
                    const char **       allowedUnits,
                    int                 allowedUnitsSize) const;
    bool isIntWithUnits(
                    const char *        str,
                    const char **       allowedUnits,
                    int                 allowedUnitsSize) const;
    bool isUnitsWithFloat(
                    const char *        str,
                    const char **       allowedUnits,
                    int                 allowedUnitsSize) const;
    bool isUnitsWithInt(
                    const char *        str,
                    const char **       allowedUnits,
                    int                 allowedUnitsSize) const;
    ...
};
```

The `isEnum()` operation takes three parameters: a string to be tested, an array of `EnumNameAndValue` structures and the size of that array. For example:

```
EnumNameAndValue colourInfo[] = {
    { "red",   0 },
    { "green", 1 },
    { "blue",  2 },
};
cfg->isEnum("red", colourInfo, 3) → true
cfg->isEnum("foo", colourInfo, 3) → false
```

The `is<Type>()` operations with `"Units"` in their name take three parameters: a string to be tested, an array of strings (specifying the allowed units) and the size of that array. For example:

```
const char * currencies[] = {"£", "$", "€"};
cfg->isUnitsWithFloat("£19.99", currencies, 3) → true
cfg->isUnitsWithFloat("foobar", currencies, 3) → false
```

2.8 The `stringTo<Type>()` Operations

Figure 2.13 lists operations that can convert a string value to another type.

An operation with a name of the form `stringTo<Type>()` converts a string into the specified type. If the conversion fails, then the operation throws an exception containing an informative error message. The error message will indicate that the problem arose with the variable identified by the fully-scoped name (obtained by merging the `scope` and `localName` parameters) in the `fileName()` configuration file.

As an example, consider a call to `stringToInt()` in which the `scope` parameter is `"foo"`, the `localName` parameter is `"my_list[3]"` and the `str` parameter is `"Hello, world"`. If the configuration file previously parsed was called `example.cfg`, then the message in the exception will be:

```
example.cfg: Non-integer value for 'foo.my_list[3]'
```

The intention is that developers will iterate over all the strings within a list and handcraft the `localName` parameter for each list element to reflect its position within the list: `"my_list[1]"`, `"my_list[2]"`, `"my_list[3]"` and so on. In this way, the `stringTo<Type>()` operations can produce informative exception messages if a data-type conversion fails. Note that although many programming languages, including C++, index arrays

Figure 2.13: The stringTo<Type>() operations

```
class Configuration {
public:
    bool stringToBoolean(
                    const char *            scope,
                    const char *            localName,
                    const char *            str) const
                                        throw(ConfigurationException);
    int stringToInt(
                    const char *            scope,
                    const char *            localName,
                    const char *            str) const
                                        throw(ConfigurationException);
    float stringToFloat(
                    const char *            scope,
                    const char *            localName,
                    const char *            str) const
                                        throw(ConfigurationException);

    int stringToDurationSeconds(
                    const char *            scope,
                    const char *            localName,
                    const char *            str) const
                                        throw(ConfigurationException);
    int stringToDurationMilliseconds(
                    const char *            scope,
                    const char *            localName,
                    const char *            str) const
                                        throw(ConfigurationException);
    int stringToDurationMicroseconds(
                    const char *            scope,
                    const char *            localName,
                    const char *            str) const
                                        throw(ConfigurationException);

    int stringToMemorySizeBytes(
                    const char *            scope,
                    const char *            localName,
                    const char *            str) const
                                        throw(ConfigurationException);
... continued on the next page
```

Figure 2.13 (continued): The stringTo<Type>() operations

```
... continued from the previous page
    int stringToMemorySizeKB(
                    const char *            scope,
                    const char *            localName,
                    const char *            str) const
                                        throw(ConfigurationException);
    int stringToMemorySizeMB(
                    const char *            scope,
                    const char *            localName,
                    const char *            str) const
                                        throw(ConfigurationException);

    int stringToEnum(
                    const char *                scope,
                    const char *                localName,
                    const char *                typeName,
                    const char *                str,
                    const EnumNameAndValue *    enumInfo,
                    int                         numEnums) const
                                        throw(ConfigurationException);

    void stringToFloatWithUnits(
                    const char *            scope,
                    const char *            localName,
                    const char *            typeName,
                    const char *            str,
                    const char **           allowedUnits,
                    int                     allowedUnitsSize,
                    float &                 floatResult,
                    const char *&           unitsResult) const
                                        throw(ConfigurationException);
    void stringToUnitsWithFloat(
                    const char *            scope,
                    const char *            localName,
                    const char *            typeName,
                    const char *            str,
                    const char **           allowedUnits,
                    int                     allowedUnitsSize,
                    float &                 floatResult,
                    const char *&           unitsResult) const
                                        throw(ConfigurationException);
... continued on the next page
```

Figure 2.13 (continued): The `stringTo<Type>()` operations

```
... continued from the previous page
    void stringToIntWithUnits(
                    const char *            scope,
                    const char *            localName,
                    const char *            typeName,
                    const char *            str,
                    const char **           allowedUnits,
                    int                     allowedUnitsSize,
                    int &                   intResult,
                    const char *&           unitsResult) const
                                        throw(ConfigurationException);
    void stringToUnitsWithInt(
                    const char *            scope,
                    const char *            localName,
                    const char *            typeName,
                    const char *            str,
                    const char **           allowedUnits,
                    int                     allowedUnitsSize,
                    int &                   intResult,
                    const char *&           unitsResult) const
                                        throw(ConfigurationException);
    ...
};
```

starting from 0, you should format the `localName` parameter so the index starts at 1. This is to be consistent with the error messages produced by the `SchemaValidator` class.

2.9 The List Operations

Figure 2.14 shows the operations for listing the names of entries within a scope. There are two list-type operations: `listFullyScopedNames()` and `listLocallyScopedNames()`. However, there are three overloaded versions of each operation, thus making for six variants in total.

The list operations merge the `scope` and `localName` parameters to form the fully-scoped name of a scope, and populate the output `names` parameter with a sorted list of the names of entries in that scope. The boolean `recursive` parameter specifies whether the list operation should recurse into nested sub-scopes. The `typeMask` parameter is a bit mask that specifies which types of entries should be listed. For example, speci-

Figure 2.14: The list operations

```
class Configuration {
public:
    enum Type {CFG_NO_VALUE       = 0, // bit masks
               CFG_STRING         = 1, // 0001
               CFG_LIST           = 2, // 0010
               CFG_SCOPE          = 4, // 0100
               CFG_VARIABLES      = 3, // 0011 = STRING | LIST
               CFG_SCOPE_AND_VARS = 7  // 0111 = STRING | LIST | SCOPE
    };

    void listFullyScopedNames(
                    const char *            scope,
                    const char *            localName,
                    Type                    typeMask,
                    bool                    recursive,
                    StringVector &          names) const
                                        throw(ConfigurationException);
    void listFullyScopedNames(
                    const char *            scope,
                    const char *            localName,
                    Type                    typeMask,
                    bool                    recursive,
                    const char *            filterPattern,
                    StringVector &          names) const
                                        throw(ConfigurationException);
    void listFullyScopedNames(
                    const char *            scope,
                    const char *            localName,
                    Type                    typeMask,
                    bool                    recursive,
                    const StringVector &    filterPatterns,
                    StringVector &          names) const
                                        throw(ConfigurationException);
    void listLocallyScopedNames(
                    const char *            scope,
                    const char *            localName,
                    Type                    typeMask,
                    bool                    recursive,
                    StringVector &          names) const
                                        throw(ConfigurationException);
... continued on the next page
```

Figure 2.14 (continued): The list operations

```
... continued from the previous page
    void listLocallyScopedNames(
                    const char *            scope,
                    const char *            localName,
                    Type                    typeMask,
                    bool                    recursive,
                    const char *            filterPattern,
                    StringVector &          names) const
                                        throw(ConfigurationException);
    void listLocallyScopedNames(
                    const char *            scope,
                    const char *            localName,
                    Type                    typeMask,
                    bool                    recursive,
                    const StringVector &    filterPatterns,
                    StringVector &          names) const
                                        throw(ConfigurationException);
    ...
};
```

fying `CFG_VARIABLES` will list only the names of variables, while `CFG_SCOPE` will list only the names of scopes.

By default, a list operation lists the names of *all* the specified entries. However, if one or more *filter patterns* are specified, then the list operation will use the `patternMatch()` operation (Section 2.3 on page 11) to compare each name against the specified patterns, and only names that match at least one filter pattern will be included in the list results.

As an example of the list functions, consider the configuration file shown below:

```
foo {
    timeout = "2 minutes";
    log {
        level = "2";
        file = "/tmp/foo.log";
    };
}
```

The following invocation of `listFullyScopedNames()`:

```
cfg->listFullyScopedNames("foo", "", Configuration::CFG_SCOPE_AND_VARS,
                          true, names);
```

results in `names` containing the following strings:

```
"foo.log"
"foo.log.level"
"foo.log.file"
"foo.timeout"
```

If the same parameters are passed to `listLocallyScopedNames()`, then `names` will contain similar strings, but each string will be missing the `"foo."` prefix.

If you intend to make use of filter patterns, then you should note that filter patterns are matched against the strings that are produced by the list operation. For example, the filter pattern `"time*"` matches against `"timeout"`, which is produced by `listLocallyScopedNames()` in the previous example, but it does *not* match against `"foo.timeout"`, which is produced by `listFullyScopedNames()`. Because of this, you should use `mergeNames()` (Section 2.3 on page 11) to prefix filter patterns with the name of the scope being listed when using `listFullyScopedNames()`. This is illustrated in the following example:

```
StringBuffer    filterPattern;
Configuration::mergeNames(scope, "time*", filterPattern);
cfg->listFullyScopedNames(scope, "", Configuration::CFG_SCOPE_AND_VARS,
                          true, filterPattern, names);
```

The list operations call `unexpandUid()`—discussed in Section 2.12 on page 36—on a name before comparing it against filter patterns. Because of this, filter patterns can work with names that have an `"uid-"` prefix. For example, the code below obtains a list of the names of all `uid-recipe` scopes:

```
StringBuffer    filterPattern;
Configuration::mergeNames(scope, "uid-recipe", filterPattern);
cfg->listFullyScopedNames(scope, "", Configuration::CFG_SCOPE,
                          true, filterPattern, names);
```

2.10 Operations for Fallback Configuration

The operations for getting and setting fallback configuration are shown in Figure 2.15.

The one-parameter version of `setFallbackConfiguration()` requires you to create and populate the fallback configuration object yourself. The three-parameter version creates an initially empty fallback configuration object and then populates it by calling `parse()` with the specified parameters.

Figure 2.15: Operations for Fallback Configuration

```
class Configuration {
public:
    void setFallbackConfiguration(Configuration * cfg);
    void setFallbackConfiguration(
                    Configuration::SourceType sourceType,
                    const char *              source,
                    const char *              sourceDescription = "")
                                        throw(ConfigurationException);
    const Configuration * getFallbackConfiguration();
    ...
};
```

A "main" configuration object takes ownership of its fallback configuration object. Because of this, when you invoke `destroy()` on the "main" configuration object, its fallback configuration object is also destroyed.

2.11 Operations for Security Configuration

As explained in the *Config4* Security* chapter of the *Config4* Getting Started Guide*, Config4* has a built-in security policy that is applied to all `Configuration` objects by default. You can query the current security policy of a `Configuration` object by calling `getSecurityConfiguration()`, which is shown in Figure 2.16.

You can change the security policy of an individual `Configuration` object by calling `setSecurityConfiguration()`. This operation is overloaded. The first variant shown in Figure 2.16 enables you to use an existing `Configuration` object as a security policy. If `takeOwnership` is `true`, then the target `Configuration` object will "take ownership" of the security policy object, which means the security policy object will be destroyed when the target `Configuration` object is destroyed.

The second variant of `setSecurityConfiguration` enables you to specify a file or `"exec#..."` that should be parsed to obtain a security policy. The target `Configuration` object will "take ownership" of the security policy object.

The security policy is specified by the combination of three variables (`allow_patterns`, `deny_patterns` and `trusted_directories`) in the specified `scope` of the security policy object. See the *Config4* Security* chapter of the *Config4* Getting Started Guide* for details.

Figure 2.16: Operations for Security Configuration

```
class Configuration {
public:
    void setSecurityConfiguration(
                    Configuration *        cfg,
                    bool                   takeOwnership,
                    const char *           scope = "")
                                        throw(ConfigurationException);
    void setSecurityConfiguration(
                    const char *           cfgInput,
                    const char *           scope = "")
                                        throw(ConfigurationException);
    void getSecurityConfiguration(
                    const Configuration *& cfg,
                    const char *&          scope);
    ...
};
```

2.12 Operations for the "uid-" Prefix

An identifier that has an "uid-" prefix has both an *expanded* and *unexpanded* form. For example, uid-000000042-recipe is an identifier in its expanded form, while uid-recipe is its unexpanded counterpart. Figure 2.17 lists the operations that Config4Cpp provides for manipulating expanded and unexpanded uid identifiers.

Figure 2.17: Operations for the "uid-" prefix

```
class Configuration {
public:
    void expandUid(StringBuffer & spelling)
                                        throw(ConfigurationException);
    const char * unexpandUid(
                    const char *           spelling,
                    StringBuffer &         buf) const;
    bool uidEquals(const char * s1, const char * s2) const;
    ...
};
```

Each Configuration object keeps an internal counter that starts at 0 and is incremented every time expandUid() encounters an "uid-" prefix. The current value of that counter is used by expandUid() to replace an

identifier with its expanded form.

If you populate a `Configuration` object by calling `parse()`, then you are unlikely to need to call `expandUid()`, because the parser invokes that operation automatically whenever it encounters an identifier with an `"uid-"` prefix.

However, if you populate a `Configuration` object by invoking the insertion operations discussed in Section 2.5 on page 17, then it is your responsibility to expand identifiers before invoking the insertion operations.

The `uidEquals()` operation calls `unexpandUid()` for both of its parameters, and tests the resulting names for equality. For example:

```
cfg->uidEquals("uid-000000042-recipe", "uid-recipe")   → true
cfg->uidEquals("uid-000000042-recipe", "uid-employee") → false
```

2.13 The `dump()` Operation

When Config4Cpp parses a configuration file, it stores information about scopes and *name=value* pairs in hash tables. Config4Cpp provides a `dump()` operation, shown in Figure 2.18, that converts information in the hash tables into the syntax of a Config4* file; this result is stored in the `buf` parameter.

Figure 2.18: The `dump()` operation

```
class Configuration {
public:
    void dump(StringBuffer &          buf,
              bool                    wantExpandedUidNames,
              const char *            scope,
              const char *            localName) const
                                        throw(ConfigurationException);
    void dump(StringBuffer & buf, bool wantExpandedUidNames) const;
    ...
};
```

The `dump()` operation is overloaded. The version that takes `scope` and `localName` parameters merges those parameters to form the fully-scoped name of an entry, and then provides a dump of that entry. This version of `dump()` will throw an exception if the fully-scoped name is of an non-existent entry.

The other version of the operation dumps the entire `Configuration` object.

Both versions of the `dump()` operation take a boolean parameter, `wantExpandedUidNames`, that specifies whether entries that have an `"uid-"` prefix should have their names dumped in expanded or unexpanded form.

Chapter 3

The `SchemaValidator` and `SchemaType` Classes

3.1 The `SchemaValidator` Class

The public and protected operations of the `SchemaValidator` class are shown in Figure 3.1. First, I will discuss the public operations, and then the protected one.

3.1.1 Public Operations

The overloaded `wantDiagnostics()` operation enables you to get and set a boolean property, the default value of which is `false`. If you set this to `true`, then detailed diagnostic messages will be printed to standard output during calls to `parseSchema()` and `validate()`. These diagnostic messages may be useful when debugging a schema.

The `parseSchema()` operation parses a schema definition and stores it in an efficient internal format. The schema can be specified as an array of strings plus the size of that array, or as a null-terminated array of strings. The `parseSchema()` operation will throw an exception if the parser encounters a problem, such as a syntax error, when parsing the schema.

After you have created a `SchemaValidator` object and used it to parse a schema, you can then call `validate()` to validate (a scope within) a configuration file. If you want, you can call `validate()` repeatedly, perhaps to validate multiple configuration files. The `validate()` operation

Figure 3.1: The SchemaValidator class

```
// Access with #include <config4cpp/SchemaValidator.h>

class Configuration {
public:
    enum Type {CFG_NO_VALUE       = 0, // bit masks
               CFG_STRING         = 1, // 0001
               CFG_LIST           = 2, // 0010
               CFG_SCOPE          = 4, // 0100
               CFG_VARIABLES      = 3, // 0011 = STRING | LIST
               CFG_SCOPE_AND_VARS = 7  // 0111 = STRING | LIST | SCOPE
    };
    ...
};
class SchemaValidator {
public:
    enum ForceMode {DO_NOT_FORCE, FORCE_OPTIONAL, FORCE_REQUIRED};

    SchemaValidator();
    void wantDiagnostics(bool value);
    bool wantDiagnostics();
    void parseSchema(const char ** schema, int schemaSize)
                                        throw(ConfigurationException);
    void parseSchema(const char ** nullTerminatedSchema)
                                        throw(ConfigurationException);
    void validate(
        const Configuration *    cfg,
        const char *             scope,
        const char *             localName,
        bool                     recurseIntoSubscopes,
        Configuration::Type      typeMask,
        ForceMode                forceMode = DO_NOT_FORCE) const
                                        throw(ConfigurationException);
    void validate(
        const Configuration *    cfg,
        const char *             scope,
        const char *             localName,
        ForceMode                forceMode = DO_NOT_FORCE) const
                                        throw(ConfigurationException);
protected:
    void registerType(SchemaType * type) throw(ConfigurationException);
};
```

merges the `scope` and `localName` parameters to form the fully-scoped name of the scope (within the `cfg` object) to be validated.

The `recurseIntoSubscopes` parameter specifies whether `validate()` should validate only entries in the scope, or recurse down into sub-scopes to validate their entries too.

The `typeMask` parameter is a bit mask that specifies which types of entries should be validated. For example, `CFG_VARIABLES` specifies that variables (but not scopes) should be validated.

By default, `validate()` respects use of the `@optional` and `@required` keywords in the schema. However, if you specify `FORCE_OPTIONAL` for the `forceMode` parameter, then `validate()` will act as if all identifiers in the schema have the `@optional` keyword. Conversely, `FORCE_REQUIRED` makes `validate()` act as if all identifiers without an `"uid-"` prefix in the schema have the `@required` keyword.

There are two versions of the `validate()` operation. The version with four parameters uses `true` for the `recurseIntoSubscopes` parameter and `CFG_SCOPE_AND_VARS` for the `typeMask` parameter.

3.1.2 Using `registerType()` in a Subclass

Later, in Section 3.2, I will explain how you can implement new schema types. If you implement new schema types, then you will need to write a subclass of `SchemaValidator` to register those new schema types. Figure 3.2 illustrates how to do this.

Registration of new schema types is trivial: the constructor of the subclass simply calls `registerType()` to register one instance of each of the new schema types.

Once you have implemented the `ExtendedSchemaValidator` class to register new schema types, your applications need only create an instance of `ExtendedSchemaValidator` (instead of `SchemaValidator`) to be able to make use of those new schema types.

3.2 The `SchemaType` Class

The `SchemaValidator` class perform very little of the validation work itself. Instead, it delegates most of this work to other classes, each of which is a subclass of `SchemaType` (shown in Figure 3.3). There is a separate subclass of `SchemaType` for each schema type. For example, the Config4Cpp library contains `SchemaTypeBoolean`, which implements the

Figure 3.2: A subclass of SchemaValidator

```
#include <config4cpp/SchemaValidator.h>
using config4cpp::SchemaValidator;

class SchemaTypeDate { ... }; // Define a new schema type
class SchemaTypeHex { ... };  // Define a new schema type

class ExtendedSchemaValidator : public SchemaValidator
{
public:
    ExtendedSchemaValidator()
    {
        registerType(new SchemaTypeDate());
        registerType(new SchemaTypeHex());
    }
};
```

boolean schema type, SchemaTypeInt, which implements the int schema type, and so on.

3.2.1 Constructor and Public Accessors

When the constructor of a subclass of SchemaType calls its parent constructor, the parameters specify the name of the schema type, the name of the class that implements it, and the configuration entry's type, which is one of: CFG_STRING, CFG_LIST or CFG_SCOPE. You can see an example of this in Figure 3.4.

Parameter values passed to the parent constructor are made available via the typeName(), className() and cfgType() operations shown in Figure 3.3.

The SchemaValidator class invokes registerType() to register an instance of each of the predefined schema types and, as previously shown in Figure 3.2, a subclass of SchemaValidator can invoke registerType() to register instances of additional schema types.

3.2.2 The checkRule() Operation

The SchemaValidator class invokes the checkRule() operation of an object representing a schema type when that type is encountered in a schema rule. I will illustrate this through the schema shown in Figure 3.5.

Figure 3.3: The SchemaType class

```
// Access with #include <config4cpp/SchemaValidator.h>
// or          #include <config4cpp/SchemaType.h>

class SchemaType {
public:
    SchemaType(
        const char *               typeName,
        const char *               className,
        Configuration::Type        cfgType);
    virtual ~SchemaType();

    const char * typeName()  const;
    const char * className() const;
    Configuration::Type cfgType()   const;
protected:
    virtual void checkRule(
        const SchemaValidator *        sv,
        const Configuration *          cfg,
        const char *                   typeName,
        const StringVector &           typeArgs,
        const char *                   rule) const
                                   throw(ConfigurationException) = 0;
    virtual void validate(
        const SchemaValidator *        sv,
        const Configuration *          cfg,
        const char *                   scope,
        const char *                   name,
        const char *                   typeName,
        const char *                   origTypeName,
        const StringVector &           typeArgs,
        int                            indentLevel) const
                                   throw(ConfigurationException);
    virtual bool isA(
        const SchemaValidator *        sv,
        const Configuration *          cfg,
        const char *                   value,
        const char *                   typeName,
        const StringVector &           typeArgs,
        int                            indentLevel,
        StringBuffer &                 errSuffix) const;
... continued on the next page
```

Figure 3.3 (continued): The SchemaType class

```
... continued from the previous page
    SchemaType * findType(
        const SchemaValidator *         sv,
        const char *                    name) const;
    void callValidate(
        const SchemaType *              target,
        const SchemaValidator *         sv,
        const Configuration *           cfg,
        const char *                    scope,
        const char *                    name,
        const char *                    typeName,
        const char *                    origTypeName,
        const StringVector &            typeArgs,
        int                             indentLevel) const
                                    throw(ConfigurationException);
    bool callIsA(
        const SchemaType *              target,
        const SchemaValidator *         sv,
        const Configuration *           cfg,
        const char *                    value,
        const char *                    typeName,
        const StringVector &            typeArgs,
        int                             indentLevel,
        StringBuffer &                  errSuffix) const;
};
```

Figure 3.4: Example constructor of a subclass of SchemaType

```
SchemaTypeInt::SchemaTypeInt()
    : SchemaType("int", "config4cpp::SchemaTypeInt",
                 Configuration::CFG_STRING)
{
    // Nothing else to do in the constructor
}
```

When parsing the first line of the schema, SchemaValidator invokes checkRule() on the object representing the durationMilliseconds schema type. When parsing the next line in the schema, the SchemaValidator invokes checkRule() on the object representing the list schema type, and so on.

Among the parameters passed to checkRule() is typeArgs (of type

Figure 3.5: Example schema

```
1  const char * schema[] = {
2      "timeout = durationMilliseconds",
3      "fonts = list[string]",
4      "background_colour = enum[grey, white, yellow]",
5      "log = scope",
6      "log.dir = string",
7      "@typedef logLevel = int[0,3]",
8      "log.level = logLevel"
9  };
```

`StringVector`), which contains the arguments, if any, for the type. This parameter will be empty for the rules in lines 2, 5 and 6 of Figure 3.5. For the rule in line 3, `typeArgs` will contain one string (`"string"`); and for the rule in line 4, it will contain three strings (`"grey"`, `"white"` and `"yellow"`). You might think that `typeArgs` should be empty for the rule in line 8. However, the `logLevel` type used in line 8 was defined in line 7 to be `int[0,3]`. Because of this, when `checkRule()` is called for the rule in line 8, `typeArgs` will contain two strings (`"0"` and `"3"`).

The implementation of `checkRule()` must determine whether the strings in `typeArgs` are valid, and throw an exception containing a descriptive error message if not. For example:

- The implementation of `SchemaTypeInt::checkRule()` throws an exception unless: (1) there are *zero* strings in `typeArgs`; or (2) there are *two* strings in `typeArgs`, both strings can be parsed as integers, and the first integer is smaller than or equal to the second integer.

- The implementation of `SchemaTypeList::checkRule()` throws an exception unless there is exactly one string in `typeArgs`, and that string is the name of a schema type whose configuration entry's type is `CFG_STRING`. This `checkRule()` operation invokes `findType()` to search for the specified schema type; `findType()` returns a nil pointer if the type does not exist.

Deciding whether the `typeArgs` parameter contains acceptable strings is the primary purpose of `checkRule()`. Most of the other parameters are provided to help `checkRule()` make that decision and to format an informative exception message if necessary.

One of the demonstration applications provided with Config4Cpp is called `extended-schema-validator`. That demo contains a class called

SchemaTypeHex that implements a hex (hexadecimal integer) schema type. That class's implementation of checkRule() is shown in Figure 3.6. A **bold** font indicates how the operation makes use of parameters.

Figure 3.6: Implementation of SchemaTypeHex::checkRule()

```
void SchemaTypeHex::checkRule(
    const SchemaValidator *  sv,
    const Configuration *    cfg,
    const char *             typeName,
    const StringVector &     typeArgs,
    const char *             rule) const throw(ConfigurationException)
{
    StringBuffer             msg;
    int                      len;
    int                      maxDigits;

    len = typeArgs.length();
    if (len == 0) {
        return;
    } else if (len > 1) {
        msg << "schema error: the '" << typeName << "' type should "
            << "take either no arguments or 1 argument (denoting "
            << "max-digits) in rule '" << rule << "'";
        throw ConfigurationException(msg.c_str());
    }
    try {
        maxDigits = cfg->stringToInt("", "", typeArgs[0]);
    } catch(const ConfigurationException & ex) {
        msg << "schema error: non-integer value for the 'max-digits' "
            << "argument in rule '" << rule << "'";
        throw ConfigurationException(msg.c_str());
    }
    if (maxDigits < 1) {
        msg << "schema error: the 'max-digits' argument must be 1 or "
            << "greater in rule '" << rule << "'";
        throw ConfigurationException(msg.c_str());
    }
}
```

The only parameter *not* used in the body of the operation is sv, which is of type SchemaValidator. That parameter is used by the checkRule() operation in the list, table and tuple types when invoking findType()

to determine if items in `typeArgs` are names of types.

3.2.3 The `isA()` and `validate()` Operations

Subclasses of `SchemaType` should implement the `isA()` and `validate()` operations. However, the default implementation of `isA()` is suitable for list-based types, and the default implementation of `validate()` is suitable for string-based types. Because of this, a subclass of `SchemaType` needs to implement only one of these two operations.

3.2.3.1 String-based Types: `isA()`

If you are providing schema support for a string-based type, then you must implement the `isA()` operation. Among the parameters passed to this operation is a `string` called `value`; the `isA()` operation should return `true` if `value` can be parsed as the schema type. For example, the `SchemaTypeInt::isA()` operation returns `true` for `"42"` and returns `false` for `"hello, world"`.

If `isA()` returns `false`, then the operation can optionally set the `errSuffix` parameter (which is of type `StringBuffer`) to be a descriptive message that explains *why* the string is not suitable. This message will be appended to an exception message.

Figure 3.7 illustrates how `isA()` might be implemented for a schema type that denotes hexadecimal integers. A **bold** font indicates how the operation makes use of parameters. This implementation of `isA()` contains two straightforward checks. First, it checks whether `value` consists of hexadecimal digits. Second, if `typeArgs` specifies a maximum number of digits, then `isA()` checks if this limit has been exceeded.

3.2.3.2 List-based Types: `validate()`

Config4* has three built-in, list-based schema types: `list`, `tuple` and `table`. Each of these schema types takes arguments, for example:

```
const char * schema[] = {
    "@typedef money = units_with_float[\"£\", \"$\", \"€\"]",
    "fonts      = list[string]",
    "point      = tuple[float,x, float,y]",
    "price_list = table[string,product, money,price]"
};
```

Figure 3.7: Implementation of `isA()` for a `hex` type

```
bool SchemaTypeHex::isA(
    const SchemaValidator *  sv,
    const Configuration *    cfg,
    const char *             value,
    const char *             typeName,
    const StringVector &     typeArgs,
    int                      indentLevel,
    StringBuffer &           errSuffix) const
{
    if (!isHex(value)) {
        errSuffix << "the value is not a hexadecimal number";
        return false;
    }
    if (typeArgs.length() == 1) {
        //--------
        // Check if there are too many hex digits in the value
        //--------
        int maxDigits = cfg->stringToInt("", "", typeArgs[0]);
        if (strlen(value) > maxDigits) {
            errSuffix << "the value must not contain more than "
                      << maxDigits << " digits";
            return false;
        }
    }
    return true;
}

bool SchemaTypeHex::isHex(const char * str)
{ ... } // implementation will be shown later in this chapter
```

Each of those list-based schema types implements `validate()` in a similar way, so I will discuss only the implementation for the `table` schema type, using the definition of `price_list` in the above example.

- A call of `cfg->lookupList(scope, name, ...)` is made to retrieve the value of the list variable from the configuration object.

- The `typeArgs` parameter contains all the arguments to the schema type (`"string"`, `"product"`, `"money"` and `"price"` for the `price_list` variable in the example). Those pairs of strings define the types and names of columns within the table. The `validate()` operation

checks that the length of the list is a multiple of the number of columns in the table's definition.

- Finally, `validate()` iterates over all the items in the list. For each item, `validate()` calls `findType()` for the item's column type (obtained from `typeArgs`) to retrieve the item's schema type; it invokes the `isA()` operation of that type, and throws an exception if `isA()` returns `false`.

  The invocation of `isA()` is not made directly. Rather, it is made indirectly by invoking `callIsA()`, which is shown in Figure 3.3 on page 43. Doing this ensures that diagnostic messages can be printed if the `SchemaValidator` was created with `true` specified for the `wantDiagnostics` constructor parameter.

If you want to implement schema support for a list-based type, then you should implement the `validate()` operation in a manner similar to that described above. I recommend that you examine the source code of the `SchemaTypeList`, `SchemaTypeTable` or `SchemaTypeTuple` class for concrete details.

3.3 Adding Utility Operations to a Schema Type

The infrastructure within Config4Cpp to support a built-in data type is split over three classes:

- The `SchemaType<Type>` class implements the schema validation infrastructure.
- The `SchemaValidator` class calls `registerType()` to register each schema type.
- The `Configuration` class provides operations with names of the form `lookup<Type>()`, `is<Type>()` and `stringTo<Type>()`.

In this chapter, I have explained how you can provide schema validation support for a new type by writing a `SchemaType<Type>` class and registering it in a subclass of `SchemaValidator`. However, I have not yet explained how you can write a subclass of `Configuration` to implement the `lookup<Type>()`, `is<Type>()` and `stringTo<Type>()` operations.

The `Configuration` class is an abstract base class, and its static `create()` operation creates an instance of a hidden, concrete subclass. This enforces a separation between the public API and the implementation details of Config4*. Most of the time, this separation is beneficial. However, it has a drawback: you cannot write a subclass of `Configuration` to add additional operations, such as `lookup<Type>()`, `is<Type>()` and `stringTo<Type>()`.

A good way to workaround this drawback is to define the desired functionality as `static` operations in the `SchemaType<Type>` class. For example, if you are writing a class called `SchemaTypeHex` (for hexadecimal integers), then you can implement `lookupHex()`, `isHex()`, and `stringToHex()` as `static` operations in the `SchemaTypeHex` class. This is illustrated in Figure 3.8.

With this technique, application code can call `cfg->lookup<Type>()` for built-in types, but must call `SchemaType<Type>::lookup<Type>()` for other types. For example:

```
try {
    logFile = cfg->lookupString(scope, "log.file");
    timeout = cfg->lookupDurationMilliseconds(scope, "idle_timeout");
    addr    = SchemaTypeHex::lookupHex(cfg, scope, "base_address");
} catch(const ConfigurationException & ex) {
    cerr << ex.c_str() << endl;
}
```

Figure 3.8: Utility operations in the SchemaTypeHex class

```
class SchemaTypeHex : public SchemaType
{
public:
    SchemaTypeHex()
        : SchemaType("hex", "SchemaTypeHex", Configuration::CFG_STRING)
    { }
    virtual ~SchemaTypeHex()

    static bool isHex(const char * str)
    {
        int                   i;
        for (i = 0; str[i] != '\0'; i++) {
            if (!isxdigit(str[i])) { return false; }
        }
        return i > 0;
    }

    static int lookupHex(
        const Configuration * cfg,
        const char *          scope,
        const char *          localName) throw(ConfigurationException)
    {
        const char * str = cfg->lookupString(scope, localName);
        return stringToHex(cfg, scope, localName, str);
    }

    static int lookupHex(
        const Configuration * cfg,
        const char *          scope,
        const char *          localName,
        int                   defaultVal) throw(ConfigurationException)
    {
        if (cfg->type(scope, localName)
            == Configuration::CFG_NO_VALUE)
        {
            return defaultVal;
        }
        const char * str = cfg->lookupString(scope, localName);
        return stringToHex(cfg, scope, localName, str);
    }
... continued on the next page
```

Figure 3.8 (continued): Utility operations in the SchemaTypeHex class

```
... continued from the previous page
    static int stringToHex(
        const Configuration * cfg,
        const char *          scope,
        const char *          localName,
        const char *          str,
        const char *          typeName) throw(ConfigurationException)
    {
        unsigned int          value;
        StringBuffer          msg;
        StringBuffer          fullyScopedName;

        int status = sscanf(str, "%x", &value);
        if (status != 1) {
            cfg->mergeNames(scope, localName, fullyScopedName);
            msg << cfg->fileName() << ": bad " << typeName
                << " value ('" << str << "') specified for '"
                << fullyScopedName;
            throw ConfigurationException(msg.c_str());
        }
        return (int)value;
    }
protected:
    ... // checkRule() and isA() were shown earlier in this chapter
}
```