# The Ringing Class Library

Martin Bright
Mark Banner
Richard Smith

Version 0.3.0. Last updated 11 May 2005.

This document was produced using GNU Texinfo. On-line versions in both HTML and GNU Info formats are also available.

# Table of Contents

**1 Introduction** . . . . . . . . . . 1

**2 Installing and using the library** . . . . . . . . . . 2
2.1 Licensing . . . . . . . . . . 2
2.2 The Standard Template Library . . . . . . . . . . 2
2.3 Unix-like systems . . . . . . . . . . 3
2.4 Microsoft Windows . . . . . . . . . . 3
2.5 Testing the library . . . . . . . . . . 4
2.6 Using the library in your programs . . . . . . . . . . 4

**3 Row and change classes** . . . . . . . . . . 5
3.1 Row operations . . . . . . . . . . 5
3.2 Row properties . . . . . . . . . . 5
3.3 Changes . . . . . . . . . . 6
3.4 The `bell` class . . . . . . . . . . 6
3.4.1 Derivation . . . . . . . . . . 6
3.4.2 Constructors . . . . . . . . . . 6
3.4.3 Operators . . . . . . . . . . 6
3.4.4 Other functions . . . . . . . . . . 6
3.5 The `row` class . . . . . . . . . . 7
3.5.1 Derivation . . . . . . . . . . 7
3.5.2 Constructors . . . . . . . . . . 7
3.5.3 Operators . . . . . . . . . . 7
3.5.4 Other functions . . . . . . . . . . 8
3.5.5 `static` functions . . . . . . . . . . 9
3.6 The `change` class . . . . . . . . . . 9
3.6.1 Derivation . . . . . . . . . . 9
3.6.2 Constructors . . . . . . . . . . 9
3.6.3 Operators . . . . . . . . . . 10
3.6.4 Other functions . . . . . . . . . . 10
3.7 The `permute` function . . . . . . . . . . 11
3.8 The `row_block` class . . . . . . . . . . 12
3.8.1 Derivation . . . . . . . . . . 12
3.8.2 Constructors . . . . . . . . . . 12
3.8.3 Other functions . . . . . . . . . . 12
3.9 Other functions in ‘`row.h`’ . . . . . . . . . . 12

**4 Method classes** . . . . . . . . . . 13
4.1 The `method` class . . . . . . . . . . 13
4.1.1 Derivation . . . . . . . . . . 13
4.1.2 Constructors . . . . . . . . . . 13
4.1.3 Other functions . . . . . . . . . . 13
4.1.4 `static` functions . . . . . . . . . . 15
4.1.5 `static` variables . . . . . . . . . . 15

**5 Method libraries** . . . . . 16

5.1 Using libraries . . . . . 16

5.1.1 Initialisation . . . . . 16

5.1.2 The `library` class . . . . . 16

5.1.2.1 Derivation . . . . . 16

5.1.2.2 Constructors . . . . . 17

5.1.2.3 Iteration functions . . . . . 17

5.1.2.4 Other functions . . . . . 17

5.1.3 The `library_entry` class . . . . . 18

5.1.4 Facets . . . . . 18

5.1.4.1 Built-in facets . . . . . 19

5.2 Built-in library types . . . . . 19

5.2.1 XML libraries . . . . . 19

5.2.1.1 The `xmllib` class . . . . . 20

5.2.1.2 XML library facets . . . . . 20

5.2.2 Central Council libraries . . . . . 20

5.2.2.1 The `cclib` class . . . . . 20

5.2.2.2 Central Council library facets . . . . . 21

5.2.3 MicroSIRIL libraries . . . . . 21

5.2.3.1 The `mslib` class . . . . . 21

5.2.3.2 MicroSIRIL library facets . . . . . 21

5.3 Library framework . . . . . 21

5.3.1 The `library_base` class . . . . . 22

5.3.2 The `library_base::const_iterator` class . . . . . 23

5.3.3 `library` internals . . . . . 23

5.3.4 The `library_entry` internals . . . . . 24

5.3.5 The `library_entry::impl` class . . . . . 25

5.4 Implementing new library classes . . . . . 26

5.4.1 Supporting auto-detection . . . . . 26

5.4.2 Alternative creation mechanisms . . . . . 26

5.4.3 Subclassing library_base . . . . . 27

5.4.4 Subclassing library_entry impl . . . . . 27

**6 Method and Touch Proving** . . . . . 29

6.1 The `proof` class . . . . . 29

6.1.1 Derivation . . . . . 29

6.1.2 Constructors . . . . . 29

6.1.3 Functions . . . . . 30

6.1.4 Operators . . . . . 30

6.1.5 Output . . . . . 30

6.1.6 How falseness is stored . . . . . 31

6.1.7 How it works . . . . . 31

6.1.8 Single Extent . . . . . 31

6.1.9 Multiple Extents . . . . . 31

**7 Analysis of Music . . . . . . . . . . 32**
7.1 Stroke Description . . . . . . . . . . 32
7.2 Music Regular Expressions . . . . . . . . . . 32
7.2.1 Music Regular Expression Language . . . . . . . . . . 32
7.2.2 Music Details class . . . . . . . . . . 33
7.3 Using the Music Class . . . . . . . . . . 34
7.4 Music Class definition . . . . . . . . . . 34
7.4.1 Derivation . . . . . . . . . . 34
7.4.2 Constructors . . . . . . . . . . 35
7.4.3 Type Definitions . . . . . . . . . . 35
7.4.4 Functions . . . . . . . . . . 35
7.5 Music Node Class definition . . . . . . . . . . 36
7.5.1 Full Description . . . . . . . . . . 36
7.5.2 Derivation . . . . . . . . . . 37
7.5.3 Constructors and Destructors . . . . . . . . . . 37
7.5.4 Type Definitions . . . . . . . . . . 38
7.5.5 Functions . . . . . . . . . . 38
7.5.6 Exceptions . . . . . . . . . . 39

**8 Producing printed output . . . . . . . . . . 40**
8.1 Auxiliary classes . . . . . . . . . . 40
8.1.1 The `dimension` class . . . . . . . . . . 40
8.1.2 The `colour` class . . . . . . . . . . 41
8.1.3 The `text_style` class . . . . . . . . . . 41
8.2 The `printpage` class . . . . . . . . . . 42
8.3 The `printrow` class . . . . . . . . . . 42
8.3.1 Using the `printrow` class . . . . . . . . . . 42
8.3.2 Row printing options . . . . . . . . . . 43
8.4 Printing whole methods . . . . . . . . . . 45
8.4.1 Method printing parameters . . . . . . . . . . 45
8.4.2 Using the `printmethod` object . . . . . . . . . . 46
8.5 Printing in PostScript . . . . . . . . . . 48
8.6 Printing to PDF files . . . . . . . . . . 48

**Appendix A Copying this manual . . . . . . . . . . 50**
A.1 GNU Free Documentation License . . . . . . . . . . 50
A.1.1 ADDENDUM: How to use this License for your documents . . . . . . . . . . 55

**Index . . . . . . . . . . 56**

# 1 Introduction

The Ringing Class Library is a collection of C++ classes which implement various concepts found in change ringing, and make it possible to implement programs which use these concepts without having to write the basic algorithms over and over again.

The library is split into several sections:

- Classes for dealing with individual rows and changes, and how the interact with each other;
- Classes for building and manipulating methods;
- Classes for accessing method libraries;
- Classes for proving and analysing sequences of rows;
- Classes for producing printed output.

Thus it is possible to use whatever part of the library is necessary for your program; for example, a method editing program would use the method classes, whereas a program to simulate call changes would just use the row and change classes.

There are also several utilities included with the library. These demonstrate how to use the Ringing Class Library in your own programs, and are useful in their own right. Currently included are `gsiril`, a peal proving program with an interface similar to the popular MicroSIRIL program; `psline`, a utility to print lines for methods in PDF or PostScript; and `methsearch`, a program which searches for methods satisfying various given criteria. These utilities may be found in the '`apps`' subdirectory.

To get information about latest releases of the Ringing Class Library, to report bugs, request features, use forums and access the CVS repository, visit the project web page at `http://ringing-lib.sourceforge.net`.

# 2 Installing and using the library

The Ringing Class Library has been developed primarily for use on Unix-like and Microsoft Windows platforms. As the library is written in portable C++ and uses only standard library functions, it should be possible to use it on any standard-conforming platform. If you do find a platform on which it is not possible to compile and use the library, the authors would be interested to hear about it.

Various characteristics of the C++ compiler need to be determined before compiling the library, and are assumed to be that same when the library is compiled and linked into an application. If you get a new C++ compiler or a new standard C++ library, you should reconfigure and (probably) recompile the library before trying to use it in an application. The necessary data is stored in '`ringing/common.h`': on Unix systems this file is generated automatically; on other systems, you may have to generate it by hand from the template in '`ringing/common.h.in`'.

## 2.1 Licensing

As of version 0.2.9, the Ringing Class Library is split into two parts, which are licensed differently. This is explained again in the file '`COPYING`' in the distribution. A summary is given here, but to be sure you should consult individual files for their licensing conditions.

- Most of the library is released under the GNU Public License (GPL). A copy of this licence can be found in the file '`COPYING.GPL`' in the top directory of the distribution. This licence allows you to copy, modify and distribute the library, provided that you stay within the stated terms. However, it does **not** allow you to include the library in your own commercial programs. Any programs which use the library must again be free. For full details, please read the text of the licence.
- A *core* part of the library is under the Lesser GNU Public License (LGPL). A copy of this licence can be found in the file '`COPYING.LGPL`' in the top directory of the distribution. Unlike the GPL, this licence *does* allow you to use this part of the library in commercial programs. The reason for releasing the core part of the library under the LGPL is principally to encourage authors to use the online method database at `http://methods.ringing.org/` in their programs, whether commercial or not. All the files you need to use that database are under the LGPL.

> **Important:** Using any part of the library in a commercial program, *except* for those parts specifically stated to be under the weaker LGPL, is against the terms of the licence. If in doubt, ask the authors.

## 2.2 The Standard Template Library

Before you can use the Ringing Class Library, you will need to have a copy of the Standard Template Library (STL) which works with your compiler. The STL is a set of templates which define containers such as lists and sets, iterators to move through them, and various other useful things. If your compiler complains about not being able to find files such as '`vector.h`' and '`list.h`' then this probably means that you don't have the STL and need to get hold of it.

A suitable version of the STL is included with most major C++ compilers. If you don't have a working version, you may be able to download one: in particular, there is a free implementation of the STL available currently at `http://www.sgi.com/tech/stl`. Implementations differ slightly in how the header files are named and whether namespaces are used. If you are compiling the Ringing Class Library on a Unix-like platform, the configuration process should take care of finding out how your particular implementation works. Otherwise, you will need to edit the file '`ringing/common.h.in`' before compiling.

## 2.3 Unix-like systems

The Ringing Class Library package uses the GNU tool `autoconf` to take away the problems of coping with differences between platforms. This means that you should be able to change to the top-level directory of the source distribution and type

```
./configure
```

to configure the package. This will determine the characteristics of your C++ compiler and create a file '`ringing/common.h`' which defines various symbols used in the compilation. This file is also installed with the rest of the class library, so that the installed header files will have access to it.

You can control many aspects of how the package is compiled and installed by passing arguments to the `configure` script; for example, you can control where the library is installed (for example, in '`/usr/lib`' or '`/usr/local/lib`'). For details of all the options which are accepted, type

```
./configure --help
```

in the top-level directory of the distribution. Some more instructions for using the `configure` script are given in the file '`INSTALL`'.

On most platforms, it should be possible to build and install either static or dynamic libraries, or both.

## 2.4 Microsoft Windows

When compiling on Microsoft Windows, the range of different compilers and lack of a standard powerful scripting language make it difficult to provide a genuinely portable infrastructure to the package. However, the task of compiling and installing the library is not complicated, and easy to achieve by hand. The following steps are necessary:

- Create the file '`ringing/common-ac.h`' by editing '`common-ac.h.in`' to reflect the characteristics of your C++ compiler;
- Compile all the '`.cpp`' source files in the '`ringing`' subdirectory into a library, which you will probably want to call '`ringing.lib`';
- Install the compiled library and the header files in places where your compiler and linker will be able to find them;
- Optionally, install and run the tests in the '`tests`' subdirectory.
- Copy the documentation to somewhere permanent.
- If desired, compile and install the utilities in the '`apps`' subdirectory.

Note that the header files are designed to be included from a subdirectory called '`ringing`' within your system's collection of header files.

If you wish to compile only the part of the library which is licensed for use in commercial programs, make sure you only compile and link those files which explicitly state that they are under the Lesser GNU Public License LGPL. See Section 2.1 [Licensing], page 2.

If you are using Microsoft Developer Studio, then you can use the supplied workspace file '`ringing-lib.dsw`' to accomplish the compiling and linking of the library without any effort. In addition, with the Microsoft Visual Studio compiler you do not need to edit '`ringing/common-ac.h`' by hand: there is a ready-customised version, called '`common-msvc.h`', which will be found and used automatically.

Currently the class library should normally be compiled into a static library ('`.lib`' file). It is not yet straightforward to make the library into a DLL; if you choose to do this, you are on your own but the authors would be please to hear of any success.

## 2.5 Testing the library

The library is provided with basic test facilities that exercise the main classes and provide confidence that they are working correctly.

The tests are found under the ‘`tests`’ directory and are linked into one executable called `test`. If all tests are succesfully a message will be printed stating this on the last line of output of the program.

## 2.6 Using the library in your programs

To use the Ringing Class Library in your programs, you simply include the relevant header files in your source code, and then link the library into your final program.

Header files are expected to be found in the subdirectory ‘`ringing`’; thus you might type

```
#include <ringing/row.h>
#include <ringing/proof.h>
```

at the top of a C++ source code file. If this does not work, either your compiler is not looking in the right place for the header files, or you have not installed them in the right place. You may be able to correct this by adding an option (`-I` on most systems) to your compiler command line.

To link to the installed library, simply add the appropriate option to the command line when linking your program. On most Unix systems, you will have to add `-lringing` to link with the library.

# 3 Row and change classes

Classes for manupulating rows and changes, and all the other classes and functions described in this chapter, are declared in '`ringing/row.h`'.

For ease of notation, we use the idea that a *row* is an individual permutation of bells (such as '`13572468`'), and a *change* is a means of getting from one row to another, by swapping pairs of bells; the most convenient way to write a change is a place notation (for example '`X`' or '`1258`').

## 3.1 Row operations

Rows have a set of operations which we can use on them; in mathematical terms, they form a group. The most important operation is that of row multiplication or transposition - in this operation, the bells in one row are rearranged according to the order given by another row.

For example,

```
21345678 * 13572468 = 23571468
```

and

```
13572468 * 21345678 = 31572468
```

Note that the two above examples do *not* give the same result; that is, the order in which two things are multiplied does matter.

As an example of how row multiplication is used, suppose that we want to know the 4th row of the lead of Plain Bob Major with lead head '`17856342`'. The 4th row of the first lead (which has a lead head of rounds) is '`42618375`'. Multiplying these together gives us the answer we are looking for, namely '`57312846`'.

The identity for this operation is rounds; in other words, any row multiplied by rounds gives itself, and rounds multiplied by any row gives that row.

It is possible to define the inverse of a row as the row which, when multiplied by that row, will give rounds. For example, the inverse of '`13572468`' is '`15263748`', as

```
13572468 * 15263748 = 12345678
```

The opposite of row multiplication is row division. If '`a * b = c`', we can define '`c / b = a`'. Using the same example as above, suppose we have a lead of Plain Bob Major and we know that the fourth row is '`57312846`', and we wish to find the lead head. Just divide by the fourth row of the plain course ('`42618375`') to get the answer.

## 3.2 Row properties

There are several properties of rows which arise from group theory and can be useful in looking at properties of methods.

The *order* of a row is the number of times which that row has to be multiplied by itself before it gets back to rounds. For example, the row '`21436587`' has order 2, because if it is multiplied by itself twice, you get back to rounds. Similarly the row '`23145678`' has order 3, and the row '`23456781`' has order 8. This can be useful, for example, in seeing how many leads of a method are needed in a plain course before it comes round.

Another useful concept is the *sign* or *parity* of a row. A row is considered *even* if it takes an even number of swaps of pairs of bells to get from rounds to that row, and *odd* if it takes an odd number of swaps. (It can be shown that whether the number is odd or even doesn't depend on exactly what the sequence of swaps is).

Finally, every row can be expressed as a set of *cycles*. A cycle is a set of bells which move round in a sequential way as the row is repeated; for example, '`21345678`' has only one cycle, which is '`(12)`'; and '`12356478`' has one cycle, which is '`(456)`'. Combining these two cycles will give us the row '`213564678`'.

## 3.3 Changes

A *change* is a means for getting from one row to another. It works by swapping over pairs of bells, and no bell may move more than one place.

The normal way of representing a change is by place notation; a single change is represented by a series of numbers which each correspond to a place being made; for example, '`12`' means that all bells swap, apart from the 1 and the 2, which stay in the same place. If all the bells swap, the place notation is '`X`'.

## 3.4 The `bell` class

The `bell` class represents a single bell. An object of type `bell` is essentially an integer, and bells are numbered starting with the treble as 0. Member functions are provided to convert between this numerical representation and a printable character. The class library can cope with up to 256 bells, though only bells numbered from 0 to 32 may be described by a printable character.

### 3.4.1 Derivation

This class is not derived from any other classes.

```
class bell;
```

### 3.4.2 Constructors

`bell::bell` (*void*); [Constructor]

This constructor initialises the bell object to zero (0).

`bell::bell` (*int* `i`); [Constructor]

This constructor initialises the bell object to the integer *i*, where bells are numbered starting from 0.

### 3.4.3 Operators

`bell& bell::operator=` (*int* `i`); [Operator]

This assigns the integer value *i* to the bell object, where bells are numbered starting from 0.

`int bell::operator int` (*void*); [Operator]

This casts a bell object to the corresponding integer.

`ostream& operator<<` (*ostream&* `o`, *const bell* `b`); [Operator]

This operator writes the character representing the bell *b* to the output stream *o*.

### 3.4.4 Other functions

`bell& bell::from_char` (*char* `c`); [Function]

This function interprets the character *c* as a bell, and assigns that value to the bell object. See the next function for the mapping between characters and bells. If the character does not represent a valid bell, then an exception of the class `bell::invalid` is thrown; if you have disabled exceptions, the bell is set to the value `bell::MAX_BELLS`. The function returns `*this`.

`char bell::to_char` (*void*) *const*; [Function]

This function returns a character indicating the bell represented by the object. The characters used are, in order:

```
1234567890ETABCDFGHJKLMNPQRSUVWYZ
```

## 3.5 The `row` class

The `row` class stores a single row. It provides extensive functions to manipulate rows and perform operations on them.

### 3.5.1 Derivation

This class is not derived from any other classes.

```
class row;
```

### 3.5.2 Constructors

`row::row` (*void*); [Constructor]

This constructs an empty row.

`explicit row::row` (*int* `bells`); [Constructor]

This constructs an empty row, where *bells* is the number of bells which the row is to contain. Note that it is not initialised to anything.

`row::row` (*const char** `s`); [Constructor]
`row::row` (*const string&* `s`); [Constructor]

This constructs a row from a string, which should be the textual representation of the row you wish to construct, for example '`"135246"`' or '`"2143658709TE"`'. If the string does not contain a valid row, and if exceptions are enabled, an exception of class `row::invalid` is thrown.

`row::row` (*const row&* `r`); [Constructor]

This is the copy constructor; it creates a copy of the given row.

### 3.5.3 Operators

`row& row::operator=` (*const row&* `r`); [Operator]

This copies one row to another.

`row& row::operator=` (*const char** `s`); [Operator]
`row& row::operator=` (*const string&* `s`); [Operator]

This sets the value of a row, given a string. The string *s* should contain a textual representation of a row, such as '`21436587`'. If the string is not valid, a `row::invalid` exception is thrown.

`bool row::operator==` (*const row&* `r`) *const;* [Operator]
`bool row::operator!=` (*const row&* `r`) *const;* [Operator]

These compare two rows.

`bell row::operator[]` (*int* `i`) *const;* [Operator]

This returns the *i*th bell in the row. Note that this is not an lvalue, so you cannot assign a value to an individual bell in a row.

`row row::operator*` (*const row&* `r`) *const;* [Operator]
`row& row::operator*=` (*const row&* `r`); [Operator]

These functions multiply two rows together as explained above. If the rows are not of the same length, the shorter row is considered to be first padded out to the length of the longer row by adding the extra bells in order at the end.

`row row::operator/` (*const row&* `r`) *const;* [Operator]
`row& row::operator/=` (*const row&* `r`); [Operator]

These functions divide two rows, as explained above. If the rows are not of the same length, the shorter row is padded as for multiplication.

`row& operator*=` (*row&* `r`*, const change&* `c`*);* [Operator]
`row row::operator*` (*const change&* `c`) *const;* [Operator]

These functions apply a change to a row. If the number of bells *c* differs from the number of bells in *r*, then *c* is considered to be padded or truncated in the obvious way.

`ostream& operator<<` (*ostream&* `o`*, const row&* `r`*);* [Operator]

This writes the row to the given output stream, in the same format as returned by `row::print()`.

`bool row::operator<` (*const row&* `other`*);* [Operator]
`bool row::operator>` (*const row&* `other`*);* [Operator]
`bool row::operator<=` (*const row&* `other`*);* [Operator]
`bool row::operator>=` (*const row&* `other`*);* [Operator]

These functions perform a lexicographical comparison of the two rows. They are necessary in order that rows may be put into certain containers.

### 3.5.4 Other functions

`int row::bells` (*void*) *const;* [Function]

This returns the number of bells which the row contains.

`void row::swap` (*row&* `other`*);* [Function]

This function swaps this row with *other* in an efficient manner.

`int row::find` (*bell const&* `b`*);* [Function]

This function locates the bell, *b*, in this row and returns its place.

`row& row::rounds` (*void*)*;* [Function]

This sets the row to rounds, and returns `*this`.

`bool row::isrounds` (*void*) *const;* [Function]

This returns true if the row is rounds, and false otherwise.

`int row::ispblh` (*void*) *const;* [Function]

If the row is a lead head of Plain Bob, Grandsire or, more generally, of the Plain Bob type method with any number of hunt bells, then this function returns an integer indicating which lead head it is. Otherwise, it returns 0.

`int row::ispblh` (*int* `h`) *const;* [Function]

If the row is a lead head of Plain Bob, Grandsire or, more generally, of the Plain Bob type method with *h* hunt bells, then this function returns an integer indicating which lead head it is. Otherwise, it returns 0.

`string row::print` (*void*) *const;* [Function]

This prints the row to a string object.

`row row::inverse` (*void*) *const;* [Function]

This returns the inverse of a row, as explained above.

`int row::sign` (*void*) *const;* [Function]

This returns the sign or parity of a row: 1 for even, -1 for odd.

`int row::order` (*void*) *const;* [Function]

This returns the order of the row.

`string row::cycles` (*void*) *const;* [Function]

This expresses the row as separate cycles. The returned string will afterwards contain a list of all the cycles in the row, separated by commas; for example `row("21453678").cycles()` will return the string `"12,345,6,7,8"`.

`template <> void swap<row>` (*row&* `a`, *row&* `b`)*;* [Function]

This specialised the `swap` function which is defined in the standard library. The result is that certain standard algorithms may become much more efficient. Note that according to your namespace setup, `swap` may need to be in namespace `std`; similarly, `row` may or may not be in namespace `ringing`. This is all taken care of.

### 3.5.5 `static` functions

`static row row::rounds` (*int* `n`)*;* [Function]
`static row row::queens` (*int* `n`)*;* [Function]
`static row row::kings` (*int* `n`)*;* [Function]
`static row row::tittums` (*int* `n`)*;* [Function]
`static row row::reverse_rounds` (*int* `n`)*;* [Function]

These return the row corresponding to rounds, queens, kings, tittums and reverse rounds respectively on *n* bells.

`static row row::pblh` (*int* `n`, *int* `h` *= 1*)*;* [Function]

This returns the first lead head of Plain Bob ($h = 1$), Grandsire ($h = 2$), or more generally the Plain Bob type method on *n* bells with *h* hunt bells.

`static row row::cyclic` (*int* `n`, *int* `h` *= 1*, *int* `c` *= 1*)*;* [Function]

This returns a cyclic row on *n* bells with *h* initial fixed (hunt) bells. The variable *c* controls the number of bells moved from the front of the row to the end. Thus, `cyclic(6,1,2) == "145623"`.

## 3.6 The `change` class

The `change` class stores a single change, i.e. a means of getting from one row to another.

### 3.6.1 Derivation

This class is not derived from any other classes.

```
class change;
```

### 3.6.2 Constructors

`change::change` (*void*)*;* [Constructor]

This constructs an empty change.

`explicit change::change` (*int* `n`)*;* [Constructor]

This constructs an empty change for *n* bells, which will initially contain no swaps, that is, all bells will remain in the same place.

`change::change` (*const change&* `c`)*;* [Constructor]

This is the copy constructor; it makes a new copy of *c*.

`change::change` (*int* `n`, *char** `pn`)*;* [Constructor]
`change::change` (*int* `n`, *const string&* `pn`)*;* [Constructor]

This constructs a change on *n* bells, with place notation as given in *pn*. This should consist of a sequence of characters which signify the places to be made, arranged in ascending order. If no places are to be made, the string `"X"` should be used. Note that you can miss out external places, unless no internal places are made at all. If the place notation is not valid, this constructor will throw an exception of class `change::invalid`.

### 3.6.3 Operators

`change& change::operator=` (*const change&* `c`); [Operator]

This assigns the value of one change to another.

`bool change::operator==` (*const change&* `c`) *const;* [Operator]
`bool change::operator!=` (*const change&* `c`) *const;* [Operator]

These compare two changes. Note that to be equal, the changes must be defined on the same number of bells; for example, '`"12"`' on 4 bells is different from '`"12"`' on 6 bells.

`bell& operator*=` (*bell&* `b`, *const change&* `c`); [Operator]
`bell operator*` (*bell* `b`, *const change&* `c`); [Operator]

These return the effect of applying the change *c* to the bell *b*. For example, `3 * change(4,"34") == 4`. This is useful in tracing the path of one particular bell through a series of changes.

`ostream& operator<<` (*ostream&* `o`, *const change&* `c`); [Operator]

This writes the place notation for the change *c* to the given output stream.

`bool change::operator<` (*const change&* `c`) *const;* [Operator]
`bool change::operator>` (*const change&* `c`) *const;* [Operator]
`bool change::operator<=` (*const change&* `c`) *const;* [Operator]
`bool change::operator>=` (*const change&* `c`) *const;* [Operator]

If both changes have the same number of bells, these performs a lexicographical comparison on the swaps present in each changes; for example `change(6, "14")` compares less than `change(6, "1236")`.

If the two changes have different numbers of bells the one with fewer bells compares less than the one with more bells.

### 3.6.4 Other functions

`void change::swap` (*change&* `c`); [Function]

This function swaps this change with the change given by *c* in an efficient manner.

`int change::bells` (*void*) *const;* [Function]

This returns the number of bells on which the change is defined.

`int change::sign` (*void*) *const;* [Function]

This returns the sign of the change: -1 if an odd number of pairs are swapped, +1 if an even number of pairs are swapped.

`bool change::findswap` (*bell* `i`) *const;* [Function]

Returns true if the change swaps bells *i* and *i+1*, and false otherwise.

`bool change::findplace` (*bell* `i`) *const;* [Function]

Returns true if the change doesn't move the bell in the *i*th place (i.e. if *i*ths place is made), and false otherwise.

`int change::count_places` (*void*) *const;* [Function]

This function returns the number of places made in the change.

`bool change::swappair` (*int* `i`); [Function]

If the change doesn't currently swap bells *i* and *i+1*, then this will add that swap. If those bells are swapped, this will remove the swap. If the bells *i-1* and *i*, or *i+1* and *i+2*, are currently swapped, those swaps are removed.

This returns true if after the function call, the pair of bells *i* and *i+1* are swapped, and false otherwise. If exceptions are enabled and the bell *i+1* is greater the number of bells in the change, an exception of class `change::out_of_range` will be thrown.

This function makes it possible for the user to edit changes in such a way that they will always end up in a sensible state.

`change change::reverse` (*void*) *const;* [Function]
This returns the reverse of a change; that is, the change is flipped over so that on 8 bells for example, 2nds place becomes 7ths place and so on.

`bool change::internal` (*void*) *const;* [Function]
This returns true if the change contains internal places, and false otherwise.

`string change::print` (*void*) *const;* [Function]
This function prints the place notation for the change to a string.

`template <> void swap<change>` (*change&* `a`, *change&* `b`)*;* [Function]
This specialised the `swap` function which is defined in the standard library. The result is that certain standard algorithms may become much more efficient. Note that according to your namespace setup, `swap` may need to be in namespace `std`; similarly, `change` may or may not be in namespace `ringing`. This is all taken care of.

## 3.7 The `permute` function

The `permute` function is a utility function which makes it possible to use some of the standard algorithms contained in the C++ library. There are two forms of the function: the first takes an unsigned integer as its argument, and the second takes a reference to a row. In either case, a function object (of type `permuter` or `row_permuter`) is returned, which can be used with standard algorithms.

`inline permuter permute` (*unsigned* `n`)*;* [Function]
This global function returns a function object of type `permuter`, which contains a row initialised to rounds on *n* bells. When the function object is called with an argument of either a row or change, its internal row is multiplied by that row or change and then returned.

`inline row_permuter permute` (*row&* `r`)*;* [Function]
This global function creates a `row_permuter` function object, which contains a reference to the row *r*. When the function object is called with a row or change as an argument, the row *r* will be multiplied by that row or change.

`const row& permuter::operator()` (*const row&* `r`)*;* [Operator]
`const row& permuter::operator()` (*const change&* `c`)*;* [Operator]
These operators multiply the permutation contained in the `permuter` object by the given row or change, and return the new permutation.

`const row& row_permuter::operator()` (*const row&* `r`)*;* [Operator]
`const row& row_permuter::operator()` (*const change&* `c`)*;* [Operator]
These operators multiply the row which the `row_permuter` object refers to by the given row or change. A reference to that row is returned.

## 3.8 The `row_block` class

The `row_block` class is an array of rows which has associated with it a reference to an array of changes, and can recalculate itself from those changes. For example, suppose that the variable `c` of type `vector<change>` holds one lead of a method; then it is possible to define a variable of type `row_class` which, once it is told what the lead head is, will calculate the rows for one lead of the method.

Note that the `row_block` object does not store its own copy of the `vector<change>` object from which it is calculated. This means that the changes may be altered and the resulting rows recalculated easily; it also means that the changes must not be deallocated while the `row_block` object is in existence.

### 3.8.1 Derivation

The `row_block` class is derived from `vector<row>`.

```
class row_block : public vector<row>;
```

### 3.8.2 Constructors

`row_block::row_block` (*const vector<change>&* `c`*);* [Constructor]

This creates a block of rows using the changes in *c*, starting from rounds.

`row_block::row_block` (*const vector<change>&* `c`, *const row&* `r`*);* [Constructor]

This creates a block of rows using the changes in *c*, starting from the row given in *r*.

### 3.8.3 Other functions

`row& row_block::set_start` (*const row&* `r`*);* [Function]

This sets the first row to *r*. Note that this function does *not* recalculate the rest of the rows afterwards; you must do this by calling `recalculate()`.

`row_block& row_block::recalculate` (*int* `start` *= 0);* [Function]

This recalculates all the rows in the array, starting from the element numbered *start*; if this is not specified, all the rows are recalculated. If, for example, you know that the `changes` object which the row block is based on has been changes halfway through, you don't have to recalculate all the rows, just the ones after that point.

`const vector<change>& row_block::get_changes` (*void*) *const;* [Function]

This returns the `vector<change>` object on which the row block is based.

## 3.9 Other functions in '`row.h`'

`template<OutputIterator, ForwardIterator> void interpret_pn` [Function]
(*int* `bells`, *ForwardIterator* `start`, *ForwardIterator* `finish`, *OutputIterator* `out`*);*

This template function is for expanding place notation into changes. The arguments are as follows: *bells* is the number of bells; *start* and *finish* should be forward iterators with value type `char`, and indicate a string of characters containing the place notation; and *out* is an output iterator of value type `change`, to which the changes will be written.

The syntax of the place notation is as follows: place notation consists of a sequence of blocks, which are delimited by commas. Each block is a sequence of changes, with '`X`' or '`-`' meaning a cross change. Changes other than cross changes are separated by full stops. A block may optionally be preceded by an ampersand '`&`', which means that the entire block, except the last change, is repeated backwards. All other characters are ignored.

For examples of this function in use, look at one of the `method` constructors in '`ringing/method.h`' or in the example code.

# 4 Method classes

The Ringing Class Library provides many useful functions for dealing with methods, finding information about them and classifying them.

A *method* consists of one block of changes which is repeated over and over again; this is called a *lead*. If, at the end of one lead, every bell is in a different position, then the method is called a *principle* and the leads are called *divisions*. Any bells is a method which end up in the same place at the end of a lead are called *hunt bells*.

The Ringing Class Library defines an object `method` which is simply a block of changes (it is derived from `vector<change>`) and which provides a large range of member functions. Many of these are used for finding out things about the method, such as the number of leads in the plain course, the type of method, the full name and so on.

## 4.1 The `method` class

The `method` class represents a method by storing the changes which make up one lead of the method. This class is declared in '`ringing/method.h`'.

A method may have a name, which is a string variable. This name is the *base name* of the method, that is, it is the name without any extra bits such as 'Surprise' or 'Triples'. For example, the base name of Plain Bob Major is simply 'Plain'. The parts of the name other than the base can be worked out by the Ringing Class Library.

### 4.1.1 Derivation

The `method` class is derived from `vector<change>`:

```
class method : public vector<change>
```

### 4.1.2 Constructors

`explicit method::method` (*int* **`length`** *= 0, int* **`bells`** *= 0, char** **`name`** *= "Untitled");* [Constructor]

This constructor creates an empty method on *bells* bells, with *length* changes in the lead, and with name *name*. All the bells stay in the same place for the entire lead.

`method::method` (*const char** **`pn`**, *int* **`bells`**, *char** **`name`** *= "Untitled");* [Constructor]
`method::method` (*const string&* **`pn`**, *int* **`bells`**, *const string&* **`name`** *= "Untitled");* [Constructor]

This constructor creates a method on *bells* bells from the given place notation, which should be an entire lead. The method is given the base name *name*. For information on how to specify the place notation, see the discussion of `interpret_pn` in Section 3.9 [Other functions in row.h], page 12.

`method::method` (*const method&* **`m`**)*;* [Constructor]

This is the copy constructor.

### 4.1.3 Other functions

`const char* method::name` (*void*) *const;* [Function]

This returns the base name of the method. Note that this may be an empty string, for example in Little Bob.

`void method::name` (*const char** **`name`**)*;* [Function]
`void method::name` (*const string&* **`name`**)*;* [Function]

This sets the base name of the method to *name*.

`int method::bells` (*void*) *const;* [Function]
This returns the number of bells on which the method is defined.

`int method::length` (*void*) *const;* [Function]
This returns the number of changes in a lead of the method.

`void method::push_back` (*const change&* `c`)*;* [Function]
`void method::push_back` (*const string&* `s`)*;* [Function]
These utility functions add a change to the end of the method. In the first case, *c* is the change to be added; in the second case, *s* is place notation for the change to be added.

`row method::lh` (*void*) *const;* [Function]
This returns the first lead head of the method.

`bool method::issym` (*void*) *const;* [Function]
This returns true if the method is symmetrical about the half lead (and has an even number of changes in the lead), or false otherwise.

`bool method::isdouble` (*void*) *const;* [Function]
This returns true if the method is double, or false otherwise. Note that double does not imply symmetrical.

`bool method::isregular` (*void*) *const;* [Function]
This returns true if the method is regular (that is, has Plain Bob lead heads), or false otherwise.

`int method::huntbells` (*void*) *const;* [Function]
This returns the number of hunt bells in the method.

`int method::leads` (*void*) *const;* [Function]
This returns the number of leads in the plain course of the method.

`char* method::lhcode` (*void*) *const;* [Function]
This returns the standard code for the lead end and lead head of the method. The result is stored in a static internal buffer.

`bool method::issym` (*bell* `b`) *const;* [Function]
This returns true if the path of bell *b* in a lead of the method is symmetrical about the half lead, or false otherwise.

`bool method::isplain` (*bell* `b` *= 0*) *const;* [Function]
This returns true if bell *b* plain hunts for the whole lead of the method, or false otherwise.

`bool method::hasdodges` (*bell* `b`) *const;* [Function]
This returns true if bell *b* dodges during the lead, or false otherwise.

`bool method::hasplaces` (*bell* `b`) *const;* [Function]
This returns true if bell *b* makes any internal places during the lead, or false otherwise.

`int method::methclass` (*void*) *const;* [Function]
This function returns an integer which depends on the class of the method. The following values are defined:

```
enum method::m_class {
  M_UNKNOWN, M_PRINCIPLE, M_BOB, M_PLACE, M_TREBLE_BOB, M_SURPRISE,
  M_DELIGHT, M_TREBLE_PLACE, M_ALLIANCE, M_HYBRID, M_SLOW_COURSE,
  M_MASK = 0x0f,
  M_DIFFERENTIAL = 0x10,
  M_LITTLE = 0x80
};
```

The `M_DIFFERENTIAL` and `M_LITTLE` bits are flags which will be either set or cleared accordingly. (For the purpose of this function, "Differential" is considered a method class. Differential hunters will have the `M_DIFFERENTIAL` bit set, together with one of the other classes; differential methods will return `M_DIFFERENTIAL | M_PRINCIPLE`.)

`char* method::fullname` (*char** `buffer`) *const;* [Function]

This function places the full name of the method in *buffer*, which should be long enough to receive it, and returns a pointer to it. The full name consists of:

- the base name of the method;
- the word "Little", if appropriate;
- the class of the method, or none if it is a principle;
- the stage (number of bells) of the method.

Note that Grandsire, Union and their related methods are treated specially: the full name of these methods does not include their class, nor in the case of Little Grandsire does it contain "Little".

`string method::fullname` (*void*) *const;* [Function]

This function returns the full name of the method as a string.

`int method::maxblows` (*void*) *const;* [Function]

This function returns the maximum consecutive blows made in any place in this method. (If the method has one bell fixed throughout, the lead length is returned.)

### 4.1.4 `static` functions

`static const char* method::stagename` (*int* `bells`)*;* [Function]

This function returns a pointer to a string describing the stage of *bells* bells: thus `"Minimus"`, `"Doubles"`, and so on. For stages above Sixteen, the number is given in digits.

`static string method::classname` (*int* `class`)*;* [Function]

This function returns a string describing the method class given in *class*, which should be as indicated in the explanation of `method::methclass()` above. For the purpose of this function, "Differential" and "Little" are both considered to be method classes.

### 4.1.5 `static` variables

`static const char* method::txt_double;` [Variable]

This points to the string `"Double"`.

`static const char* method::txt_little;` [Variable]

This points to the string `"Little"`.

# 5 Method libraries

The Ringing Class Library provides a extensible framework for loading and manipulating method libraries (collections) of various types. The classes that provide this framework are declared in '`ringing/library.h`'.

The class library contains built-in code for accessing certain types of libraries. Currently these are:

- Central Council libraries in text format;
- MicroSIRIL libraries;
- XML libraries.

The class framework is designed so that programmers can easily add their own types of libraries, and handle them in exactly the same way as the build-in types.

When opening a library, it is not necessary to specify what type of library it is: usually this can be detected automatically.

## 5.1 Using libraries

This section describes how to open and read method libraries.

### 5.1.1 Initialisation

Before the auto-detection of libraries will work, the `library` class needs to know which library types it should try when opening a library. This is done by *registering* the different library types, and should be done sometime during program startup. To make this as simple as possible, each of the built-in library types has a static member function called `registerlib` for this purpose.

```
#include <ringing/xmllib.h>
#include <ringing/cclib.h>

xmllib::registerlib();
cclib::registerlib();

// Now we can open XML and Central Council libraries automatically
```

The order in which library types are registered is the order in which they will be tried when opening a library.

### 5.1.2 The `library` class

The `library` class provides all the functions you need for opening and accessing method libraries. These include:

- Constructors for opening method libraries;
- Iterator-style functions to access methods;
- Functions for accessing methods by name.

Note that the names of methods in libraries may not be what the user expects. For example, in MicroSIRIL libraries, method names are run together into one word, say '`Doubledublin`'.

#### 5.1.2.1 Derivation

This class is not derived from any other classes.

```
class library;
```

#### 5.1.2.2 Constructors

`library::library` (*void*); [Constructor]

Constructs an empty (and useless) library object.

`library::library` (*const string&* ***`filename`***); [Constructor]

This constructor attempts to automatically detect what type of library *filename* is, and create an appropriate `library` object to interpret it. For a library type to be correctly auto-detected, that type must be registered see Section 5.1.1 [Initialisation for libraries], page 16.

After constructing a library, the `library::good` function can be used to see whether auto-detection succeeded.

In addition to this constructor, which auto-detects the type of library, it is also possible to open a library of a fixed type, by constructing an object for a specific library type. See Section 5.2 [Built-in library types], page 19.

#### 5.1.2.3 Iteration functions

The `library` class has an associated iterator class.

`library::const_iterator;` [Data type]

In the language of the Standard Template Library, this is an *input iterator*. This means that you may only use it to read methods from the library in order, starting at the beginning. The value type of the iterator is `library_entry` see Section 5.1.3 [The library_entry class], page 18.

`library::const_iterator library::begin` (*void*) *const;* [Function]
`library::const_iterator library::end` (*void*) *const;* [Function]

These functions return iterators pointing to the beginning and end of the library, respectively.

This is how these functions may be used to iterate through the library:

```
// Open a library
library l("my_library.xml");
library::const_iterator i;

// Print the base name of each method in it
for(i = l.begin(); i != l.end(); ++i)
  cout << i->base_name() << endl;
```

#### 5.1.2.4 Other functions

`bool library::good` (*void*) *const;* [Function]

This function returns true if the library is open and ready to read methods, and false otherwise.

`virtual method library::load` (*const char** ***`name`***, *int* ***`stage`***) *const;* [Function]

This function loads a method from the library by name. Note that the names under which methods are stored in a library depend entirely on the type of library.

If no such method is found, if exceptions are enabled, an exception of class `library_base::invalid_name` will be thrown; otherwise, the method returned will be an empty method with the name ‘`"Not Found"`’.

`virtual int library::dir` (*list<string>&* ***`result`***) *const;* [Function]

This function returns a list of the names of the methods in the library.

`virtual int library::mdir` (*list<method>&* ***`result`***) *const;* [Function]

This function loads all the methods in the library and returns them as a list.

### 5.1.3 The `library_entry` class

The `library_entry` class is the value type of the iterator on a library. It is used to get information about the method from the library without constructing a `method` object.

`string library_entry::name` (*void*) *const;* [Function]

This returns the method's name in whatever form the library stores it internally. Thus it may or may not contain the class name(s) or stage name of the method. In the case of MicroSIRIL libraries, it will not even have whitespace in the name.

`string library_entry::base_name` (*void*) *const;* [Function]

This returns the base name of the method (i.e. the name with the class name(s) and stage name removed). The value returned is suitable for passing to the `method` constructor.

`string library_entry::pn` (*void*) *const;* [Function]

This returns a string containing the method's place notation.

`int library_entry::bells` (*void*) *const;* [Function]

This returns the number of bells on which the method is defined.

`method library_entry::meth` (*void*) *const* [Function]

This constructs and return a `method` object.

`template <class Facet> bool library_entry::has_facet` (*Facet const** `f` *= NULL* ) *const;* [Function]

This function returns `true` if the method entry contains information about the given facet. Note that just because a method library type supports a certain facet, that does not necessarily mean that every method in the library actually has that facet. For example, an entry in a Central Council library may not contain any information about the first peals in a method; or an entry in an XML library may not contain certain elements.

The contents of the argument *f* are not used, but this may be used to indicate the type of the facet as an alternative to explicit template instantiation.

`template <class Facet> typename Facet::type library_entry::get_facet` (*Facet const** `f` *= NULL* ) *const;* [Function]

This function returns the value of the specified facet of the method entry.

The contents of the argument *f* are not used, but this may be used to indicate the type of the facet as an alternative to explicit template instantiation.

For more information on facets and an example of how these functions are used, see Section 5.1.4 [Facets], page 18.

### 5.1.4 Facets

In addition to usual information such as name, stage and place notation, some method collections contain further information. For example, the Central Council method collections contain the date and place of the first peal in each method. To give access to such information, the Ringing Class Library has a fully typed system of method *facets*.

A facet is something which has a C++ type, and which may or may not be part of an entry in a method library. A few facets are declared as standard; library types are free to declare new facets of their own.

For example, one of the standard facets is called `rw_ref` and is of type `string`. The facet is intended to contain a string citing the method's appearance in the *Ringing World*. The following code checks to see whether the first entry in the library `l` contains this information, and if so prints it.

```
library::iterator i = l.begin();
if( i->has_facet< rw_ref >() )
  cout << i->get_facet< rw_ref >();
```

For more information on these functions, see Section 5.1.3 [The library_entry class], page 18.

#### 5.1.4.1 Built-in facets

Some facets are declared in '`ringing/peal.h`'. These facets are available in some of the built-in library types.

`rw_ref`
: This facet, of type `string`, contains a reference to the method's appearance in the *Ringing World*.

`first_tower_peal`
: This facet, of type `peal`, contains the date and place of the first tower bell peal in the method.

`first_hand_peal`
: This facet, of type `peal`, contains the date and place of the first handbell peal in the method.

The last two are of type `peal`, which is a class also declared in '`ringing/peal.h`'. It is a very simple class and defines the following functions.

`peal` (*void*); [Constructor]
`peal` (*const peal::date&* ***DATE***, *const string&* ***PLACE***); [Constructor]
These functions construct objects in the obvious way.

`peal::date const& peal::when` (*void*); [Function]
This function returns the date of the peal.

`string const& peal::where` (*void*); [Function]
This function returns the location of the peal.

The `peal::date` type is also very simple.

```
struct peal::date {
  int day, month, year;
};
```

`peal::date` (*void*); [Constructor]
`peal::date` (*int* ***DAY***, *int* ***MONTH***, *int* ***YEAR***); [Constructor]
These functions construct objects in the obvious way.

## 5.2 Built-in library types

### 5.2.1 XML libraries

A specification for method libraries in XML can be found at `http://methods.ringing.org/`. This site is also home to a server which serves XML method data in response to HTTP queries. The classes declared in '`ringing/xmllib.h`' can not only parse the XML data, but can also automatically download method data from the server.

To use XML method libraries, you need to have the Xerces library from the Apache project, available at `http://xml.apache.org/`.

#### 5.2.1.1 The `xmllib` class

The `xmllib` class is derived from the `library` class.

```
class xmllib : public library;
```

`xmllib::xmllib` (*xmllib::file_arg_type* `TYPE`*, const string&* `NAME`*);* [Constructor]

This constructor opens an XML library. The first argument *TYPE* describes what sort of resource is to be opened, and should be one of the following:

`xmllib::filename`
: Open a file; *NAME* contains the name of the file.

`xmllib::url`
: Open a URL; *NAME* contains the URL. To use this, Xerces must have been built with Internet access.

`xmllib::default_url`
: Query the online method database at `http://methods.ringing.org/`. The argument *NAME* contains the query string, that is, the part of the URL after the '`?`' character. For information on what this should contain, see the detailed information on the database's web site.

```
// Query the database for all methods named Lincolnshire
xmllib l(xmllib::default_url, "name=Lincolnshire");
```

An XML method collection in a file may also be opened using the library auto-detection mechanism, by simply constructing a `library` object. See Section 5.1.2.2 [library constructors], page 17.

`static void xmllib::registerlib` (*void*) [Function]

This function registers the XML library type with the library auto-detection mechanism.

#### 5.2.1.2 XML library facets

The XML library type supports only the built-in facets. See Section 5.1.4.1 [Built-in facets], page 19.

### 5.2.2 Central Council libraries

The Methods Committee of the Central Council publishes collections of methods on its web site, in a text format. The classes declared in '`ringing/cclib.h`' can read these method collections.

#### 5.2.2.1 The `cclib` class

The `cclib` class is derived from the `library` class.

```
class cclib : public library;
```

There is one constructor.

`cclib::cclib` (*const string&* `filename`*);* [Constructor]

This function attempts to open the given file as a Central Council method library.

`static void cclib::registerlib` (*void*) [Function]

This function registers the Central Council library type with the library auto-detection mechanism. It will agree to open any file which has the correct CC copyright notice in the first line.

#### 5.2.2.2 Central Council library facets

The Central Council library type supports the built-in facets see Section 5.1.4.1 [Built-in facets], page 19 and one further facet.

`cclib::ref`
: This facet, of type `int`, contains the reference number of the method in the library file. This is the number which appears at the beginning of the line containing the method.

### 5.2.3 MicroSIRIL libraries

MicroSIRIL was an ingenious and widely-used early peal proving program for PCs. The method libraries used are in a simple text format, with one method per line; they contain, for each method, simply the name, lead head code and place notation.

The Ringing Class Library can read MicroSIRIL libraries, providing they satisfy one requirement: the last portion of the filename, before any extension, must be the number of bells on which all the methods in the file are. This is because it is impossible to tell the stage from just the place notation. So, for example, a method library file might be called ‘`s8`’ or ‘`new_principles10.txt`’.

Beware that the MicroSIRIL library interface will agree to open any file having a name satisfying the above requirement. For this reason, if auto-detecting method library types, you may want to put the MicroSIRIL library type at the end of your list.

When reading methods from MicroSIRIL libraries, remember that the name of each method may only consist of a single word; usually names consisting of several words are run together into one.

#### 5.2.3.1 The `mslib` class

The `mslib` class is derived from the `library` class.

```
class mslib : public library;
```

There is one constructor.

`mslib::mslib` (*const string&* ***`filename`***); [Constructor]
: This function attempts to open the given file as a MicroSIRIL method library.

`static void mslib::registerlib` (*void*) [Function]
: This function registers the MicroSIRIL library type with the library auto-detection mechanism. It will agree to open any file which has a name containing a number just before any extension.

#### 5.2.3.2 MicroSIRIL library facets

There are no facets available for MicroSIRIL libraries.

## 5.3 Library framework

This section describes in greater detail how the various classes which implement method libraries interact. It is intended for those writing new classes to support new library types.

The basic library framework comprises five classes, all of which are declared in the header file ‘`ringing/library.h`’.

`class library_base` [Class]
: This is the basic interface that all implementations will need to implement. It describes how to iterate through a library, and provides hooks that can be used to provide more efficient algorithms for searching through the library.

`class library` [Class]

The `library` class serves two purposes: it acts as a shared pointer to a library_base, and it handles the auto-detection of libraries. Implementations only need to derive from this if they want to provide an alternative to the standard auto-detection mechansim.

`class library_entry` [Class]

This class represents an entry in a library. It has value semantics; thus it is safe to hold onto multiple entries from a single library.

`class library_base::const_iterator` [Class]

This is an input iterator that iterates over the entries in a library. Dereferencing the iterator returns a `library_entry`.

`class library_entry::impl` [Class]

Each `library_entry` class holds onto a pointer to one of these. It provides an interface to read library entries that all library implementations must implement.

### 5.3.1 The `library_base` class

The `library_base` class provides the interface that library implementations must implement to create iterators on a library. It also has virtual functions can be overriden to provide more efficient ways to search and list libraries.

`virtual library_base::~library_base` (*void*); [Destructor]

A virtual destructor.

`virtual library_base::const_iterator library_base::begin` (*void*) *const = 0;* [Function]

This should be overriden to construct a `library_base::const_iterator` pointing to the start of the library.

`library_base::const_iterator library_base::end` (*void*) *const;* [Function]

This returns a default constructed `library_base::const_iterator` representing one past the end of the library.

`virtual bool library_base::good` (*void*) *const = 0;* [Function]

This function should be overriden to return true if a valid library is currently loaded and false otherwise.

`virtual method library_base::load` (*const char** `name`, *int* `stage`) *const;* [Function]

The default implementation of `load` iterates through the library searching for an entry with the correct name. When comparing method names, it does so in a case-insensitive manner. If *stage* is non-zero, `load` requires that the method is on that number of bells.

If no such method is found, if exceptions are enabled, an exception of class `library_base::invalid_name` will be thrown; otherwise, the method returned will be an empty method with the name '`"Not Found"`'.

Library implementations may wish to override this if there is a more efficient way to locate the method.

`virtual int library_base::dir` (*list<string>&* `result`) *const;* [Function]
`virtual int library_base::mdir` (*list<method>&* `result`) *const;* [Function]

The default implementations of `dir` and `mdir` iterate through the library, pushing each name (in the case of `dir`) or method (in the case of `mdir`) onto *result*. The return value is the number of values pushed onto `result`.

Library implementations may wish to override this if there is a more efficient way to locate the method.

`virtual bool library_base::writeable` (*void*) *const;* [Function]
`virtual bool library_base::save` (*const method& `m`*); [Function]
`virtual bool library_base::rename_method` (*const string& `name1`, const string& `name2`*); [Function]
`virtual bool library_base::remove` (*const string& `name`*); [Function]

These functions are for writing to method libraries. This functionality is still experimental.

### 5.3.2 The `library_base::const_iterator` class

In the terminology of the STL, the `library_entry::const_iterator` class is an Input Iterator but not a Forward Iterator. This means that multiple iterators on the same library will not work as expected.

`library_base::const_iterator::const_iterator` (*void*); [Constructor]

This is the default constructor. It is undefined behaviour to dereference or increment the default constructed `library_base::const_iterator`.

`library_base::const_iterator::const_iterator` (*library_base* `lb`, library_entry::impl* `val`*); [Constructor]

This constructor creates an iterator on library *lb* and with a library entry implementation of *val*. The constructor takes ownership of *val*, which must have been allocated using `new`; it does not take ownership of *lb*. Typically, this function should only be called by the `begin` function of classes derived from `library_base`.

`bool library_base::const_iterator::operator==` (*const library_base::const_iterator& `other`*) *const;* [Function]
`bool library_base::const_iterator::operator!=` (*const library_base::const_iterator& `other`*) *const;* [Function]

Compare two iterators. It is undefined behaviour to compare two iterators from different libraries.

`library_entry library_base::const_iterator::operator*` (*void*) *const;* [Function]
`library_entry* library_base::const_iterator::operator->` (*void*) *const;* [Function]

Dereferences an iterator to retrieve the current `library_entry`. It is undefined behaviour to dereference an iterator that compares equal to the end iterator of that library.

`library_base::const_iterator& library_base::const_iterator::operator++` (*void*) *const;* [Function]
`library_base::const_iterator library_base::const_iterator::operator++` (*int*) *const;* [Function]

The prefix and postfix increment operators move the iterator on to point to the next entry in the library. If there are no further entries in the library, the iterator will compare equal to the library's end iterator.

### 5.3.3 `library` internals

`library::library` (*void*); [Constructor]

Default constructs a library holding a `NULL` `library_base` pointer.

`library::library` (*const string& `filename`*); [Constructor]

This constructor attempts to automatically detect what type of library *filename* is, and create an appropriate `library_base` to interpret it. For a library type to be correctly auto-detected a function of type `library::init_function` must be registered with `library::addtype`.

After constructing a library, the `library::good` function can be used see if auto-detection succeeded.

`library::library` (*library_base** `lb`); [Constructor]

This constructor is provided to allow libraries the flexibility not to use the auto-detection mechanism. The argument, *lb*, must have allocated by `new` and gets deleted by the `library`'s destructor. For more information on how to use this, see Section 5.4.2 [Alternative creation mechanisms], page 26.

`typedef library_base* (* library::init_function )` (*const string&* `filename`); [Typedef]

Functions with this signature are used by the `library` construtor to attempt to load a library. Each derived library class that supports auto-detection should register a function of this signature with the `library` class by calling the `library::addtype` function. When these functions get called, *filename* is the name of the library file. If a library can read the file it should return a pointer to a class derived from `library_base` allocated with `new`; otherwise the function should return NULL.

`static void library::addtype` (*library::init_function* `lt`) [Function]

This is used to register a specific (derived) library class to the list of classes that can be auto-detected. This function is typically called by the static function `registerlib` in the derived classes.

`typedef library_base::const_iterator library::const_iterator;` [Typedef]
`library::const_iterator library::begin` (*void*) *const;* [Function]
`library::const_iterator library::end` (*void*) *const;* [Function]

If the library holds a `library_base`, these function call down to the corresponding functions on that, otherwise both functions return a default constructed `const_iterator`.

`bool library_base::good` (*void*) *const;* [Function]

This function returns true if the library is valid, and false otherwise.

`virtual method library::load` (*const char** `name`, *int* `stage`) *const;* [Function]
`virtual int library::dir` (*list<string>&* `result`) *const;* [Function]
`virtual int library::mdir` (*list<method>&* `result`) *const;* [Function]
`virtual bool library::writeable` (*void*) *const;* [Function]
`virtual bool library::save` (*const method&* `m`); [Function]
`virtual bool library::rename_method` (*const string&* `name1`, *const string&* `name2`); [Function]
`virtual bool library::remove` (*const string&* `name`); [Function]

These functions all call down to the corresponding functions on the `library_base`; see Section 5.3.1 [The library_base class], page 22 for details. They must not be called if `library::good()` is false.

### 5.3.4 The `library_entry` internals

The `library_entry` class holds a pointer to a class derived from `library_entry::impl` which reads data from a particular library entry.

`library_entry::library_entry` (*void*); [Constructor]

This constructor initialises the `library_entry` with a `NULL` `library_entry::impl` pointer. It is undefined behaviour to call any accessor functions on a default constructed `library_entry`.

`library_entry::library_entry` (*library_entry::impl** `PIMPL`); [Constructor]

This constructor creates a new object from the given pointer.

`string name` (*void*) *const;* [Function]
`string base_name` (*void*) *const;* [Function]
`string pn` (*void*) *const;* [Function]
`int bells` (*void*) *const;* [Function]
`method meth` (*void*) *const* [Function]

These functions all call the corresponding function on the `library_entry::impl` object.

### 5.3.5 The `library_entry::impl` class

The `library_entry::impl` class provides the interface that library implementations must implement to parse individual library entries. Each `library_entry` and `library::const_iterator` holds onto one of these.

`virtual library_entry::impl::~impl` (*void*)*;* [Destructor]

A virtual destructor.

`virtual library_entry::impl* library_entry::impl::clone` (*void*) *const = 0;* [Function]

This must be overriden to perform a deep copy of the library. Almost always this should be implemented as follows:

```
virtual library_entry::impl* mylib::entry_type::clone() const {
  return new mylib::entry_type(*this);
}
```

`virtual bool library_entry::impl::readentry` (*library_base&* `lb`) *const = 0;* [Function]

The function should be overriden to read the next entry from the library and return true unless the end of the library has been reached. The source of the data should be held within the `library_base` implementation, and not this class. For example, if the library implementation reads from a file via an `ifstream`, the `library_base` class should contain the `ifstream` and this class should access it via *lb*:

```
virtual bool mylib::entry_type::readentry( library_base& lb ) {
  ifstream& ifs = dynamic_cast< mylib& >( lb ).ifs;
  std::getline( ifs, this->entry );
}
```

None of `library_entry`'s other functions should ever access the parent `library_base` class.

This function gets called whenever a `library::const_iterator` is constructed or incremented.

`virtual string library_entry::impl::name` (*void*) *const = 0;* [Function]
`virtual string library_entry::impl::base_name` (*void*) *const = 0;* [Function]

These should be overriden to return the name of the current entry. The `name` function should return a name which is unique within the library (this may require the class name(s) and/or stage name be included); the `base_name` function should return the name without either the class or stage name.

These functions may be implemented to simply return a name that was parsed by `readentry`; alternatively it might be implemented to parse some internal representation of the entry (such as a string containing the whole entry). It must not attempt to access the parent `library_base` which may no longer exist. The `readentry` function will always have been called before this function gets called.

`virtual string library_entry::impl::pn` (*void*) *const = 0;* [Function]

This should be overriden to return the place notation of the current method.

`virtual string library_entry::impl::pn` (*void*) *const = 0;* [Function]

This should be overriden to return the number of bells that the current method is on.

## 5.4 Implementing new library classes

There are two ways to implement new library classes, depending on whether or not the new library class can support the automatic detection of library type done in the `library(const string& filename)` constructor. In general, if the library is based on a single text file, the auto-detection mechanism adequately provides a simple way for users of to transparently use various types of library. More complicated types of library, such as those based on a database, or on multiple files, fit less well with the auto-detection mechanism, and some alternative mechanism is often better.

### 5.4.1 Supporting auto-detection

The implementation of a new type of library that can be auto-detected needs to register a function with the signature `library::init_function` with `library::addtype`. This gets used by the library class when it attempts to open a file. It is often convenient to provide the library user with a `registerlib` function that handles this registration. For example, for a type of library file that always starts with the line “My Library File”, the following might be used.

```
class mylibrary : public library_base {
public:
  // Allow the user to initialise the library
  static void registerlib(void) {
    library::addtype(&canread);
  }

private:
  static library_base *canread(const string& name) {
    ifstream ifs(name); string first_line;  getline(ifs, first_line);
    if (first_line == "My Library File") return new mylibrary(name);
    return NULL;
  }

  mylibrary(const string &name);
};
```

In addition, the implementation must override the `library_base::good` and `library_base::begin` virtual functions. This is discussed elsewhere.

### 5.4.2 Alternative creation mechanisms

In cases where it is inappropriate to use the auto-detection mechanism an alternative way of initialising `library` with a pointer to a subclass of `library_base` must be provided. To do this, the implementation should provide subclasses of both `library` and `library_base`. The constructor of the subclass of `library` should call down to protected `library::library (library_base* lb)` constrctor with a newly allocated subclass of `library_base`. For example,

```
class mylibrary : public library {
private:
  class impl : public library_base {
    // override virtual functions
  };

  impl *init_library() {
    // Do any necessary initialisation
    return new impl;
  }

public:
  mylibrary() : library( init_library() ) {}
};
```

Usually it is sensible to avoid putting any data members or virtual functions in `mylibrary`, and to put them into `mylibrary::impl` instead. This means that if a `mylibrary` class is sliced to a `library` during a copy (for example when passing to a function taking a `library` by value) the class will still work correctly.

Note that `library::~library` is *not* virtual, and thus, if `mylibrary` is ever allocated on the heap, care is required to avoid deletion via a pointer to its base class.

### 5.4.3 Subclassing library_base

In addition to providing mechanism for opening libraries, implementations of new library types must also override at least the `library_base::good` and `library_base::begin` virtual functions. The implementation of the `good` function is usually trivial, and `begin` only slightly less so. Continuing with the example used in Section 5.4.1 [Supporting auto-detection], page 26, they might be as follows:

```
    class mylibrary : public library_base {
      mylibrary(const string &filename) : ifs(filename) {}

      class entry_type : public library_entry {
        // ...
      };

      const_iterator make_begin() {
        ifs.clear(); ifs.seekg(0, ios::beg); // Rewind the stream
        return const_iterator( this, new entry_type );
      }

      virtual bool good() const { return ifs; }

      virtual const_iterator begin() const {
        return const_cast< mylibrary* >( this )->make_begin();
      }

      ifstream ifs;
    }
```

Notice the slight awkwardness of the `const_cast` in `begin`. This is necessary because the `begin` function is sematically constant—it does not change the visible state of the library—but it needs to rewind *ifs*. The stream's position is not part of the visible state of the library because the `library::const_iterator` is only an Input Iterator and so only one of them can ever be present at any time.

### 5.4.4 Subclassing library_entry impl

The last step of implementing a library is to provide a class that implements the `library_entry::impl` interface, and most importantly, the `library_entry::impl::readentry` function. There are various possible implementation strategies: the simplest is to get `readentry` to parse the whole of the library entry, store the name, place notation and so on in data members, and then have the `name`, `pn`, etc. functions return these. This is illustrated below:

```
class mylibrary : public library_base {
  class my_library::entry_type : public library_entry::impl {

    // Do a deep-copy of the entry
    virtual library_entry::impl* clone() const {
      return new entry_type(*this);
    }

    // Parse the entry
    virtual bool readentry( library_bas& lb ) {
      ifstream &ifs = dynamic_cast< cclib& >( lb ).ifs;
      if (!ifs) return false; // Have we reached the end of file?
      string entry; getline( ifs, entry );

      // Assume the entry is in the format Method Name:Place Notation
      string::size_type p = linebuf.find(':');
      name_ = string( entry, 0, p );
      pn_   = string( entry, p+1, string::npos );
      return true;
    }

    // Access parts of the entry
    virtual string name() const { return name_; }
    virtual string pn() const   { return pn_;   }

    string name_, pn_;
  };

  ifstream ifs;
};
```

An alternative implementation strategy would be to have `readline` store the whole line in data member (i.e. make *entry* a data member), and then get `name` and `pn` to generate these on-the-fly from the *entry* string.

# 6 Method and Touch Proving

## 6.1 The `proof` class

The proof class provides a simple way of checking that a block of rows are true. It provides functions for finding out where a particular block has failed. Additionally, it will can check blocks that are more than one extent long.

It is a template class, defined as `template <class RowIterator> class proof;`. You can therefore decide on your own method of generating rows, as long as you can provide an iterator (see the STL) for it.

It will check ALL rows given; therefore, if rounds are at the start and end, then one of them should be removed before passing to the object.

The proof class only stores the results: it does not make a copy of the actual rows. However, pointers to false rows will be stored, so don't destroy your rows before getting all the proof results. This may be changed in a later release.

Note: To avoid problems, make sure that all the rows have the same number of bells in them, otherwise 12345 will not 123456.

### 6.1.1 Derivation

This class is not derived from any other classes.

```
template <class RowIterator> class proof;
```

### 6.1.2 Constructors

Where the variable *qp* is passed into a function, if this is set to true, the proof class will only check until the row blocks are found to be false, as soon as this is the case, the function will end and store the true/false result. No details will be stored about what lines the row block failed on.

`proof::proof` (*void*); [Constructor]
: Default Constructor. Sets trueness to false. Use the prove function to provide the rows to prove.

`proof::proof` (*RowIterator* `first`, *RowIterator* `last`, *bool* `qp` = *false*); [Constructor]
: This checks for repeated rows from first to last. Expects up to 1 extent only. Result is stored in class for later retrieval.

`proof::proof` (*RowIterator* `first`, *RowIterator* `last`, *const int* `max`, *bool* `qp` = *false*); [Constructor]
: This checks for repeated rows from first to last. Expects up to *max* extents. Result is stored in class for later retrieval.

`proof::proof` (*RowIterator* `true_first`, *RowIterator* `true_last`, *RowIterator* `unknown_first`, *RowIterator* `unknown_last`, *bool* `qp` = *false*); [Constructor]
: This constructor takes two blocks of rows as it parameters. *true_first* to *true_last* are assumed to be a block of rows that are true. No trueness checking is performed on these rows. The function looks at rows *unknown_first* to *unknown_last* comparing them to the true block, and to the unknown block to see if they are repeated anywhere. The failure information is reset before the analysis takes place, hence any falseness details obtained will be for the new block only.

`proof::proof` (*RowIterator* **`true_first`**, *RowIterator* **`true_last`**, *RowIterator* **`unknown_first`**, *RowIterator* **`unknown_last`**, *const int* **`max`**, *bool* **`qp`** *= false);* [Constructor]

This constructor acts like the previous, except it will expect up to *max* extents (and hence repititions).

## 6.1.3 Functions

Where the variable *qp* is passed into a function, if this is set to true, the proof class will only check until the row blocks are found to be false, as soon as this is the case, the function will end and store the true/false result. No details will be stored about what lines the row block failed on.

`bool proof::prove` (*RowIterator* **`first`**, *RowIterator* **`last`**, *bool* **`qp`** *= false);* [Function]

This checks for repeated rows from first to last. Expects up to 1 extent only. Full result is stored in class for later retrieval, but trueness of rows is returned. This function will erase any falseness details stored in the class.

`bool proof::prove` (*RowIterator* **`first`**, *RowIterator* **`last`**, *const int* **`max`**, *bool* **`qp`** *= false);* [Function]

This checks for repeated rows from first to last. Expects up to *max* extents. Full result is stored in class for later retrieval, but trueness of rows is returned. This function will erase any falseness details stored in the class.

`bool proof::prove` (*RowIterator* **`true_first`**, *RowIterator* **`true_last`**, *RowIterator* **`unknown_first`**, *RowIterator* **`unknown_last`**, *bool* **`qp`** *= false);* [Function]

This function takes two blocks of rows as it parameters. *true_first* to *true_last* are assumed to be a block of rows that are true. No trueness checking is performed on these rows. The function looks at rows *unknown_first* to *unknown_last* comparing them to the true block, and to the unknown block to see if they are repeated anywhere. The failure information is reset before the analysis takes place, hence any falseness details obtained will be for the new block only.

`bool proof::prove` (*RowIterator* **`true_first`**, *RowIterator* **`true_last`**, *RowIterator* **`unknown_first`**, *RowIterator* **`unknown_last`**, *const int* **`max`**, *bool* **`qp`** *= false);* [Function]

This function acts like the previous, except it will expect up to *max* extents (and hence repititions).

`const failinfo& proof::failed` (*void*) *const;* [Function]

This function simply returns the data detailing where the touch has failed.

## 6.1.4 Operators

`operator proof::int` (*void*) *const;* [Operator]

This returns true (or 1) if the touch is true.

`int proof::operator!` (*void*) *const;* [Operator]

This returns true (or 1) if the touch is not true.

## 6.1.5 Output

Output of the proof class is achieved in one of two ways. Either, the functions proof::int and failed() are used, or an overloaded ostream operator for a fixed format.

`template <class RowIterator> ostream& operator<<` (*ostream&* `o`, *const proof<RowIterator>&* `p`); [Operator]

Outputs to the ostream, *o*, if *p* is true or false in a predefined format. If p is false, then additional information regarding where is also output.

### 6.1.6 How falseness is stored

When the proof class finds two or more rows that are the same, it stores the details in a private variable, *where*. This can be retrieved by the function `failed`. The variable is defined as:

```
typedef list<linedetail> proof::failinfo;
failinfo where;
```

linedetail is a struct defined as:

```
struct linedetail {
 row _row;
 list<int> _lines;
};
```

When a row is repeated more than once, a new linedetail item is added to *where*. In the linedetail data is stored a pointer, *_row*, to the first of the matching rows, and the row numbers are put into *_lines*. If the first row given to the class matches the 11th row, then 0 and 10 will be placed into *_lines*. If there are three or more rows that all match, then the relevant numbers will be placed into the *_lines* list without repeating any.

### 6.1.7 How it works

There are two ways that the class can prove blocks of rows depending on if the number of rows is less or greater than an extent. In both methods, the procedures are carried through to the end to allow the user to see where the block of rows is false in all places, not just the first.

### 6.1.8 Single Extent

If the single extent version of the function `proof::prove` is called, then we follow this procedure.

Here we look at the first row and compare it to the rest. Then we proceed to the second and compare that to the rest (starting from the third, and going to the end). We then continue this until we have completed all the rows. If two rows match the details are stored and the trueness is set to false.

### 6.1.9 Multiple Extents

If the multiple extent version of the function `proof::prove` is called, then we follow this procedure.

Now we proceed through the list just once. Each row is put into a multimap, if at any time the amount of rows in this map exceed the maximum number of extents then the details are added into the failed information and the trueness is set to false.

# 7 Analysis of Music

The music class provides analysis for music within a set of rows. The class has been designed around providing a regular-expression style matching algorithm. This provides flexibility in the way the user can enter their own music. Additionally, for each 'regular-expression' a score can be allocated. For a full description of the regular expressions the ringing library currently provides see Music Regular Expressions.

The music class processes each row just once, it does not store any rows. It assumes that the first row given is a handstroke row. The class should be able to handle any number of bells that the row class can. Note, that matches for complete rows should be done on a same number of bells basis.

Each time the analysis function is called, the results stored in the class are reset to zero, and then incremented based on the rows given.

## 7.1 Stroke Description

Part of the music class functionality is to allow handstrokes and backstrokes to be specified independently, or together. An enumeration, `EStroke` is provided to enable this to take place.

It is defined as follows:

```
enum EStroke
{
  eHandstroke,
  eBackstroke,
  eBoth
};
```

## 7.2 Music Regular Expressions

The `music_details` class provides the means to describe musical rows to the `music` class. The `music_details` class stores details of the expression to be matched (as a string) and an associated rating, if the user so desires.

Ratings may be positive or negative, this allows a total weighted score to be calculated from the sum of the result scores. For example, 678s may be rated at a +2, but 87s at backstroke rated as -10.

When a match sequence is run via the music class, if a row matches that described by the details provided to a `music_details` object, then the count will be incremented.

The result can either be returned as a raw count, or as a score calculated from the multiplication of the count and the given rating.

If it is specified, then the count or score can be return for just the handstroke, backstroke or both.

Further details relating to the music_details class can be accessed from the menu below:

### 7.2.1 Music Regular Expression Language

The regular expression strings describing rows that can be matched as 'musical' can be specified using the following characters:

Representing a bell:

```
1234567890ETABCDFGHJKLMNPQRSUVWYZ
```

Representing any bell:

```
?
```

Representing any bells in any order:

`*`

Representing one of several bells in a set position:

`[34]`

These constructs can be formed into the following (in all these cases we are matching with a 6 bell method):

```
"12??56" In this case we are wishing to know how many rows occur that are 124356 or 12
"135246" In this case we are wishing to know how many times queens occur.
"*456" In this case we wish to know how many rows there are that finish in 456.
"654*" In this case we wish to know how many rows there are that start with 654.
"*456*" In this case we wish to know how many rows have 456 in the middle of them, note
"*[56]78" in this case we wish to know how many rows end in 578 or 678.
```

### 7.2.2 Music Details class

The music_details class allows the specification of music via a regular expression to the music class. Although functions are provided to return the score, they are mainly intended for use by the music class itself. It should be noted that the music class takes a copy of the music_details object rather than a pointer.

`music_details` (*const string &e* = `""`, *const int &s = 1*); [Constructor]
`music_details` (*const char *e, const int &s = 1*); [Constructor]

Creates the object and sets the initial expression to e and the score to use to s. By default the score is 1. If the specified expression is invalid, the class will throw an `music_details::invalid_regex` exception (if they are enabled) or reset the expression to `""`.

`~music_details ();` [Destructor]

Destructor. Doesn't do much in this case.

`void set` (*const string &e, const int &s = 1*); [Function]
`void set` (*const char *e, const int &s = 1*); [Function]

Sets the expression to e and the score to s. It will also reset the counts of rows matched.

`string get` () *const;* [Function]

Returns the currently stored expression.

`unsigned int possible_matches` (*const unsigned int &bells*) *const;* [Function]

Returns the maximum number of possible matches (in one true extent) for the currently specified expression. *bells* must be specified as the `music_details` class has no knowledge of this under normal circumstances.

`int possible_score` (*const unsigned int &bells*) *const;* [Function]

Returns the maximum possible score (based on one true extent) for the currently specified expression. *bells* must be specified as the `music_details` class has no knowledge of this under normal circumstances.

`unsigned int count` (*const EStroke& = eBoth*) *const;* [Function]

Returns the number of rows matched based on the stroke that is passed in. If eBoth is passed, then the result will be the handstroke count plus the backstroke count.

`int total` (*const EStroke& = eBoth*) *const;* [Function]

Returns the total score based on the stroke that is passed in. If eBoth is passed, then the result will be the handstroke score plus the backstroke score.

`int raw_score` *const;* [Function]

Returns the score previously specified to the object.

## 7.3 Using the Music Class

The basic process of using the music class is to create it, specify to it the music that you wish to look for in a set of rows, tell it to process the rows and then obtain the results.

We now describe this in more detail. First create an instance of the music class, specifying to it the number of bells you wish to work with:

```
music mu(5);
```

Now specify the music that you are looking for via the music details class (see Section 7.2.1 [Expression Language], page 32 for how to do this). The music class behaves like a `vector<music_details>` structure so you can access the elements individually before and after. To add a music specification, do something like this:

```
music_details md("*45");
mu.push_back(md);
```

If you want, more can be added in the same way or by modifying md and pushing the new structure onto the back of the music list:

```
md.set("*345*");
mu.push_back(md);
```

Now you can supply the music class with rows to search against the specified music (note pbdoubles is a vector<row>)

```
mu.process_rows(pbdoubles.begin(), pbdoubles.end());
```

Now you can obtain the results from the music class. You can do this in several ways. The first is to obtain the overall total score, for both strokes, or for one:

```
cout << "Total Score: " << mu.get_score() << endl;
cout << "Total Score (Backstroke): " << mu.get_score(eBackStroke) << endl;
```

The second way is to obtain results for individual items:

```
music::const_iterator k;

for (k = mu.begin(); k != mu.end(); k++)
  cout << "Total for " << k->get() << ": " << t->count() << " : " << k->total() << endl
```

This produces three columns, the first is the music expression that was searched for, the second is the number of matches, the third is the number of matches based on the score. See the Section 7.2.2 [Music_Details class], page 33 for more details on these functions.

## 7.4 Music Class definition

The `music` class is the main control of the music classes; it stores `music_details` objects to match music against and controls the construction of the `music_node` tree (see Section 7.5 [music node class definition], page 36.

The class must know the number of bells in the rows that are to be examined before the `music_class` objects are passed to it; this allows it to create trees of the correct depth and structure to store the regular expressions in search format.

The `music_detail` objects are stored in a `vector<music_details>` structure, this is accessible via the functions provided (detailed below).

### 7.4.1 Derivation

This class is not derived from any other classes.

```
class music;
```

### 7.4.2 Constructors

`music::music` (*const unsigned int &b = 0);* [Constructor]

This constructor initialises the number of bells to zero. If you don't specify bells here, you should specify the number of bells by the `set_bells()` function before adding expressions to be matched.

### 7.4.3 Type Definitions

This section details the type definitions that the music class creates for the user to use.

`vector<music_details> music::mdvector;` [typedef]

This type defines the storage vector that records the expressions to be matched and is also used to obtain the results.

`vector<music_details>::iterator music::iterator;` [typedef]
`vector<music_details>::const_iterator music::const_iterator;` [typedef]

This type defines a general iterator that can be used to iterate through the stored expressions.

`vector<music_details>::size_type music::size_type;` [typedef]

This type defines the type of variable returned from the `size()` function.

### 7.4.4 Functions

`void push_back` (*const music_details &);* [Function]

This function adds a `music_details` expression onto the vector of expressions to be matched. It will also add it to the tree ready for processing against rows.

`music::iterator begin` (*void);* [Function]
`music::const_iterator begin` (*void*) *const;* [Function]

These functions return an iterator that points at the beginning of the expression list.

`music::iterator end` (*void);* [Function]
`music::const_iterator end` (*void*) *const;* [Function]

These functions return an iterator that points past the end of the expression list.

`music::size_type size` (*void);* [Function]

This function returns the number of expressions currently stored.

`void set_bells` (*const unsigned int &b);* [Function]

This function sets the number of bells that will be in the matched rows to `b`.

`template <class RowIterator> void process_rows` (*RowIterator first, RowIterator last, bool backstroke = false);* [Function]

This function first resets the all of the music counts, and then processes the supplied rows. If backstroke is true, then a backstroke is assumed to be the first row, otherwise it will be noted as a handstroke. The stroke will be toggled for every row processed.

`int get_score` (*const EStroke& = eBoth);* [Function]

Returns the score for music matched at the specified stroke. If eBoth is specified as the stroke, then scores for both strokes are added together and returned.

`unsigned int` *get_count* (*const EStroke& = eBoth);* [Function]

Returns the total number of matches for all music expressions at the specified stroke. If eBoth is specified, then the counts for both strokes are added and returned.

`int get_possible_score` (*void);* [Function]

This function returns the maximum possible score that could be obtained matching the expressions currently stored against a true extent on the number of bells previously specified.

## 7.5 Music Node Class definition

The `music_node` class provides the main expression matching routines for the `music` class. A `music` object stores one `TopNode` object, (of type `music_node`), when expressions are added to the `music` object, via a `music_details` class, a tree is formed under the `TopNode` object.

The `music_node` class is normally hidden from the user by the `music` class, however, it's functionality is described here to aid maintainance and understanding.

### 7.5.1 Full Description

Each node in the tree stores the expressions that finish at the node, and pointers to nodes below them, if any have been created. Note, for any node there are a maximum of n pointers, where n is the number of bells, plus one; The extra case in this instance is for a ? or * expression to match any bell.

When a row is to be matched against the expressions stored, each bell in the row is examined individually. If a pointer to a node for the same number as the current bell exists, the matching passes onto the node pointed at. For all cases, if an 'any' pointer exists, then the matching will be passes onto the any node.

When the expression matching comes up against a node where an expression is specified to finish at that node, then the counter for matches against the node is incremented. If there are no matches to the current bell, and the any pointer does not exist, then the matching simply finishes and returns back up the tree.

This is probably better explained by means of an example. Suppose the expression '`321654`' is added to the TopNode. If '`+`' is a node, and '`|n`' is a pointer between the nodes, where n is the number of bell pointed to, then the following tree is formed:

```
TopNode
|3
+
|2
+
|1
+
|6
+
|5
+
|4
+ '321654' finishes here.
```

When the row '`326154`' is attempted to be matched, the following happens:

```
TopNode -> Pointer to 3 exists, so go to the next node.
|3
+       -> Pointer to 2 exists, so go to the next node.
|2
+       -> Pointer to 6 (or to any node) does not exist, so terminate.
|1
```

Now suppose '`321*`' and '`?2*`' is added to the list, the following tree is formed:

```
              TopNode
                 |
-------------------------------
|3                            |any
+                             +
|2                             |2
+                              + '?2*' finishes here.
|1
+ '321*' finishes here.
|6
+
|5
+
|4
+ '321654' finishes here.
```

Now, when the row '321654' is put through the matching process, the following happens. Taking first the pointer to bell '3' option of TopNode:

```
TopNode -> Pointer to 3 exists, so go to the next node.
|3
+       -> Pointer to 2 exists, so go to the next node.
|2
+       -> Pointer to 1 exists, so go to the next node.
|1
+       -> '321*' finishes here so increment, and then as pointer to 6
           exists, carry on to the next node.
|6
+       -> Pointer to 5 exists, so go on.
|5
+       -> Pointer to 4 exists, so go on.
|4
+       -> '321654' finishes here so increment matches. No further
           pointers, so go back up the tree checking for 'any' pointers
```

When the matching gets back to the top of the tree and finds the any pointer in the top node, it will follow it as below:

```
TopNode -> Any pointer exists, so follow it
|any
+       -> Note, we are now on the second bell of the row, which in our case
           is '2', pointer exists, so follow it.
|2
+       -> '?2*' finishes here so increment matches.
```

From these examples it can be seen that an optimisation is performed when there is a single '*' left, which saves the matching algorithm going through exvery any node.

## 7.5.2 Derivation

This class is not derived from any other classes.

```
class music_node;
```

## 7.5.3 Constructors and Destructors

`music_node::music_node` (*void*); [Constructor]
`music_node::music_node` (*const unsigned int &b*); [Constructor]

The constructor initialises the object and sets the number of bells in the rows that will be matched. If the default constructor is used, then the `set_bells()` function should be used to perform this action.

`music_node::~music_node` (*void*); [Destructor]

The destructor will go through the tree structure, deleting the sub-nodes of the current node.

### 7.5.4 Type Definitions

`map<unsigned int,` *music_node*> BellNodeMap;* [typedef]

This map is used to form the structure of the tree. The branches are stored in the structure, with the unsigned int being an index by bell number. Index 0 is for matching any bell. Note, where a node of the tree has no branches coming off of it (because it is the end, or there is nothing else to match against) the map will be empty.

`BellNodeMap::iterator BellNodeMapIterator;` [typedef]

This type definition is used to provide a shorter type name to iterate the map described above.

`vector<unsigned int>` *DetailsVector;* [typedef]

This type definition defines the vector that is used to store the id of the music_details expressions that end at this node.

`DetailsVector::iterator DetailsVectorIterator;` [typedef]

This type definition is used to provide a shorter type name to iterate the details vector defined above.

### 7.5.5 Functions

`void set_bells` (*const unsigned int &b*)*;* [Function]

This function is used to set the number of bells that will be in the rows to be matched against. This provides a way for the algorithms to know when they are finished processing the expressions or row.

`void add` (*const music_details &md, const unsigned int &i, const unsigned int &key, const unsigned int &pos*)*;* [Function]

This function is used to add the supplied expression (`md`), to the tree. The location in the expression to look at is supplied by `i`, the `key` is the reference number that the `music` class stores the `md` expression against, and the `pos` is the number of bells deep we are in the tree.

Depending on the item in the expression at index `i`, the algorithm will do different things, normally, it will call the `add_to_subtree(...)` function which in turn will usually call this function again; sometimes it will call itself (in the case of a ‘`'*'`’) to move along the current expression without necessarily adding it to a subtree. Hence, the `add(...)` function is an iterative one.

`void add_to_subtree` (*const unsigned int &place, const music_details &md, const unsigned int &i, const unsigned int &key, const unsigned int &pos, const bool &process_star*)*;* [Function]

This private function is used to create a branch (if required) and add an expression to a branch of a node. If it cannot allocate memory for the new node, it will either throw a `memory_error` exception (if enabled), or it will simply return without going any further.

The `place` parameter provides the branch number (= bell) to create and add to, the `process_star` parameter is used to indicate if the number of ‘`*`’s remaining should be processed or ignored. The rest of the parameters are as for the `add(...)` function.

`void match` (*const row &r, const unsigned int &pos, vector*`<`*music_details*`>` *&results, const EStroke &stroke*)*;* [Function]

This function is iterative and is used to match the expressions stored to a row, `r`. If a node is found to have an expression ending there, the number of counts for that expression are incremented according to the specified `stroke`. The `pos` parameter is used to specify the current position in the row `r`, and the `results` parameter is the expressions for which the indexes have previously been stored, and against which the count should be incremented as appropriate.

### 7.5.6 Exceptions

There is one exception defined for this class:

```
struct memory_error : public overflow_error() {
  memory_error()
};

music_node::memory_error::memory_error()
  : overflow_error("Not enough memory to allocate to music_node item") {}
```

This is used to indicate that there is not enough memory to add a music_node item.

# 8 Producing printed output

The Ringing Class Library provides a flexible mechanism for producing printed output of rows, methods and lines. The way this mechanism is designed means that the low-level details of printing are hidden from the programmer. Currently the classes in the library support output in PostScript and PDF formats, and the modularity of the printing system makes it easy to extend the library to provide other output formats.

Basic printing is provided in two classes: the `printpage` class and the `printrow` class. The first of these deals with placing text and other objects directly on the page; the second allows for formatted output of rows and lines. These classes are both pure virtual classes, which are specialised by subclasses to allow for different output formats.

Printing of whole methods is accomplished by the `printmethod` object. This is not as flexible as the more generic functions of the lower-level printing objects, but exists to provide a quick and easy way to print out methods in a moderately variable format.

There are certain rules which should be followed when using the printing classes. For a single `printpage` object, only one `printrow` object may exist at any one time. All `printrow` objects associated to a `printpage` object must be destroyed before starting a new page, and before destroying the `printpage` object. Failure to observe these rules may result in unusable output.

## 8.1 Auxiliary classes

The printing mechanism of the Ringing Class Library declares several auxiliary classes, for specifying printing parameters in a device-independent way. This section describes those classes.

### 8.1.1 The `dimension` class

The `dimension` class is used extensively by the printing classes to specify physical dimensions on the page. The class has three public data members and one public enumerated type:

```
class dimension {
public:
  enum units {
    points, inches, cm, mm
  };

  int n, d;
  units u;
}
```

The integers `n` and `d` hold the numerator and denominator respectively of the dimension. The member `u` defines which units the dimension is given in. For example, a `dimension` object representing three quarters of an inch would have `n` equal to 3, `d` equal to 4, and `u` equal to `dimension::inches`. Points here are PostScript points, which are equal to 1/72 of an inch.

The `dimension` class has several member functions.

`dimension::dimension` (*void*); [Constructor]

This constructs a dimension equal to 0.

`dimension::dimension` (*int* `i`); [Constructor]

This constructs a dimension equal to *i* points.

`dimension::dimension` (*int* `i`, *int* `j`, *dimension::units* `u`); [Constructor]

This constructs a dimension equal to the fraction *i*/*j* in units *u*.

`dimension dimension::operator-` (*void*) [Operator]

This operator returns the negation of the dimension.

`dimension& dimension::operator*=` (*int* `i`) [Operator]
`dimension dimension::operator*` (*int* `i`) [Operator]
These operators multiply the dimension by the integer *i*.

`dimension& dimension::operator/=` (*int* `i`) [Operator]
`dimension dimension::operator/` (*int* `i`) [Operator]
These operators divide the dimension by the integer *i*.

`bool dimension::operator==` (*int* `i`)*;* [Operator]
`bool dimension::operator!=` (*int* `i`)*;* [Operator]
These operators are only useful for comparing a dimension to 0.

`void dimension::reduce` (*void*)*;* [Function]
This function reduces the fraction `n/d` to lowest terms.

`float dimension::in_points` (*void*)*;* [Function]
This returns a floating point numbers representing the dimension in points.

`void dimension::set_float` (*float* `value`*, int* `denom`*, units* `uu`*=points*)*;* [Function]
This function sets the dimension to an approximation of the given floating-point number *value*. Specifically, the denominator is set to *denom*, and the numerator is set to the integer closest to the product of *value* and *denom*.

`void dimension::read` (*const string&* `s`)*;* [Function]
`void dimension::read` (*const char** `s`)*;* [Function]
These functions read a dimension from a string. The string should contain following parts:

- An optional minus sign;
- a number, in either decimal format or as a fraction;
- the units.

For example, the following strings would all be recognised: '`0.4 cm`'; '`1 1/2"`', '`56 pt`'. The allowed unit names are '`pt`', '`points`', '`point`', '`in`', '`inch`', '`inches`', '`"`', '`cm`', '`mm`'.

`string& dimension::write` (*string&* `s`)*;* [Function]
This function writes the dimension to the string *s*.

### 8.1.2 The `colour` class

The `colour` class is a structure which represents a colour. It is defined thus:

```
struct colour { bool grey; float red, green, blue; };
```

The red-green-blue (RGB) colour space is used to specify colours. This is not particularly well suited to printing, where a subtractive colour model such as CMYK is more useful, but RGB is at least very widely used. Colours printed using the Ringing Class Library will probably vary quite a lot from one device to another, but your blue lines will be blue and your red lines will be red.

To specify a colour in a `colour` object, simply set `grey` to `false` and set the other three members to numbers between 0 and 1, inclusive. You can also specify a shade of grey by setting `grey` to `true`; then the `red` element specifies the shade of grey, from 0 (black) to 1 (white).

### 8.1.3 The `text_style` class

The `text_style` class is a structure which represents a text style. It contains a font name, a size in tenths of a point and a colour. It also defines an enumerated type which may be used to describe text alignment.

```
struct text_style {
  string font;
  int size;
  colour col;
  enum alignment { left, right, centre };
};
```

Font names are dependent on the output device; as the only output device currently supported is a PostScript device, the font name should be a PostScript font name such as '`Helvetica-Oblique`' or '`Times-Roman`'.

## 8.2 The `printpage` class

The `printpage` class takes control of producing printed output on a page. To start printing, an object of some subclass of `printpage` is created, which will handle all further printing. For example, to print to a PostScript stream or file, an object of class `printpage_ps` would be created. Once the `printpage` class has been created, it can be used to place text directly on the page. It can also be used to create `printrow` objects for printing rows and lines.

Note that positions on the page are always measured from the bottom left corner.

The `printpage` class is a pure virtual class: the only way to create one is by constructing an object of a derived class.

`void printpage::new_page` (*void*); [Function]

This tells the `printpage` object to finish printing the current page and start a new page.

`virtual void printpage::text` (*const string* `t`, *const dimension&* `x`, *const dimension&* `y`, *text_style::alignment* `a`, *const text_style&* `s`); [Function]

This function places a text string on the page. The string is given in *t* and should consist of printable characters. The position at which the string is to be printed, measured from the bottom left corner of the page, is given by *x* and *y*. The alignment of the text about this point is given by *a*, and the text style is given by *s*.

## 8.3 The `printrow` class

Printing of rows and lines is handled by the `printrow` class and its subclasses. An object of type `printrow` is created, attached to a given `printpage` object. After the object has been created, it can be given rows to print. The rows can be printed in various ways: the actual numbers representing the row may or may not be printed, and lines can be drawn for selected bells.

### 8.3.1 Using the `printrow` class

A `printrow` object is created by one of the following constructors. Note that there are certain options which cannot be changed once the object has been created, so unless the default options are going to be used it is necessary to use the second form of the constructor.

`printrow::printrow` (*const printpage&* `pp`); [Constructor]

This constructor builds a `printrow` object associated to the `printpage` object given by *pp*. The options are initialised to the default options.

`printrow::printrow` (*const printpage&* `pp`, *const printrow::options&* `op`); [Constructor]

This constructor builds a `printrow` object associated to the `printpage` object given by *pp*, and initialised the options to *op*.

Once the `printrow` object has been created, it must be positioned on the page before any rows can be printed.

`void printrow::set_position` (*const dimension&* `x`, *const dimension&* `y`*);* [Function]

This function sets the position of the `printrow` object to the given place on the page, measured from the bottom left. The position given is where the first bell of the next row printed will appear.

`void printrow::move_position` (*const dimension&* `x`, *const dimension&* `y`*);* [Function]

This function is like the preceding one, but the position given by *x* and *y* are taken relative to the current position of the `printrow` object. The current position is either the place where the top left bell of the last column printed appeared, or as set by a `set_position` or `move_position` function call, whichever is more recent.

Certain options (see Section 8.3.2 [Row printing options], page 43) may be changed while the `printrow` object exists.

`const printrow::options& printrow::get_options` (*void*)*;* [Function]

This function returns the current options.

`void printrow::set_options` (*const printrow::options&* `o`*);* [Function]

This function sets the options to those given by *o*. Note that only certain option changes will have any effect once the object has been constructed.

After setting the position, rows can be printed.

`printrow& printrow::operator<<` (*const printrow&* `pr`, *const row&* `r`*);* [Operator]

This operator is used to print a row *r* using the `printrow` object *pr*. This prints the numbers of the row if the `numbers` flag of the options is set, and also draws the lines for any bells which have been specified in the options. The row is printed underneath the previous row.

`void printrow::rule` (*void*)*;* [Function]

This function draws a horizontal rule below the last row printed.

`void printrow::dot` (*int* `i`*);* [Function]

This function draws a small dot at the place where bell *i* was in the last row printed. If *i* is -1, prints dots for all bells which are having lines drawn.

`void printrow::placebell` (*int* `i`*);* [Function]

This function finds what place the bell *i* was at in the last row printed, and then prints this number in a circle to the right of that row.

`void printrow::text` (*const string&* `t`, *const dimension&* `x`, *text_style::alignment* `al`, *bool* `between`, *bool* `right`*);* [Function]

This function prints some text to one side of the last row printed. The text is given in *t* and is printed in the same style as the rows. If the boolean argument *right* is true, the text is printed to the right of the row; otherwise the text is printed to the left of the row. The dimension *x* is a distance from the leftmost or rightmost number in the row, and the text is aligned about this point as given by *al* (see Section 8.1.3 [The text_style class], page 41). If the argument *between* is true, then the text is not aligned vertically with the previous row, but rather appears halfway between the previous row and the next one to be printed. This last option is useful, for example, for printing place notation beside a method.

### 8.3.2 Row printing options

The way in which rows are printed by a `printrow` object is controlled by various options. These options are given by a `printrow::options` object.

```
struct options {
  enum o_flags {
    numbers = 0x01,
    miss_numbers = 0x02
  };
  unsigned int flags;
  text_style style;
  dimension xspace, yspace;
  struct line_style {
    dimension width;
    colour col;
  };
  map<int, line_style> lines;

  void defaults(void);
};
```

The `flags` member contains various flags; bit values are given by the enumerated type `o_flags`, and possible values for `flags` can be obtained by logically ORing these values together. The bits are as follows:

- The `options::numbers` bit controls whether the actual numbers of the rows are printed. If this bit is not set, only lines will be printed.
- If the `options::miss_numbers` bit, then any numbers which would appear underneath lines are missed out. An example of this may be seen in the *Ringing World* diary, where all rows except lead heads are printed with this property.

This option may be changed, with immediate effect, at any point during printing.

The `style` member controls the style in which the rows are printed; for the declaration of the `text_style` class, see Section 8.1.3 [The text_style class], page 41. This option may not be changed during printing.

The `xspace` and `yspace` members give the horizontal space between numbers in a row, and the vertical space between rows, respectively. For the declaration of the `dimension` class, see Section 8.1.1 [The dimension class], page 40. These options may not be changed during printing.

The `lines` member controls which "blue lines" are printed. Each element of the map has an integer, denoting the bell number, and a `line_style` element which defines the style of the line. The two members of `line_style` are a `dimension` object, giving the width of the line, and a `colour` object, giving the colour of the line. This option may be changed in between columns, but not in the middle of a column.

For example, suppose that we wish to print rows in 10-point Helvetica, with spacing of 12 points horizontally and vertically. We want to draw a blue line half a point thick for the second bell. The following code sets up a `printrow::options` object to accomplish this:

```
printrow::options o;
o.flags = printrow::options::numbers;
o.style.size = 100;
o.style.font = "Helvetica";
o.xspace.n = 12;
o.yspace.n = 12;
line_style s; s.width.n = 1; s.width.d = 2; s.width.u = dimension::points;
s.col.grey = false; s.col.red = 0; s.col.green = 0; s.col.blue = 1.0;
o.lines[1] = s;
```

There is one member function of `printrow::options`.

`void printrow::options::defaults` (*void*); [Function]

This function sets up the `printrow::options` object to the defaults as given in the example above.

## 8.4 Printing whole methods

It should be reasonably clear how the `printrow` object might be used to print out numbers and lines for a method. This simple but tedious task is automated by the `printmethod` object. The `printmethod` object is not designed to be as flexible as the rest of the printing mechanism, but rather to accomplish its one aim, that of printing out methods, in as simple a way as possible.

The appearance of the printed method is controlled by setting the public data members of the `printmethod` object. When all the parameters have been set up, the function `printmethod::print()` is used to print the method to a given `printpage` object.

The output consists of a series of *columns*, each of which contains the same number of rows. When one column is full, a new column is started to the right of the last one. A number of columns, side by side, are called a *column set*. When one column set is complete, a new column set is started below the last. Whenever a new column is started, the last row of the previous column is duplicated at the top of the new column.

### 8.4.1 Method printing parameters

The `printmethod` object has many public data members, which control the format of the printed method. These public data members are described here.

`printrow::options printmethod::opt` [Variable]

This member contains the options which are used to print each separate row of the method. For more information on this object, see Section 8.3.2 [Row printing options], page 43. Note that the `miss_numbers` flag is overridden by the setting of the `number_mode` member of the `printmethod` object (see below).

`dimension printmethod::hgap` [Variable]
`dimension printmethod::vgap` [Variable]

These two members control the horizontal gap between columns, and the vertical gap between column sets, respectively. For more information on the `dimension` class, see Section 8.1.1 [The dimension class], page 40.

`int printmethod::total_rows` [Variable]

This integer sets the total number of rows which will be printed. This total does not count those rows which are duplicated because they appear at the end of a column.

`int printmethod::rows_per_column` [Variable]

This integer controls the number of rows printed in each column. In fact each column will contain one more than this number of rows, because the last row of each column is duplicated at the top of the next column.

`int printmethod::columns_per_set` [Variable]

This integer controls the number of columns which appear in each horizontal column set, that is, the number of columns across the printed output.

`int printmethod::sets_per_page` [Variable]

This integer controls the number of column sets which appear on each page, that is, the number of columns down the printed page. If the total number of columns printed exceeds this numbers, more than one page of output will be produced.

`dimension printmethod::xoffset` [Variable]
`dimension printmethod::yoffset` [Variable]

These two members position the output on the page. Together, they contain the position of the top left bell of the top left column of the output. measured from the bottom left hand corner of the page. For more information on the `dimension` class, see Section 8.1.1 [The dimension class], page 40.

`list<pair<int, int> > printmethod::rules` [Variable]

This member controls the printing of horizontal rules in the output. Each entry in the list is a pair $(a, b)$ of integers; this entry means that a rule will be drawn after the $a$th row in each lead, and every $b$ rows after that. For example, the usual setting for a treble dodging method would be the pair (4,4), indicating that rules should be drawn every four rows, starting with a rule after the fourth row. Similarly the usual setting for Stedman would be the pair (3,6).

`printmethod::number_mode_t printmethod::number_mode` [Variable]

This member sets the behaviour with regard to missing numbers underneath lines. There are four possible settings, described by an enumerated type:

```
enum number_mode_t { miss_never, miss_always,
                     miss_column, miss_lead };
```

The meaning of these options is as follows:

- `miss_never`: numbers are always drawn;
- `miss_always`: numbers are never drawn when they appear underneath a line;
- `miss_column`: numbers are only drawn underneath lines in the first and last rows of a column;
- `miss_lead`: numbers are only drawn underneath lines at lead heads.

`printmethod::pn_mode_t printmethod::pn_mode` [Variable]

This member controls the printing of place notation next to the method. There are three possible settings, described by an enumerated type:

```
enum pn_mode_t { pn_none, pn_first, pn_all };
```

These options are interpreted as follows:

- `pn_none`: place notation is never printed;
- `pn_first`: place notation is printed to the left of the first lead of the method and, if the method is symmetrical about the half lead, only the first half of the place notation and the lead end are printed;
- `pn_all`: full place notation is printed to the left of every lead.

`int printmethod::placebells` [Variable]

This integer controls whether place bells are printed, and for which bell. If this member contains an integer greater than or equal to zero, then place bells are printed for the corresponding bell. To suppress the printing of place bells, set this member to `-1`.

## 8.4.2 Using the `printmethod` object

Before printing a method using a `printmethod` object, two things must be set up: the object must be given a method to print, and the parameters for printing described in the previous section must be initialised.

The class has two constructors:

`printmethod::printmethod` (*void*); [Constructor] `printmethod::printmethod` (*const method&* `m`); [Constructor]

These two constructors create a `printmethod` object. In the first case, no method is specified; a method will have to be given later, before anything can be printed. In the second case, a reference to the method *m* is stored for printing after the parameters have been set up.

After the object has been created, the method can be queried and changed with these functions:

`const method& printmethod::get_method` (*void*); [Function]

This returns the method referred to by this `printmethod` object.

`void printmethod::set_method` (*const method&* `m`); [Function]

This function stored a reference to the method *m* in the `printmethod` object, for subsequence printing.

The parameters for printing, described in the previous section, may be manipulated with several member functions.

`void printmethod::defaults` (*void*); [Function]

This function sets the parameters to default values, which relate to the method to be printed; so it should not be called before a method has been given to the `printmethod` object. Specifically, the parameters are set as follows:

- Numbers are to be printed in 10-point Helvetica, and the spacing between bells is 12 points both horizontally and vertically;
- Each column contains a single lead; an entire plain course of the method will be printed, with leads side by side;
- The output is positioned at the bottom left of the page;
- Numbers will be missed underneath lines, except for lead heads;
- Place notation will be printed to the left of the first lead only;
- Each hunt bell is shown with a red line, a quarter of a point thick;
- The smallest working bell which makes a place over the lead end is shown with a blue line, half a point thick, and place bells are shown for this bell. If no working bell makes a place over the lead end, the smallest working bell is chosen instead.

`void printmethod::fit_to_space` (*const dimension&* `width`, *const dimension&* `height`, *bool* `vgap_mode`, *float* `aspect`); [Function]

This function scales and arranges the method to fit in the given space, as specified by *width* and *height*. The parameters which are calculated by this function are `rows_per_column`, `columns_per_set`, `hgap`, `vgap`, `opt.style.size`, `opt.xspace` and `opt.yspace`. Two possible layouts are possible: if *vgap_mode* is `false`, then only one column set is used, with each column containing a whole number of leads; if *vgap_mode* is `true`, then each column contains precisely one lead, and as many column sets as necessary are used. The effect of this is that, in the first case, the second lead appears below the first one; in the second case, the second lead appears to the right of the first one. The last argument *aspect* sets the ratio of `opt.xspace` to `opt.yspace`.

The chosen layout is that which, given the constraints specified, allows the largest of output.

`float printmethod::total_width` (*void*); [Function]
`float printmethod::total_height` (*void*); [Function]

These functions give the total width and height, respectively, in points of the printed output using the current parameters.

## 8.5 Printing in PostScript

The Ringing Class Library provides subclasses of the `printpage` and `printrow` classes for producing output in the PostScript language. These classes are declared in the '`ringing/print_ps.h`' header file. The output conforms to the Document Structuring Conventions (DSC) version 3.0 and uses only Level 1 language.

The `printpage_ps` class is derived from `printpage`, and handles all output to a PostScript stream. The new functions in this class are constructors and some PostScript-specific operations.

`printpage_ps::printpage_ps` (*ostream&* `o`); [Constructor]

This creates a new PostScript printing object, which will send its output to the stream *o*.

`printpage_ps::printpage_ps` (*ostream&* `o`, *const dimension&* `page_height`); [Constructor]

This creates a new PostScript printing object, which will send its output to the stream *o*. Each page will be printed with landscape orientation; because of the way PostScript works, this means that the page height must be given.

`printpage_ps::printpage_ps` (*ostream* `o`, *int* `x0`, *int* `y0`, *int* `x1`, *int* `y1`); [Constructor]

This creates a new object which will write an Encapsulated PostScript (EPS) file to the stream *o*. The bounding box for the EPS file is specified in points by the arguments *x0*, *y0*, *x1* and *y1*. This bounding box has no effect on where you can print things using this object, but other applications manipulating the EPS file may rely on the bounding box in order to work properly.

The `printrow_ps` class is for printing rows to a `printpage_ps` object. You will probably never have to deal with this class directly, as it should be created and dealt with by the interface for the `printrow` class.

## 8.6 Printing to PDF files

Subclasses of the `printpage` and `printrow` classes are also provided for providing output in Adobe Portable Document Format (PDF). These classes are declared in the `print_pdf.h` header file.

Some specific issues must be considered when creating PDF files. Firstly, although the classes support output to an arbitrary stream, some care must be taken. This is because PDF files contain tables which cross-reference different parts of the file; and these tables therefore contain byte offsets into the file. If you use the PDF printing classes to write to a text-mode file on some system, such as Microsoft Windows, which convert single line feeds to carriage-return and line-feed pairs, then this byte count will be incorrect. The safe way round this is to **always open PDF output streams in binary mode**.

The second specific issue which arises when creating a PDF file is that of fonts. In order to position text correctly on the page, the class library needs to know the metrics of every character in each font used. As a result, it is only possible to use the standard 14 PDF fonts when printing from the Ringing Class Library. These standard fonts are:

- Courier, Courier-Bold, Courier-Oblique, Courier-BoldOblique
- Helvetica, Helvetica-Bold, Helvetica-Oblique, Helvetica-BoldOblique
- Times-Roman, Times-Bold, Times-Italic, Times-BoldItalic
- Symbol
- ZapfDingbats

The `printpage_pdf` class is derived from `printpage`. The only new functions defined in this class are its two constructors:

`printpage_pdf::printpage_pdf` (*ostream& `o`, const dimension& `width`, const dimension& `height`, bool `l` =false);* [Constructor]

This constructor creates an object for writing a PDF file to the output stream *o*. The page width and height are set to *width* and *height* respectively. If the Boolean parameter *l* is set, the page will be printed in landscape format.

`printpage_pdf::printpage_pdf` (*ostream& `o`, bool `l` =false);* [Constructor]

This constructor works as the previous one, but the page size is set to A4.

The subclass of `printrow` for printing to a PDF file is called `printrow_pdf`; but as an object of this class will always be created through the parent `printpage` object, the programmer should never need to deal with the `printrow_pdf` class.

# Appendix A Copying this manual

## A.1 GNU Free Documentation License

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other written document *free* in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

   The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

   A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor,

### A.1.1 ADDENDUM: How to use this License for your documents

# Index

## A

algebra of rows . . . 5

## B

`bell` class . . . 6

## C

change . . . 5, 6
`change` class . . . 9
cycles . . . 5

## E

eBackstroke . . . 32
eBoth . . . 32
eHandstroke . . . 32
EStroke . . . 32

## F

facets . . . 18
falseness storage . . . 31
FDL, GNU Free Documentation License . . . 50

## H

How proof works . . . 31

## I

installation . . . 2
introduction . . . 1

## L

library . . . 16
`library` class . . . 16
`library` internals . . . 23
`library::const_iterator` class . . . 23
`library_base` class . . . 22
`library_base::const_iterator` class . . . 23
`library_entry` class . . . 18
`library_entry` internals . . . 24
`library_entry::impl` class . . . 25

## M

method . . . 13
`method` class . . . 13
method library . . . 16
Music Analysis . . . 32
music class . . . 32, 34
Music Expressions . . . 32
music_details . . . 33
music_node . . . 36
Musical Rows . . . 32

## O

order . . . 5

## P

parity . . . 5
`permute` function . . . 11
place notation . . . 6
printing . . . 40
`proof` class . . . 29
proving rows . . . 29

## R

Regular Expressions . . . 32
row . . . 5
row algebra . . . 5
`row` class . . . 7
row operations . . . 5
row properties . . . 5
`row_block` class . . . 12

## S

sign . . . 5
Specifiying Musical Rows . . . 32
Standard Template Library . . . 2
STL . . . 2
swap . . . 6

## T

transposition . . . 5

## X

XML libraries . . . 19
`xmllib` class . . . 20