Mxx_ru

User manual

by Yauheni Akhotnikau

Contents

1 Introduction **6**
- 1.1 About Mxx_ru . . . 6
- 1.2 Features . . . 6
- 1.3 A bit of history . . . 7
- 1.4 Thanks . . . 7

2 Installation **8**
- 2.1 Prerequisites . . . 8
- 2.2 Installing Mxx_ru . . . 8
  - 2.2.1 Setting up Mxx_ru for C/C++ projects . . . 9

3 Versions Mxx_ru **11**
- 3.1 Version 1.6 . . . 11
- 3.2 Version 1.5 . . . 11
- 3.3 Version 1.4 . . . 11
- 3.4 Version 1.3 . . . 12
- 3.5 Version 1.2 . . . 14

4 Examples **15**
- 4.1 Simple application . . . 15
- 4.2 Application with static library . . . 16
- 4.3 Application with shared and static libraries . . . 18
- 4.4 Application with shared and static libraries in different folders . . . 21
  - 4.4.1 "say" Subproject . . . 22
  - 4.4.2 "inout" Subproject . . . 23
  - 4.4.3 "main" subproject . . . 24
  - 4.4.4 build.rb file . . . 25
  - 4.4.5 Building a project . . . 26
  - 4.4.6 Building a single subproject only . . . 26
- 4.5 Cleaning . . . 26
- 4.6 Rebuilding . . . 27

5 The basic idea **28**
- 5.1 More details about MxxRu::AbstractTarget . . . 28
  - 5.1.1 A names for files created by target . . . 29
  - 5.1.2 Subtargets . . . 29
  - 5.1.3 Project aliases . . . 31
  - 5.1.4 Source code generators . . . 31

5.2 More details about MxxRu::setup_target 33

6 Mxx_ru for C/C++ projects **34**
6.1 Introduction 34
6.1.1 Toolset conception 34
6.1.2 obj_placement conception 36
6.1.3 Targets for C/C++ projects 38
6.1.4 An order of build/clean execution 40
6.1.5 Runtime modes 41
6.1.6 Local, global and upspread parameters 42
6.1.7 mxx-cpp-1 argument 43
6.1.8 mxx-cpp-no-depends-analyzer argument 44
6.1.9 mxx-cpp-extract-options 44
6.2 Mxx_ru setup for C/C++ projects 45
6.3 Toolset access 45
6.4 Runtime mode selection inside project file 46
6.5 Runtime library type selection 46
6.6 Multi-threading mode selection 46
6.7 RTTI mode selection 47
6.8 Setting target file name 48
6.8.1 target_root method 48
6.8.2 target method 48
6.8.3 target_prefix method 48
6.8.4 target_ext/target_suffix method 49
6.8.5 implib_path method 49
6.8.6 Examples 49
6.9 Selecting an application type (GUI/Console) 50
6.10 Default values for some global parameters 50
6.11 Source files definition 51
6.11.1 sources_root method 51
6.11.2 c_source(s), cpp_source(s) methods 53
6.11.3 mswin_rc_file method 55
6.12 Additional object files linking 55
6.13 Additional libraries linking 56
6.13.1 Selecting static or dynamic libraries explicitly 56
6.13.2 Enumeration of search paths for the libraries 58
6.14 Optimization mode selection 59
6.15 Functions for setting local, global and upspread parameters 59
6.15.1 include_path, global_include_path 60
6.15.2 define, global_define 60
6.15.3 compiler_option, global_compiler_option 60
6.15.4 c_compiler_option, global_c_compiler_option 60
6.15.5 cpp_compiler_option, global_cpp_compiler_option 60
6.15.6 linker_option, global_linker_option 60
6.15.7 librarian_option, global_librarian_option 60
6.15.8 Resource compiler on Microsoft Windows platform 60
6.16 Manifest support for Visual C++ 8/9/10 61
6.16.1 Installation of the rules of work with the manifests 61

6.16.2 The name of the manifest file for generation of the manifest by a linker 62
6.16.3 Manifest Tool 63
6.17 Mac OS bundle support 64
6.18 Custom tool names 65

7 Additional features **67**
7.1 mxx-show-cmd argument 67
7.2 mxx-keep-tmps argument 67
7.3 mxx-show-tmps argument 68
7.4 mxx-dry-run argument 68
7.5 Exceptions 69
7.6 Adding make-rules into a project 69

8 Unit-testing support **72**
8.1 Unit-testing with binary applications 72
8.1.1 Definition of unit-test application 72
8.1.2 The basic idea 72
8.1.3 Unit-test target class 72
8.1.4 Example 73
8.2 Running unit-tests using comparison of text files 74
8.2.1 MxxRu::TextfileUnittestTarget class 75
8.2.2 Example 75
8.2.3 Features 76

9 Qt4 generator **77**
9.1 Introduction 77
9.2 Qt4 generator usage 78
9.2.1 Adding definitions required to a project file 78
9.2.2 Adding definitions in presence of pkg-config 78
9.2.3 Creation of Qt4 generator 79
9.2.4 Header files definition to generate signal/slot implementation sources 80
9.2.5 Source files definition to generate signal/slot implementation sources 80
9.2.6 UI files definition 81
9.2.7 Resource files definition 81
9.2.8 TS files definition 82
9.2.9 Output folder for moc tool 83
9.2.10 Output folder for uic tool 83
9.2.11 Output folder for rcc tool 83
9.2.12 Extension for source files generated 84
9.2.13 Extension for header files generated 84
9.2.14 Extension for moc files generated 84
9.2.15 The file name of moc tool 84
9.2.16 The file name of uic tool 84
9.2.17 The file name of rcc tool 85
9.2.18 The file name of lupdate tool and custom lupdate options 85

10 Project file stub generators **86**
10.1 Introduction . . . 86
10.2 The mxxrugen generator . . . 87
10.3 Standard templates for C/C++ projects . . . 88
10.3.1 cpp-composite, cpp-dll, cpp-exe, cpp-lib, cpp-lib-collection . . . 88
10.3.2 cpp-build-root . . . 90
10.4 Standard templates for unit-tests . . . 90
10.4.1 bin-unittest . . . 90
10.5 The usage of the stub generator from text editors . . . 91
10.5.1 VIM . . . 91
10.6 The creation of custom templates . . . 92

A Qt3 generator **93**
A.1 Introduction . . . 93
A.2 Qt generator usage . . . 94
A.2.1 Adding definitions required to a project file . . . 94
A.2.2 Creation of Qt generator . . . 94
A.2.3 Header files definition to generate signal/slot implementation sources 94
A.2.4 Source files definition to generate signal/slot implementation sources 95
A.2.5 UI files definition . . . 95
A.2.6 Output folder for moc tool . . . 96
A.2.7 Output folder for uic tool . . . 96
A.2.8 Extension for source files generated . . . 97
A.2.9 Extension for header files generated . . . 97
A.2.10 Extension for moc files generated . . . 97
A.2.11 The file name of moc tool . . . 97
A.2.12 The file name of uic tool . . . 97

Chapter 1

Introduction

1.1 About Mxx_ru

Mxx_ru (Make++ on Ruby[1]) is a cross-platform build tool. It's intended for C/C++ languages in general, but it's easy to extend for handling other languages. In contrast to well-known make utility, which provides explicit make rules with targets, dependencies and build commands, Mxx_ru have a template-based approach. Project file in Mxx_ru is a small Ruby program using already defined class (template) from Mxx_ru. The only thing developer is have to worry about is to create an object of required class and execute some of it's methods.

1.2 Features

Mxx_ru unifies the build process for applications across different compilers and platforms. It allows to work with different programming languages, and functionality of Mxx_ru may vary to reflect each language specifics.

Here is the features provided for C/C++ languages:

- Using one project file for different compilers and platforms;
- Automatic dependency tracking using internal source code analyzer;
- Ready-to-use templates for applications, dynamic and static libraries;
- Ready-to-use template for composite projects (when more then one project are combined together);
- Resource file compilation on Microsoft Windows platform;
- Visual C++ 8.0 manifest support on Microsoft Windows platform;
- Support for "Source code generators" and some of ready-to-use generators, such as Qt moc.

[1] `http://www.ruby-lang.org`

1.3 A bit of history

Mxx_ru is an evolution of Make++ tool, originally developed by Yauheni Akhotnikau[2]. Make++ was based on wmake from Watcom C++ initially. Then it was completely rewritten on C++. The main task for Make++ now was a creation of makefile for concrete compilers based on simple text project description. This solution was found to be hard to adapt to new tools and platforms, so new version of Make++ was required. Next iteration of Make++ was make tool and self-script language interpreter simultaneously. But this tool also had a problems with new tools, especially when source code generation was required. As a result, new, fourth version of Make++ was created. It was self-script interpreter at most, and make module functionality was accessible as API functions.

Fourth version of Make++ was widely used by Intervale company. But it's also reached a limit of opportunities in course of time. In particular, transitions to new platforms and compilers become more and more difficultly, because of limited algorithmic features of Make++ script language.

The basic idea of the fifth version remains the same: Project file — is a small program on interpretable script language. Make module functionality is accessible through API functions and classes. The only difference was in language chosen - it was Ruby.

1.4 Thanks

The author expresses gratitude to the colleagues from the Intervale company[3] for their patience shown during experiments with Mxx_ru. And as to a management of the company for support during development of Mxx_ru.

Separate thank to Mikhail Lyossin for his heavy and difficult work on translation of the documentation and comments in Mxx_ru to the English language and that due to his enthusiasm, Mxx_ru has received the second breath and opportunity of the further development.

[2] `http://eao197.narod.ru`
[3] `http://www.intervale.ru`

Chapter 2

Installation

2.1 Prerequisites

Ruby 1.9.1 is required for using Mxx_ru since v.1.5.0 (Ruby 1.8.* is required for previous versions of Mxx_ru). You may download Ruby installation from `http://www.ruby-lang.org`. One-Click Ruby Installer for Microsoft Windows is available on `http://rubyinstaller.org/`.

2.2 Installing Mxx_ru

There is a few ways to install Mxx_ru:

- Install Mxx_ru as RubyGem:

  ```
  gem install Mxx_ru
  ```

  Or to upgrade currently installed Mxx_ru RubyGem:

  ```
  gem update Mxx_ru
  ```

  This is a preferred way.

- Check out Mxx_ru from Subversion-repository:

  ```
  svn co http://svn.code.sf.net/p/mxxru/code/<PATH>
  ```

  Where 'PATH' is a path to concrete Mxx_ru version in repository. For example, to get stable version 1.6.4:

  ```
  svn co http://svn.code.sf.net/p/mxxru/code/tags/1.6.4
  ```

  or to get development-version 1.6

  ```
  svn co http://svn.code.sf.net/p/mxxru/code/1.6
  ```

Working copy of Mxx_ru can be placed into any folder. It's typically named `c:/ruby/lib/ruby/site-ruby` on Microsoft Windows platform, or `/usr/local/lib/ruby/site-ruby` on Unix. In such case Ruby can automatically find Mxx_ru because these paths used by Ruby as default paths for searching Ruby libraries.

Also it's possible to check out Mxx_ru into any folder you prefer and define the RUBYLIB environment variable. This variable should be the directory where a Mxx_ru archive was checked out. For example, on Microsoft Windows platform it may be:

```
set RUBYLIB=d:/my/mxx_ru
```

Or, on Unix platform:

```
export RUBYLIB=~/my/mxx_ru
```

Now, you have to setup Mxx_ru for languages and tools you planning to use.

Note. Under some Ruby installation (where RubyGems was not installed as part of Ruby by default) may be required to specify Ruby interpreter to use 'rubygems' with any Ruby script:

```
export RUBYOPT=rubygems
```

2.2.1 Setting up Mxx_ru for C/C++ projects

`MXX_RU_CPP_TOOLSET` environment variable should be defined to use Mxx_ru with a C/C++ projects. The syntax used to define this environment variable is as follows:

```
MXX_RU_CPP_TOOLSET="<file> [tag=value [tag=value [...]]"
```

where `<file>` – is the name of .rb-file from Mxx_ru responsible for C/C++-toolset object creation. For example:

bcc_win32_5 Borland C++ 5.* compiler on Microsoft Windows platform;

clang_linux Clang compiler on Linux platform;

vc7 Visual C++ 7.* compiler on Microsoft Windows platform;

vc8 Visual C++ 8.* compiler on Microsoft Windows platform;

vc9 Visual C++ 9.* compiler on Microsoft Windows platform;

vc10 Visual C++ 10.* compiler on Microsoft Windows platform;

vc11 Visual C++ 11.* (Visual Studio 2012) compiler on Microsoft Windows platform;

vc12 Visual C++ 12.* (Visual Studio 2013) compiler on Microsoft Windows platform;

vc14 Visual C++ 14.* (Visual Studio 2015) compiler on Microsoft Windows platform;

gcc_darwin GNU C++ compiler on Mac OS X platform;

gcc_linux GNU C++ compiler on Unix platform (including FreeBSD and Linux);

gcc_sparc_solaris GNU C++ compiler on SPARC Solaris platform;

gcc_cygwin GNU C++ compiler from Cygwin package on Microsoft Windows platform.

icc_win Intel C++ Compiler on Microsoft Windows platform;

c89_nsk c89 compiler on HP NonStop platform in Open System Services environment.

The pairs tag and value would be defined as a tags for selected C/C++ toolset. Examples:

```
export MXX_RU_CPP_TOOLSET="gcc_linux unix=linux arch=x86"
```

```
set MXX_RU_CPP_TOOLSET=vc7
```

```
set MXX_RU_CPP_TOOLSET=bcc_win32_5
```

Chapter 3

Versions Mxx_ru

The following chapter describes all changes in Mxx_ru during version changes. It's intended to give compact information of changes in Mxx_ru to advanced users, to avoid them search for that changes in the whole text. This chapter may be skipped by beginners.

3.1 Version 1.6

Mxx_ru has been moved to Ruby 2.0 and 2.1.

Support for Visual C++ 11.0, 12.0 and 14.0 has been added.

Support for Clang C++ compiler on Linux and FreeBSD platforms has been added.

Methods `force_cpp03`, `force_cpp11` **and** `force_cpp14` **have been added to** `MxxRu::Cpp::Toolset` **(8.1.4 on page 74).**

The `MxxRu::NegativeBinaryUnittestTarget` **target type has been added as addition for** `MxxRu::BinaryUnittestTarget`**.**

3.2 Version 1.5

Mxx_ru has been moved to Ruby 1.9.1. Older versions of Ruby are not supported in versions 1.5.*.

Support for Visual C++ 10.0 has been added.

Initial support for Intel C++ on Windows has been added. Tested on Intel Parallel Studio 2011.

3.3 Version 1.4

Support for Mac OS X has been added.

Support for Qt4 has been added. For more details see 9 on page 77.

Support for Visual C++ 9.0 has been added.

A new target type for C/C++ projects has been added: MxxRu::Cpp::LibCollection. For more details see 6.1.3 on page 39.

A new approach of checking command line arguments has been released. Now, if a project file is started with argument `--help` then a list of options, which are available for this project's type, is shown. Command line arguments are checked now and unknown arguments lead to the error.

Custom names of compiler, linker, librarian and so on can be specified now. It is sometimes necessary to start some special version of compiler instead of the standard one. For example, *gcc336* must be used for legacy code when the standard compiler is GCC version 4.*. Or *remote_cl* is used for launching *cl* on a remote host. In such situations it is possible to specify names of tools in `MXX_RU_CPP_TOOLSET` environment variable:

```
export MXX_RU_CPP_TOOLSET="gcc_linux compiler_name=gcc336"
```

For more details see 6.18 on page 65.

An ability of project file stub generation has been added. A project file stub generator has been added into Mxx_ru to avoid manual creation of most widespread types of project files. For example, if the following command had been run:

```
mxxrugen cpp-lib -t my_lib -o my/lib/prj.rb
```

then file `my/lib/prj.rb` would be created with the content similar to:

```
1  require 'rubygems'
2
3  gem 'Mxx_ru', '>= 1.3.0'
4
5  require 'mxx_ru/cpp'
6
7  MxxRu::Cpp::lib_target {
8
9    # Define your target name here.
10   target 'my_lib'
11
12   # Enumerate one or more required projects.
13   #required_prj 'some project'
14
15   # Enumerate your C/C++ files here.
16   #c_source 'C source file'
17   #cpp_source 'C++ source file'
18
19 }
```

For more details see 10 on page 86.

3.4 Version 1.3

Libraries list creation method has been changed for GCC toolset. The names of all libraries are now inside `-Wl,--start-group` and `-Wl,--end-group` option pair. This allows to set library names in random order in project file (It was required to set them keeping dependencies resolution order in mind in earlier versions).

The names of modules and classes of Mxx_ru are now brought to conformity with standard conventions of Ruby language. Now, `MxxRu` is used instead of `Mxx_ru`, `AbstractTarget` instead of `Abstract_target`, `BinaryUnittestTarget` instead of `Binary_unittest_target` and so on. All old names are preserved as a synonims of new

names, therefore backwards compatibility is preserved. Even in the given manual some examples using old names may be found. Compatibility with old naming system will be preserved in the future, names were changed just to avoid problems induced by non-standard names in future evolution of Mxx_ru.

Added `--mxx-rebuild` mode (4.6 on page 27). Now this command:

```
ruby prj.rb --mxx-rebuild
```

is equivalent to this command sequence:

```
ruby prj.rb --mxx-clean
ruby prj.rb
```

Added `BinaryTarget#lib_static`, `BinaryTarget#lib_shared` methods. They allow to force Unix linker to use concrete type of given library. Thus, if `lib_static` method is used to define a library, linker would use only static version of the library, ignoring shared library with the same name. For more details, see 6.13.1 on page 56.

Added `QtGen#uic_result_subdir` method. It defines which subfolder would be used to store results of uic utility from Trolltech Qt. In earlier versions result files were stored in the same folder where source .ui file was. For more details, see A.2.7 on page 96.

For uic utility, when it ran for .cpp file generation, short name of .hpp file is defined in '-i' argumentuntethered. I.e. instead of 'src/module/some_header.hpp', 'some_header.hpp' is sent. Accordingly, into generated .cpp file the following directive would be written:

```
1 #include "some_header.hpp"
```

This becomes important in some cases of project file structure organization.

New task 'test' is added into Rakefile to automate run of Mxx_ru unit tests. Some tests were converten into unit-tests. Special mix-in TestWithCompilation was implemented (in `tests/test_with_compilation.rb`), which makes easier creation of unit tests, where compilation of test projects is needed.

Were added `BinaryTarget#lib_path`, `BinaryTarget#lib_paths` methods. With their help linker knows where to find libraries (in case when it isn't convenient or even impossible to do with `lib`, `lib_static` or `lib_shared` methods). For more details, see 6.13.2 on page 58.

On Windows platform now definition of '.lib' file extension for libraries is optional. In previous versions of Mxx_ru it was actual for Visual C++ toolset. Since version 1.3 this toolset checks availability of extension in the name of the library and adds it if needed.

Was added `Cpp::Target#target_ext` method. It allows to set any extension for build result. For example, for DLL it may be '.mll' if we compiling Maya plugin for windows:

```
1 MxxRu::Cpp::dll_target {
2   target 'my_plugin'
3   target_ext '.mll'
4   ...
5 }
```

For more details, see 6.8.4 on page 49.

Added `--mxx-cpp-extract-options` mode, which forces Mxx_ru to show C/C++ compiler and linker options. Given mode allows to get options, which would be used during project build on Mxx_ru to help make integration with other build system easier. For more details, see 6.1.9 on page 44.

Was added `MxxRu::Cpp::CustomSubdirObjPlacement` class, which gives one more method to place build results of C/C++ projects. It stores all exe/dll(so)/lib(a) in one folder, and all intermediate compilation results, such as obj(o)/res e.t.c. in another, duplicating file structure of source files in the intermediate folder. For more details, see 6.1.2 on page 36.

3.5 Version 1.2

Added command-line arguments –mxx-brief-show, –mxx-brief-hide and method `MxxRu::enable_show_brief` for controlling brief description of executed commands.

Argument prj_alias for `composite_target`, `exe_target`, `lib_target` and `dll_target` now optional. If it omitted than Mxx_ru try detect prj_alias form `Kernel#caller` result (assume what call to *_target method was made directly in project file). This allow to write:

```
1 MxxRu::Cpp::exe_target {
2   ...
3 }
```

instead of:

```
1 MxxRu::Cpp::exe_target( 'some/module/path/prj.rb' ) {
2   ...
3 }
```

See 5.1.3 on page 31 for more details.

Added methods `Abstract_target#required_prjs`, `Cpp::Target#c_sources`, `Cpp::Target#cpp_sources` those accept Enumerable (plural form for corresponding singular analogs). New methods can accept result of Dir.glob(). For example:

```
1 MxxRu::Cpp::exe_target {
2   target 'some_target'
3   cpp_sources Dir.glob( 'src/**/*.cpp' )
4 }
```

See 5.1.2 on page 30 and 6.11.2 on page 53 for more details.

Chapter 4

Examples

4.1 Simple application

Imagine we need to compile one C++ file and make an application:

```
#include <iostream>
#include <string>

void
say_hello( std::ostream & to )
  {
    to << "Hello!" << std::endl;
  }

void
say_bye( std::ostream & to )
  {
    to << "Bye!" << std::endl;
  }

int
main()
  {
    say_hello( std::cout );

    std::cout << "Simple exe main..." << std::endl;

    say_bye( std::cout );

    return 0;
  }
```

Following project file would be required for that (named prj.rb):

```
require 'mxx_ru/cpp'

MxxRu::Cpp::exe_target {
  target( "simple_exe" )
```

```
5
6     cpp_source( "main.cpp" )
7   }
```

You have to run Ruby interpreter with project file to start build process:

```
ruby prj.rb
```

As a result, on Microsoft Windows platform, simple_exe.exe file would be built. Or simple_exe executable on unix platform.

4.2 Application with static library

Now, let's try to break the example resulted above into static library, exporting functions say_hello(), say_bye(), and basic module, using that exported functions.

We will need those files for the static library:

- say.hpp header file

```
1  #if !defined( _SAY_HPP_ )
2  #define _SAY_HPP_
3
4  #include <iostream>
5
6  void
7  say_hello( std::ostream & to );
8  void
9  say_bye( std::ostream & to );
10
11 #endif
```

- say.cpp implementation file

```
1  #include <iostream>
2
3  void
4  say_hello( std::ostream & to )
5    {
6      to << "Hello!" << std::endl;
7    }
8
9  void
10 say_bye( std::ostream & to )
11   {
12     to << "Bye!" << std::endl;
13   }
```

- say.rb project file

```
require 'mxx_ru/cpp'

MxxRu::Cpp::lib_target {
  target( "say" )

  cpp_source( "say.cpp" )
}
```

Also we will need those files for the basic module:

- main.cpp implementation file

```
#include <iostream>

#include <say.hpp>

int
main()
  {
    say_hello( std::cout );

    std::cout << "Exe and lib main..." << std::endl;

    say_bye( std::cout );

    return 0;
  }
```

- prj.rb project file

```
require 'mxx_ru/cpp'

MxxRu::Cpp::exe_target {
  target( "exe_and_lib" )

  required_prj( "say.rb" )

  include_path( "." )

  cpp_source( "main.cpp" )
}
```

You have to run ruby interpreter with a prj.rb file as a parameter to build the project, including static library and basic module. Mxx_ru would build say library and link it to the basic module automatically.

Notice that you have no need to specify the library name in prj.rb project file. Moreover, the basic module doesn't know at all, whether the library static or dynamic. It just points to it's requirements — it needs say.rb project file to be built. All other actions are handled by Mxx_ru — subproject compilation, library name detection, linking library to basic module.

4.3 Application with shared and static libraries

Let's make things more complex — now we need to create a shared (dynamically linked) library, exporting inout_t class. This class is intended for printing "Hello!" in it's constructor and "Bye!!" in the destructor. We can define objects of inout_t class in a functions to print messages to standard output about entering/leaving a function. Inout library will use modified version of say library — there is one more argument in say_hello, say_bye functions. Here is what we got for :

- say.hpp header file

```
#if !defined( _SAY_HPP_ )
#define _SAY_HPP_

#include <iostream>
#include <string>

void
say_hello( std::ostream & to, const std::string & where );
void
say_bye( std::ostream & to, const std::string & where );

#endif
```

- say.cpp implementation file

```
#include <say.hpp>

void
say_hello( std::ostream & to, const std::string & where )
  {
    to << where << ": Hello!" << std::endl;
  }

void
say_bye( std::ostream & to, const std::string & where )
  {
    to << where << ": Bye!" << std::endl;
  }
```

- say.rb project file

```
require 'mxx_ru/cpp'

MxxRu::Cpp::lib_target {
  target( "say" )

  include_path( "." )

  cpp_source( "say.cpp" )
}
```

Here is inout library files:

- inout.hpp header file

```
#if !defined( _INOUT_HPP_ )
#define _INOUT_HPP_

#include <string>

#if defined( INOUT_MSWIN )
  #if defined( INOUT_PRJ )
    #define INOUT_TYPE __declspec(dllexport)
  #else
    #define INOUT_TYPE __declspec(dllimport)
  #endif
#else
  #define INOUT_TYPE
#endif

class   INOUT_TYPE  inout_t
  {
  public :
    inout_t( const std::string & method );
    ~inout_t();

  private :
    std::string m_method;
  };

#endif
```

- inout.cpp implementation file

```
#include <say.hpp>

#include <inout.hpp>

inout_t::inout_t(
  const std::string & method )
  : m_method( method )
  {
    say_hello( std::cout, method );
  }

inout_t::~inout_t()
  {
    say_bye( std::cout, m_method );
  }
```

- inout.rb project file

```
require 'mxx_ru/cpp'

MxxRu::Cpp::dll_target {
  target( "inout" )
  implib_path( "." )

  required_prj( "say.rb" )

  include_path( "." )

  define( "INOUT_PRJ" )

  if "mswin" == toolset.tag( "target_os" )
    define( "INOUT_MSWIN" )
  end

  cpp_source( "inout.cpp" )
}
```

Basic module contains following files:

- main.cpp implementation file

```
#include <iostream>

#include <inout.hpp>

void
some_func()
  {
    inout_t braces( "some_func" );

    std::cout << "Some functionality..." << std::endl;
  }

int
main()
  {
    inout_t braces( "main" );

    std::cout << "Exe, dll, lib main..." << std::endl;

    some_func();

    return 0;
  }
```

- prj.rb project file

```
require 'mxx_ru/cpp'

```

```
3  MxxRu::Cpp::exe_target {
4    target( "exe_dll_lib" )
5
6    required_prj( "inout.rb" )
7
8    include_path( "." )
9
10   cpp_source( "main.cpp" )
11 }
```

Notice that basic module doesn't know anything about it's dependency from say library. That dependency is tracked automatically by Mxx_ru. This dependency is very important for Unix platforms, unlike such platforms as Microsoft Windows or OS/2. In particular, during linking of basic module on unix platform, both libraries should be defined in linker parameters. In case of Mxx_ru there is no need to define that dependency explicitly — Mxx_ru tracks that dependencies and uses them on each platform in a preferred way.

4.4 Application with shared and static libraries in different folders

There is two serious lacks in the example above:

1. All files, including headers, sources, project, results of compilation and linking are inside one folder. For example, these files are located in that folder after building g++ project on Cygwin platform:

```
exe_dll_lib.exe*
inout.cpp
inout.hpp
inout.rb
libinout.a
libinout.so*
libsay.a
main.cpp
o/
prj.rb
say.cpp
say.hpp
say.rb
```

   It's obvious there would be hard to work in case of growing amount of files in a project.

2. Each project file contains `include_path(".")` instruction. It would be more convenient to have one place to set common parameters for a group of subprojects.

It is necessary to place each subproject in individual directory under one global folder of project in order to overcome the first lack. Assume the following structure of folders and files in our example:

```
/
|--inout/
|  |--inout.cpp
|  |--inout.hpp
|  `--prj.rb
|--lib/
|--main/
|  |--main.cpp
|  `--prj.rb
|--say/
|  |--prj.rb
|  |--say.cpp
|  `--say.hpp
`--build.rb
```

"lib" folder is intended to store static and shared libraries.

4.4.1 "say" Subproject

Here is files say subproject contains:

- say.hpp header file

```
1  #if !defined( _SAY_HPP_ )
2  #define _SAY_HPP_
3
4  #include <iostream>
5  #include <string>
6
7  void
8  say_hello( std::ostream & to, const std::string & where );
9  void
10 say_bye( std::ostream & to, const std::string & where );
11
12 #endif
```

- say.cpp implementation file

```
1  #include <say/say.hpp>
2
3  void
4  say_hello( std::ostream & to, const std::string & where )
5    {
6      to << where << ": Hello!" << std::endl;
7    }
8
9  void
10 say_bye( std::ostream & to, const std::string & where )
11   {
```

```
12      to << where << ": Bye!" << std::endl;
13    }
```

Note: say.hpp file is loaded as `<say/say.hpp>`;

- prj.rb project file

```
1  require 'mxx_ru/cpp'
2
3  MxxRu::Cpp::lib_target {
4    target_root( "lib" )
5    target( "say" )
6
7    cpp_source( "say.cpp" )
8  }
```

4.4.2 "inout" Subproject

Here is files inout subproject contains:

- inout.hpp header file

```
1  #if !defined( _INOUT_HPP_ )
2  #define _INOUT_HPP_
3
4  #include <string>
5
6  #if defined( INOUT_MSWIN )
7    #if defined( INOUT_PRJ )
8      #define INOUT_TYPE __declspec(dllexport)
9    #else
10     #define INOUT_TYPE __declspec(dllimport)
11   #endif
12 #else
13   #define INOUT_TYPE
14 #endif
15
16 class   INOUT_TYPE  inout_t
17   {
18   public :
19     inout_t( const std::string & method );
20     ~inout_t();
21
22   private :
23     std::string m_method;
24   };
25
26 #endif
```

- inout.cpp implementation

```
#include <say/say.hpp>

#include <inout/inout.hpp>

inout_t::inout_t(
  const std::string & method )
  : m_method( method )
  {
    say_hello( std::cout, method );
  }

inout_t::~inout_t()
  {
    say_bye( std::cout, m_method );
  }
```

Also notice that you have to point subproject name in `#include` directive to load header files from that subproject: `<say/say.hpp>`, `<inout/inout.hpp>`.

- prj.rb project file

```
require 'mxx_ru/cpp'

MxxRu::Cpp::dll_target {
  target( "inout" )
  implib_path( "lib" )

  required_prj( "say/prj.rb" )

  define( "INOUT_PRJ" )

  if "mswin" == toolset.tag( "target_os" )
    define( "INOUT_MSWIN" )
  end

  cpp_source( "inout.cpp" )
}
```

4.4.3 "main" subproject

Here is the files Main subproject contains:

- main.cpp implementation file

```
#include <iostream>

#include <inout/inout.hpp>

void
some_func()
```

```
7   {
8     inout_t braces( "some_func" );
9
10    std::cout << "Some functionality..." << std::endl;
11  }
12
13 int
14 main()
15  {
16    inout_t braces( "main" );
17
18    std::cout << "Exe, dll, lib main..." << std::endl;
19
20    some_func();
21
22    return 0;
23  }
```

- prj.rb project file

```
1 require 'mxx_ru/cpp'
2
3 MxxRu::Cpp::exe_target {
4   target( "exe_dll_lib" )
5
6   required_prj( "inout/prj.rb" )
7
8   cpp_source( "main.cpp" )
9 }
```

4.4.4 build.rb file

build.rb — is a special project file. It should be placed in project root folder and ruby interpreter should be run from that folder. Here all global project parameters should be defined.

If project contains subprojects, just like in our case, then build.rb usually represented as a composite project.

Here is build.rb for our example:

```
1 require 'mxx_ru/cpp'
2
3 MxxRu::Cpp::composite_target( MxxRu::BUILD_ROOT ) {
4   required_prj( "main/prj.rb" )
5
6   global_include_path( "." )
7 }
```

4.4.5 Building a project

To start build process, you should enter project root folder (the build.rb file should be there) and run Ruby interpreter with build.rb file as a parameter. In our example, after building g++ project on Cygwin platform, these files should appear in project root folder:

```
build.rb
exe_dll_lib.exe*
inout/
lib/
libinout.so*
main/
say/
```

And these files should appear in lib folder:

```
libinout.a
libsay.a
```

4.4.6 Building a single subproject only

To build a single subproject, you should enter project root folder (where build.rb file is) and run Ruby interpreter with a subproject file name as a first parameter, and `--mxx-cpp-1` argument as a second.

For example, to build the inout subproject, you should use this command:

```
ruby inout/prj.rb --mxx-cpp-1
```

In that case only "inout" subproject would be built, without any other subprojects. In other words, build will fail if "say" subproject wasn't built before because "inout" project depends from it.

4.5 Cleaning

Mxx_ru builds a project by default (compilation and linking in case of C/C++ projects). Mxx_ru also supports reverse operation – *clean*, i.e. removing all files were built. In case of C/C++ project clean operation removes all object files, libraries, executables and other results of compilation and linking.

You need to run Ruby interpreter with project file name and `--mxx-clean` argument to perform clean operation:

```
ruby build.rb --mxx-clean
```

Actions performed by clean operation are extended (by default) to all subprojects the main project depends from. If you want to clean a single C/C++ project only, you have to pass `--mxx-cpp-1` argument also:

```
ruby some/subproject/prj.rb --mxx-clean --mxx-cpp-1
```

4.6 Rebuilding

Sometimes it's required to perform full cleanup of a project and then compile it again. Instead of running Ruby twice:

```
ruby build.rb --mxx-clean
ruby build.rb
```

in version 1.3 to Mxx_ru was added `--mxx-rebuid` argument, which does that at once:

```
ruby build.rb --mxx-rebuild
```

As well as in case of `--mxx-clean`, adding `--mxx-cpp-1` argument would force `--mxx-rebuild` mode to only one C/C++ project:

```
ruby some/subproject/prj.rb --mxx-rebuild --mxx-cpp-1
```

Chapter 5

The basic idea

The basic idea of Mxx_ru consist of each project should define *target* to build. The target is defined by an object of class, inherited from `MxxRu::AbstractTarget`. Only one target per project file is allowed. Each target must have an unique alias (*prj_alias*). Alias for target is the name of project file it's defined in.

The basic task for project file is to create a target object and to pass it to Mxx_ru using `MxxRu::setup_target` function. As a result, Mxx_ru project files are defined by that template in common:

```
require 'mxx_ru/<something>'

MxxRu::setup_target(
  <some class>.new( <the name of project file> ) {
    <definition of parameters in the project>
  }
)
```

In the most widespread cases for simplification of target installation the auxiliary functions are used, which hide constructor's calls of the appropriate classes and `MxxRu::setup_target` call. For example, used in the previous chapter `MxxRu::Cpp::exe_target`, `MxxRu::Cpp::lib_target`, `MxxRu::Cpp::dll_target` and `MxxRu::Cpp::composite_target` just are such auxiliary functions. Usually they have trivial implementation of a kind:

```
def exe_target( prj_alias, &block )
  MxxRu::setup_target( MxxRu::Cpp::ExeTarget.new( prj_alias, &block ) )
end
```

5.1 More details about MxxRu::AbstractTarget

Class `MxxRu::AbstractTarget` is a base class for all types of targets. This class defines two important methods: `build` for building a target and `clean` for cleaning build results. These methods are defined as abstract inside of `MxxRu::AbstractTarget`. In other words, it's required to define them in classes inherited from `MxxRu::AbstractTarget`. But, because there is no definition of "abstract method" in Ruby, a try to call any of them from `MxxRu::AbstractTarget` class will throw an exception `MxxRu::AbstractMethodEx`.

To support `--mxx-rebuild` mode in `MxxRu::AbstractTarget` class also `reset` method was added. It's main task — to clear status of a target after `build` and `clean` methods call. After call of `reset` method target will be in it's initial state (as if there was no previous calls to `build` or `clean`). Method `reset` in `MxxRu::AbstractTarget` class is implemented as empty method, so inherited classes that doesn't have it's internal state, there is no need to redefine it.

There is one more method defined — you can get an alias of a target by calling a `prj_alias` method.

5.1.1 A names for files created by target

As usual, target is defined to produce one file as a result. Executable file for C++ project, DVI-file for LaTeXproject or jar file for Java project are examples of that. But target may produce more then one file in general — it's dependent from project type only.

No matter how much files are produced by a target, all their names should be passed to the base class `MxxRu::AbstractTarget` using a `mxx_add_full_target_name` method. You may get all the names of files produced using `mxx_full_target_names` method later.

It's important to understand that names of intermediate files are not included in that list. For example, for C/C++ projects object files are not accessible through `mxx_full_target_names` method.

5.1.2 Subtargets

Sometimes target requires files produced by other target. Library files, for example, are required by an application target. That required targets are called subtargets and their projects are called subprojects.

Subproject is a usual project, created using usual rules of Mxx_ru project files. Subprojects are defined with a call of a method `required_prj`:

```
1  require 'mxx_ru/cpp'
2
3  MxxRu::Cpp::composite_target( MxxRu::BUILD_ROOT ) {
4
5      global_include_path( '.' )
6
7      required_prj( 'oess_1/stdsn/prj.rb' )
8      required_prj( 'oess_1/scheme/prj.rb' )
9      required_prj( 'oess_1/util_cpp_serializer/prj.rb' )
10     required_prj( 'oess_1/file/prj.rb' )
11     required_prj( 'oess_1/db/prj.rb' )
12     required_prj( 'oess_1/util_ent_enum/prj.rb' )
13     required_prj( 'oess_1/util_slice_create/prj.rb' )
14     required_prj( 'oess_1/tlv/prj.rb' )
15 }
```

`required_prj` method does more then a storage of subproject name — it loads and runs that subproject too.

`mxx_required_prjs` method allows to get all subproject names.

Subproject handling depends from project type, but if we proceed from common sense, we may assume all subrojects `build` methods are called from `build` method of basic project. At least this is true for C++ projects.

required_prjs method

Starting from version 1.2 in Mxx_ru `required_prjs` method exists, which gets an object of `Enumerable` type as an argument. This allows to rewrite an example above with the following:

```
require 'mxx_ru/cpp'

MxxRu::Cpp::composite_target( MxxRu::BUILD_ROOT ) {

    global_include_path( '.' )

    required_prjs [ 'oess_1/stdsn/prj.rb',
                    'oess_1/scheme/prj.rb',
                    'oess_1/util_cpp_serializer/prj.rb',
                    'oess_1/file/prj.rb',
                    'oess_1/db/prj.rb',
                    'oess_1/util_ent_enum/prj.rb',
                    'oess_1/util_slice_create/prj.rb',
                    'oess_1/tlv/prj.rb' ]
}
```

Also class `Dir` from standard Ruby library can be used:

```
require 'mxx_ru/cpp'

MxxRu::Cpp::composite_target( MxxRu::BUILD_ROOT ) {

    global_include_path( '.' )

    required_prjs Dir[ 'oess_1/**/prj.rb' ]
}
```

here `Dir::[]` is used to recursively search all `prj.rb` files in subfolders of `oess_1` folder.

A combination of classes from standard Ruby library and abilities of Ruby language itself allows to use `required_prjs` with much more complicated constructions. For example, the project file written below is intended for compilation of all tests, placed in subfolders of `test` folder. But, some of them are defined as unit-tests (see 8 on page 72), and for each of them `prj.rb` files exists for build and `prj.ut.rb` for build and running the unit test. For such projects only `prj.ut.rb` file should be defined. All other projects aren't unit tests and contains only `prj.rb` files. Exactly them should be included into `required_prjs` call. A solution is to find all `prj.ut.rb` files at first, and then to add all `prj.rb` file names from folders where are no `prj.ut.rb`:

```
MxxRu::Cpp::composite_target {
  required_prjs Dir[ 'test/**/prj.ut.rb' ]
  required_prjs Dir[ 'test/**/prj.rb' ].delete_if { |f|
```

```
4       # Deleting the name of found project file if prj.ut.rb is found in the same folder.
5       File.exists?( File.join( File.dirname( f ), 'prj.ut.rb' ) )
6     }
7   }
```

5.1.3 Project aliases

Mxx_ru requires for each project to have unique alias — the name of project file, where project is defined. This alias is used by Mxx_ru to ensure that each project is handled only once, but not each time it's found in `required_prj`. Before Mxx_ru version 1.2.0 in project files it was required to repeat each file name as an alias of the project:

```
1 Mxx_ru::setup_target(
2   Mxx_ru::Cpp::Exe_target.new( 'some/module/prj.rb' ) {
3     ...
4   }
5 )
```

that gave rise to unfavourable criticism of users. So, in versions 1.2.0-1.4.10 auxiliary methods `exe_target`, `dll_target`, `lib_target`, `composite_target` and `macos_bundle_target` are learned to automatically detect an alias of a project, if it wasn't given explicitly. In that cases ruby project file name is taken, where one of auxiliary methods takes place. For example, given example above can be rewritten to the following:

```
1 MxxRu::Cpp::exe_target {
2   ...
3 }
```

and, if file "some/module/prj.rb" is run as:

```
ruby some/module/prj.rb
```

then "some/module/prj.rb" would be used as an alias.

It's important to understand, that Mxx_ru still requires an unique alias for each project. And that classes, inherited from `MxxRu::AbstractTarget` (such as `MxxRu::Cpp::ExeTarget` or `MxxRu::BinaryUnittestTarget`), are required to get an alias explicitly in their constructors. It's just some auxiliary functions are added (and will be added) to wide used target types, which can detect a project alias based on ruby file name.

5.1.4 Source code generators

Sometimes we need to generate source files from something, using special tools for doing that. For example, "yacc" generates source code for syntax analyzers from .y-files.

It's clear that different generators operating different types of files and different parameters for them are required. And they produce different amount of source files. For example, "uic" tool from Trolltech Qt may produce header and implementation files.

Keeping that in mind, Mxx_ru uses a generalized conception of a generator. There is `MxxRu::AbstractGenerator` base class, with two methods defined: `build` and `clean`. All generators should be inherited from that base class.

As long as different generators requires different approaches to work with, Mxx_ru doesn't specifies any other interfaces for generators. There is only one requirement: all generators should be passed to `MxxRu::AbstractTarget` class using `generator` method.

Class, inherited from `MxxRu::AbstractTarget` is responsible to execute a generator. For instance, C++ target classes are executing generators after all subtargets are processed and just before a compilation process.

Since Mxx_ru does not define any interface to pass instructions to a generator, usually usage of a generator looks like that:

```
require 'mxx_ru/<something>'

require '<file describing a generator>'

MxxRu::setup_target(
    MxxRu::<target class>.new( <alias> ) {
        ...
        gen = generator( <generator class>.new( <parameters> ) )
        gen.<some method>( <parameters> )
        ...
    }
)
```

For example:

```
require 'mxx_ru/cpp'

require 'oess_1/version'
require 'oess_1/util_cpp_serializer/gen'

MxxRu::Cpp::dll_target {

  ...
  ddl_cpp_generator = generator(
    Oess_1::Util_cpp_serializer::Gen.new( self ) )

  sources_root( "impl" ) {

    sources_root( "db_struct" ) {
       cpp_source( "unit.cpp" )
       ddl_cpp_generator.ddl_file( "unit.ddl" )

       cpp_source( "slice_unit.cpp" )
       ddl_cpp_generator.ddl_file( "slice_unit.ddl" )

       cpp_source( "slice_stream_item.cpp" )
       ddl_cpp_generator.ddl_file( "slice_stream_item.ddl" )

       cpp_source( "db_slicemap.cpp" )
       cpp_source( "std_db_slicemap.cpp" )

       cpp_source( "db_struct.cpp" )
       cpp_source( "std_db_struct.cpp" )
```

```
29      }
30    }
31    ...
32  }
```

Or, considering `generator` method returns a generator, you may call it's methods without saving a reference to it in a variable:

```
1  require 'mxx_ru/cpp'
2
3  require 'oess_1/util_cpp_serializer/gen'
4
5  MxxRu::Cpp::exe_target {
6
7    target( "test.stdsn.shptr" )
8
9    required_prj( "oess_1/defs/prj.rb" )
10   required_prj( "oess_1/io/prj.rb" )
11
12   generator( Oess_1::Util_cpp_serializer::Gen.new( self ) ).
13     ddl_file( "main.ddl" )
14
15   cpp_source( "main.cpp" )
16 }
```

`mxx_generators` method return a list of all generators installed.
One more way of generators usage is described in 7.6 on page 69

5.2 More details about MxxRu::setup_target

Mxx_ru contains a map describing compliance of targets to aliases (i.e. names of project files). `MxxRu::setup_target` method should be used to add new target to that map.

Moreover, `MxxRu::setup_target` method automatically executes a build or cleaning procedure, if target passed to `MxxRu::setup_target` is a top target (i.e. target is created in a file passed to Ruby interpreter explicitly) [1].

During processing of a top target `MxxRu::setup_target` method checks if `--mxx-clean` argument was passed in. If it was, `clean` method is executed for the target, otherwise `build` method is called. If `--mxx-rebuld` mode is set, then `MxxRu::setup_target` will call `clean`, `reset` and `build` in a sequence.

[1] In fact, top target is a first target that executes a `MxxRu::AbstractTarget` constructor

Chapter 6

Mxx_ru for C/C++ projects

6.1 Introduction

Mxx_ru offers a ready to use tool sets for building C/C++ projects. That tools are described in `mxx_ru/cpp` file. That file should be loaded with `require` directive:

```
1 require 'mxx_ru/cpp'
2 ...
```

To support C/C++ two important conceptions are introduced to Mxx_ru: *toolset* and *obj_placement*.

Mxx_ru introduces ready classes for such target types as executable file, shared library file (also known as dynamically loaded library, including support of import library), library file, composite project.

6.1.1 Toolset conception

Toolset — is a collection of tools, used for building a project. In other words, toolset defines a type, version and other features of concrete compiler on specific platform.

Mxx_ru is already adapted for some compilers. In particular, here is short list of compilers supported:

- Borland C++ (5.* on Windows platform)
- Visual C++ (12.*, 11.*, 10.*, 9.*, 8.*, 7.* and 6.* versions)
- Intel C++ on Windows (Intel Parallel Studio 2011)
- GNU C++ on unix and windows (MinGW and Cygwin) platforms
- c89 for HP NonStop (native version and cross-compiler from eToolkit for Windows)
- Clang C++ on Linux and FreeBSD platforms

Toolset type should be defined during setup process (see 6.2 on page 45 for details), otherwise it wouldn't work properly.

Toolset is an object of a class, inherited from `MxxRu::Cpp::Toolset`. Toolset object is created by Mxx_ru and used for a whole project (project name passed to Ruby interpreter and all it's subprojects). It's impossible to replace toolset object. Access to toolset object is provided through `toolset()` method.

Toolset names

Each toolset have it's name, available through `name` method of a `MxxRu::Cpp::Toolset` class. That name can be used for tuning a project for specific compiler. For example:

```
MxxRu::Cpp::exe_target {
    ...
    if "vc" == toolset.name
        # Visual C++ adaptation
        cpp_source( "mswin/vc/exception_handler.cpp" )
    elif "gcc" == toolset.name
        # GNU C++ adaptation
        cpp_source( "gcc/exception_handler.cpp" )
    end
    ...
}
```

Table 6.1 shows already defined names of compilers for a moment:

bcc	Borland C++ (Windows).
c89_nsk	c89 for HP NonStop (HP NonStop and Windows).
clang	Clang C/C++ (Unix, Mac OS X and Windows).
gcc	GNU C/C++ (Unix, Mac OS X and Windows).
vc	Visual C++ (Windows).
icc_win	Intel C++ (Windows).

Table 6.1: Toolset names defined.

Toolset tags

As long as toolset adapted for a specific compiler on specific platform, toolset may be used in a project for tuning it for specific platform. Tags conception is introduced for that purposes. Tag — is an unique text key with a text value attached. Tags are defined by toolsets themselves, and also they may be defined during Mxx_ru setup process.

Values of tags are available through `tag(a_name, a_default=nil)` method. Value of tag is returned if tag with name defined is exists in a toolset. Otherwise, value of `a_default` argument is returned. If tag wasn't defined and `a_default==nil`, `MxxRu::Cpp::Toolset::UnknownTagEx` exception is thrown.

Some tags are mandatory, so they should be defined for all toolsets and all platforms. (see table 6.2).

host_os	Operating system build process was run on.
target_os	Operating system build process was run for.

Table 6.2: Mandatory tags for all toolsets.

Values for host_os and target_os may vary if cross-compiler is used, in case of c89 for HP NonStop, as an example.

Operating systems are identified by the names, enumerated in the 6.3 table.

mswin	32-bit version of Microsoft Windows.
tandem_oss	OSS subsystem (Open System Services) for HP NonStop.
unix	different ports of Unix.

Table 6.3: The names of operating systems.

For unix platform *unix_port* tag is also mandatory. It defines the port of unix system. It is recommended to use values enumerated in the 6.4 table. [1].

freebsd	FreeBSD.
cygwin	Cygwin on Microsoft Windows.
darwin	Mac OS X.
linux	GNU/Linux.
solaris	Sun Solaris.

Table 6.4: Unix port names.

There is more tags may be defined during a Mxx_ru setup.
Example:

```
MxxRu::Cpp::dll_target {
    # We want to place results to lib folder on Unix.
    if "unix" == toolset.tag( "target_os" )
        target_root( "lib" )
    end
    target( "my" )

    # If any other architecture of CPU used, different from Intel x86,
    # it's name should be defined in "arch" tag.
    if "x86" == toolset.tag( "arch", "x86" )
        ...
    else
        ...
    end
    ...
}
```

6.1.2 obj_placement conception

Mxx_ru allows to control a place to store C/C++ project build results in. *obj_placement* is used for that purposes — it's an object of a class inherited from `MxxRu::Cpp::ObjPlacement`. That object tells to Mxx_ru where to store object files, libraries, compiled resources and so on.

Project may define the type of obj_placement it needs. `obj_placement` should be used for that:

```
MxxRu::Cpp::dll_target( "my.rb" ) {
    ...
```

[1] Other values will be added according to adaptation of Mxx_ru to other platforms.

```
3     obj_placement(
4         MxxRu::Cpp::RuntimeSubdirObjPlacement.new(
5             "output" ) )
6     ...
7 }
```

Global obj_placement may be defined using `global_obj_placement` function. Action of obj_placement will be extended to all subprojects in that case.

A try to use global and local obj_placement simultaneously is erroneous and exception would be thrown. It is recommended to set a global obj_placement only in a top, composite project.

There are following classes for obj_placement in Mxx_ru by default:

- `MxxRu::Cpp::SourceSubdirObjPlacement`, placing all object files (.obj, .o) to a subfolder of source file folder. By default, `o` folder is used. For instance, for `src/win/init.cpp` source file, object file would be created in `src/win/o`. That folder will be created if not exists. Compiling of resources on Windows platform work the same way. The placement of all other results (exe, lib, dll, import library etc) are defined based on target definition related to current folder.

- `MxxRu::Cpp::RuntimeSubdirObjPlacement`, placing all compilation results to a folder with a name based on runtime type. And source folder tree is recreated inside of that folder.

- `MxxRu::Cpp::CustomSubdirObjPlacement`, which places all final build results (exe/dll(so)/lib(a)) in one folder, but intermediate results (obj(o)/res) in a second, duplicating the source folders structure in it.

For instance, for that directory tree:

```
|-- engine
|-- interface
|   |-- high
|   `-- low
`-- monitor
```

and `SourceSubdirObjPlacement` used, the tree would become like this:

```
|-- engine
|   `-- o
|-- interface
|   |-- high
|   |   `-- o
|   |-- low
|   |   `-- o
|   `-- o
`-- monitor
    `-- o
```

But if `RuntimeSubdirObjPlacement` used with `output` subfolder as a root and debug runtime mode, then the tree would be like that:

```
|-- engine
|-- interface
|   |-- high
|   `-- low
|-- monitor
`-- output
    `-- debug
        |-- engine
        |-- interface
        |   |-- high
        |   `-- low
        `-- monitor
```

And all compilation results, including exe, dll, lib etc, would be placed inside `output/debug`.

If `CustomSubdirObjPlacement` used with parameters `out32` (for final results) and `tmp32` (for intermediate), then the tree would be like that:

```
|-- engine
|-- interface
|   |-- high
|   `-- low
|-- monitor
|-- out32
`-- tmp32
    |-- engine
    |-- interface
    |   |-- high
    |   `-- low
    `-- monitor
```

`SourceSubdirObjPlacement` is used, if obj_placement was not defined explicitly.

6.1.3 Targets for C/C++ projects

These target classes are defined for C/C++ projects in Mxx_ru:

- `MxxRu::Cpp::ExeTarget` for building application targets (executable files). Can be created by helper function `MxxRu::Cpp::exeTarget`;
- `MxxRu::Cpp::LibTarget` for building static libraries (lib files). Can be created by helper function `MxxRu::Cpp::lib_target`;
- `MxxRu::Cpp::DllTarget` for building shared libraries (dll or so files) and import libraries for them. Can be created by helper function `MxxRu::Cpp::dll_target`;
- `MxxRu::Cpp::MacOSBundleTarget` for building shared libraries in the form of MacOS bundles. Can be created by helper function `MxxRu::Cpp::macos_bundle_target`;

- `MxxRu::Cpp::CompositeTarget` for composite projects. Can be created by helper function `MxxRu::Cpp::composite_target`.
- `MxxRu::Cpp::LibCollection` for building and using several libraries (static of shared) at the same time. Can be created by helper function `MxxRu::Cpp::lib_collection_target`.

Object of one of that classes should be created and passed into `MxxRu::setup_target` function for C/C++ projects.

All classes have constructors allowing to get a code block (it's a Ruby language specific). So there is no need to create a new class inherited from one of above. It's enough to create an object of existing class and pass a code block to it's constructor, with all tuning inside that block:

```
1 MxxRu::Cpp::exe_target( ... ) { ... }
```

Note. To ensure Ruby knows that code block is a parameter, opening brace should be placed on the same line with a closed round brace of constructor call. For example:

```
1  # Right.
2  MxxRu::Cpp::exe_target { target( "my" ) c_source( "my.c" ) }
3
4  # Right.
5  MxxRu::Cpp::exe_target { target( "my" )
6      c_source( "my.c" ) }
7
8  # Right.
9  MxxRu::Cpp::exe_target {
10     target( "my" )
11     c_source( "my.c" )
12 }
13
14 # Wrong!
15 MxxRu::Cpp::exe_target
16     {
17         target( "my" )
18         c_source( "my.c" )
19     }
20
21 # Right. Reversed slash at the line end means
22 # the next line is a continue of current
23 MxxRu::Cpp::exe_target \
24     {
25         target( "my" )
26         c_source( "my.c" )
27     }
```

A bit more information about MxxRu::Cpp::LibCollection

Sometimes it is necessary to enumerate the same libraries in several project files:

```
MxxRu::Cpp::exe_target {
  ...
  required_prj 'cfg/prj.rb'
  required_prj 'utils/prj.rb'
  required_prj 'logging/prj.rb'
  ...
}
```

It is possible to create a project with target type `MxxRu::Cpp::LibCollection` to avoid repetition of such enumerations. That project will contain all necessary subprojects. For example file `lib/common.rb`:

```
MxxRu::Cpp::lib_collection_target {
  required_prj 'cfg/prj.rb'
  required_prj 'utils/prj.rb'
  required_prj 'logging/prj.rb'
}
```

A then that file is included via `required_prj`:

```
MxxRu::Cpp::exe_target {
  ...
  required_prj 'lib/common.rb'
  ...
}
```

And this will automatically add libraries `cfg/prj.rb`, `utils/prj.rb` and `logging/prj.rb` to the current project.

The target type `MxxRu::Cpp::LibCollection` distinguishes from `MxxRu::Cpp::CompositeTarget` in that way: `LibCollection` builds full list of all necessary libraries which should be linked with target project.

6.1.4 An order of build/clean execution

During a build of C/C++ targets these actions are performed:

1. All subprojects build is performed.
2. `build` method is called for all source code generators.
3. C/C++ dependency analyzer is executed.
4. Compilation of all C files is performed.
5. Compilation of all C++ files is performed.
6. Resources are compiled if defined (on Windows platform).
7. In case of lib target, static library is built. In case of application and shared library (Mac OS bundle) target, link is performed. If import library place was defined for shared library, import library is built.

During a clean of C/C++ targets these actions are performed:

1. All subprojects clean is performed.
2. `clean` method is called for all source code generators.
3. Removing all object files for all C files.
4. Removing all object files for all C++ files.
5. Compiled resources are removed if were defined (on Windows platform).
6. In case of lib, application and shared library (Mac OS bundle) target, target files are removed. If import library place was defined for shared library, import library is also removed.

Note. `CompositeTarget` class is building and cleaning subprojects only.

6.1.5 Runtime modes

Three runtime modes are allowed in Mxx_ru for C/C++ projects:

- debug. Debug runtime libraries are used. Debug information is added to target;
- release. Release runtime libraries are used. Optimization is turned on. NDEBUG macro is defined;
- default. Release runtime libraries are used. Optimization is turned off. NDEBUG macro is not defined (so asserts would work). This mode is used by default, if no other mode was defined.

Runtime mode can be defined in two ways:

1. `runtime_mode` function in project file. For example:

```
MxxRu::Cpp::exe_target {
    runtime_mode( MxxRu::Cpp::RUNTIME_RELEASE )
    ...
}
```

2. In command line using `--mxx-cpp-release` or `--mxx-cpp-debug` argument. For example:

```
ruby build.rb --mxx-cpp-release
```

6.1.6 Local, global and upspread parameters

C/C++ project definition in Mxx_ru consists of parameters enumeration, such as target file name, source file names, libraries and define-symbols, compiler and linker options, etc. Some of them, such as source file names, belongs to project itself. But other part can be divided in three parts:

Local parameters applied to the project itself. They have no effect on subprojects, nor projects using that project as a subproject.

Here is an example of local define symbol:

```
MxxRu::Cpp::dll_target {
    target( "my_dll" )
    implib_path( "lib" )

    cpp_source( "my_dll/impl.cpp" )

    defines( "MY_DLL_PRJ" )
}
```

Symbol MY_DLL_PRJ, if set this way, will be available only during a build of `my_dll/impl.cpp`.

Global parameters are applied to all projects.

Global parameters are defined using a functions with names like `global_<name>`, where "name" is a name of function used to set a local parameter. For instance, global_define, global_include_path, global_compiler_option etc.

Here is an example of global include path for a composite project:

```
MxxRu::Cpp::composite_target( MxxRu::BUILD_ROOT ) {

    global_include_path( "." )

    required_prj( "engine/prj.rb" )
    required_prj( "interface/prj.rb" )
}
```

Current folder would be used as an include directory for all files during a build of both `engine/prj.rb` and `interface/prj.rb` projects.

Here is an example of global compiler option:

```
MxxRu::Cpp::dll_target( "packer/prj.rb" ) {
    ...
    # 4 bytes alignment should be used.
    if "vc" == toolset.name
        global_compiler_option( "-Zp4" )
    end
    ...
}
```

`-Zp4` compiler option would be used for Visual C++ compiler for all projects, combined within a common project with `packer/prj.rb`.

Upspread parameters are applied for current project and all projects using that project as a subproject (no matter how deep they are include each other).

Upspread parameters are set with the same functions as local are, but with a second argument set to a `MxxRu::Cpp::Target::OPT_UPSPREAD` constant.

Here is an example of upspread define-symbol and upspread include path:

```
MxxRu::Cpp::lib_target( "pcre/prj.rb" ) {
    target_root( "lib" )
    target( "pcre.4.5.0" )

    c_source( "get.c" )
    c_source( "maketables.c" )
    c_source( "pcre.c" )
    c_source( "study.c" )

    define( "PCRE_STATIC", MxxRu::Cpp::Target::OPT_UPSPREAD )
    define( "SUPPORT_UTF8", MxxRu::Cpp::Target::OPT_UPSPREAD )

    include_path( "pcre", MxxRu::Cpp::Target::OPT_UPSPREAD )
}
```

During a build of all projects, directly or indirectly used `pcre/prj.rb` as a subproject, `PCRE_STATIC` and `SUPPORT_UTF8` define symbols would be used, and `pcre` folder would be added to their include folders list.

6.1.7 mxx-cpp-1 argument

Build and clean operation are not applied to subprojects if `--mxx-cpp-1` argument was passed into Ruby interpreter.

Argument `--mxx-cpp-1` is useful for a single subproject build inside a big composite project. The whole composite could take too much time to be built. It's wasteful especially in a case when recompilation and relinking are not needed for all other subprojects. Subproject name and `--mxx-cpp-1` argument are passed into Ruby interpreter instead of composite project name in that case:

```
ruby some/project/prj.rb --mxx-cpp-1
```

`--mxx-cpp-1` argument can be used for *clean* (and *rebuild*) operation also — only subproject's files would be cleaned (rebuilt) in that case.

```
ruby some/project/prj.rb --mxx-cpp-1 --mxx-clean
ruby another/project/prj.rb --mxx-cpp-1 --mxx-rebuild
```

6.1.8 mxx-cpp-no-depends-analyzer argument

`--mxx-cpp-no-depends-analyzer` argument [2] forces Mxx_ru not to analyze C/C++ dependencies in a source files. That analysis is performed by default, that may slowdown build process significantly. Sometimes (with a partial rebuild of a project, as example), it may be optimal to ignore it. `--mxx-cpp-no-depends-analyzer` argument should be passed to Ruby interpreter in that case:

```
ruby some/project/prj.rb --mxx-cpp-no-depends-analyzer
```

6.1.9 mxx-cpp-extract-options

`--mxx-cpp-extract-options` argument forces Mxx_ru to show options list of compiler/linker instead of project build. For example:

```
ruby so_4/prj.rb --mxx-cpp-extract-options
```

for `vc7` toolset may print the following:

```
C Compiler Options:

C++ Compiler Options:
  -EHsc
  -GR
Compiler options:
  -nologo
  -MD
  -LD
  -D__WIN32__
  -DSO_4_PRJ
  -DSO_4_DLL
  -DMXX_RU_ACE__PLATFORM_WIN32
  -DWIN32
  -DMXX_RU_ACE__ACE_HAS_STANDARD_CPP_LIBRARY
  -I.
  -Iso_4/zlib
Defines:
  __WIN32__
  SO_4_PRJ
  SO_4_DLL
  MXX_RU_ACE__PLATFORM_WIN32
  WIN32
  MXX_RU_ACE__ACE_HAS_STANDARD_CPP_LIBRARY
Include Paths:
  .
  so_4/zlib
Librarian Options:
  /NOLOGO
Linker Options:
  /NOLOGO
```

[2] Added in a 1.0.9 version.

```
  /SUBSYSTEM:CONSOLE
MSWindows Resource Compiler Defines:

MSWindows Resource Compiler Include Paths:

MSWindows Resource Compiler Options:
  /d__WIN32__
  /dSO_4_PRJ
  /dSO_4_DLL
  /dMXX_RU_ACE__PLATFORM_WIN32
  /dWIN32
  /dMXX_RU_ACE__ACE_HAS_STANDARD_CPP_LIBRARY
  /i.
  /iso_4/zlib
MSWindows Resource Linker Options:
```

This mode may be useful if project, built by Mxx_ru should be linked against project, built in other build system.

6.2 Mxx_ru setup for C/C++ projects

`MXX_RU_CPP_TOOLSET` environment variable should be defined to use Mxx_ru features with a C/C++ projects. The syntax used to define this environment variable is as follows:

```
MXX_RU_CPP_TOOLSET="<file> [tag=value [tag=value [...]]"
```

where `<file>` – is the name of .rb-file from Mxx_ru, responsible for C/C++ toolset object creation. Toolset names supported are enumerated in a 6.5 table.

The pairs tag and value would be defined as a tags for selected C/C++ toolset.

Examples:

```
export MXX_RU_CPP_TOOLSET="gcc_linux unix=linux arch=x86"
```

```
set MXX_RU_CPP_TOOLSET=vc7
```

6.3 Toolset access

Toolset object can be accessed using a `toolset` method of a `MxxRu::Cpp::Target` class. For example:

```
1 MxxRu::Cpp::exe_target( "my/prj.rb" ) {
2     ...
3     if "vc" == toolset.name
4         ...
5     end
6     ...
7 }
```

Note Since `toolset` method is not static, it can be executed only for created target object. For instance, it was used inside a code block passed into the constructor in example above.

6.4 Runtime mode selection inside project file

Runtime mode may be defined in a project file using `runtime_mode` method. The values for runtime modes are defined as a constants: `MxxRu::Cpp::RUNTIME_DEBUG`, `MxxRu::Cpp::RUNTIME_DEFAULT`, `MxxRu::Cpp::RUNTIME_RELEASE`. By default, `MxxRu::Cpp::RUNTIME_DEFAULT` is used.

Runtime mode is a global parameter, so it's applied to all projects at once. Exception would be thrown if two projects will try to set different runtime modes.

Current runtime mode is available through `mxx_runtime_mode` method.

See also 6.1.5 on page 41.

6.5 Runtime library type selection

On platforms with shared libraries supported, some toolsets allow to select one of two runtime library (RTL) types: static (RTL code is statically linked into application) and shared (RTL code is inside shared library, linked against application). If application have it's own shared libraries and exports C++ objects created in one library into another for deletion, shared RTL should be used (all the shared libraries have a common heap in that case).

Type of RTL may be defined using a `rtl_mode` method. The values for RTL type are defined as a constants: `MxxRu::Cpp::RTL_DEFAULT` (project have no preference about RTL type used), `MxxRu::Cpp::RTL_STATIC` (static RTL should be used), `MxxRu::Cpp::RTL_SHARED` (shared RTL should be used).

`MxxRu::Cpp::RTL_DEFAULT` mode is used by default.

RTL type is a global parameter, so it's applied to all projects at once. An exception would be thrown on a try of two projects to define different types of RTL.

If RTL type wasn't defined (in default mode), toolset will determine it. Usually, default compiler's RTL type is used.

An example:

```
1 MxxRu::Cpp::dll_target {
2     target_root( "lib" )
3     target( "my" )
4
5     rtl_mode( MxxRu::Cpp::RTL_SHARED )
6     ...
7 }
```

Current RTL mode is available through `mxx_rtl_mode` method.

6.6 Multi-threading mode selection

On some platforms (Microsoft Windows for example), it's required to explicitly define a type of threading mode used in a project. `threading_mode` method should

be used to set threading mode in Mxx_ru. The values for threading mode are defined as a constants: `MxxRu::Cpp::THREADING_DEFAULT` (project have no preference about threading modes. It's a singlethreaded by itself, but still work well in a multithreaded application), `MxxRu::Cpp::THREADING_MULTI` (multithreading is required), `MxxRu::Cpp::THREADING_SINGLE` (singlethreading is required, the project can't work in a multithreaded environment). By default, `MxxRu::Cpp::THREADING_DEFAULT` mode is used.

If threading mode is not equal to `MxxRu::Cpp::THREADING_MULTI` and compiler has a requirement to set threading mode explicitly, singlethreaded mode is used.

Threading mode is a global parameter, so it's applied to all projects at once. An exception would be thrown if two projects will try to define different threading modes.

```
MxxRu::Cpp::dll_target {
    target( "threads_1.4.0" )

    rtl_mode( MxxRu::Cpp::RTL_SHARED )
    threading_mode( MxxRu::Cpp::THREADING_MULTI )
    ...
}
```

Current threading mode is available through `mxx_threading_mode` method.

Note. For some compilers, Visual C++ for example, the names for RTL libraries used are depended on both RTL type and threading mode. And compiler may not have libraries for all combinations of RTL types and threading modes. For example, Visual C++ 7.* doesn't have a singlethreaded shared RTL library, and for Visual C++ 8.* RTL libraries for multithreaded mode only are available.

6.7 RTTI mode selection

Run-Time Type Identification (RTTI) is turned off by default on some compilers (Visual C++, for example), or may be turned off (GNU C++, for instance). If project requires RTTI, for safe *`dynamic_cast`* usage, as example, then RTTI mode should be turned on explicitly. Otherwise, if RTTI is not required in a project and code should be as fast as possible, it may be efficient to turn RTTI off.

RTTI mode may be defined with a `rtti_mode` method, using these constants as a parameter: `MxxRu::Cpp::RTTI_DEFAULT` (project have no preference about RTTI usage), `MxxRu::Cpp::RTTI_ENABLED` (RTTI is required) and `MxxRu::Cpp::RTTI_DISABLED`(RTTI is not needed).

RTTI mode is a global parameter, so it's applied to all projects at once. An exception would be thrown if two projects will try to define different RTTI modes.

```
MxxRu::Cpp::dll_target {
    target( "threads_1.4.0" )

    rtl_mode( MxxRu::Cpp::RTL_SHARED )
    threading_mode( MxxRu::Cpp::THREADING_MULTI )
    rtti_mode( MxxRu::Cpp::RTTI_ENABLED )
    ...
}
```

Current RTTI mode is available through `mxx_rtti_mode` method.

6.8 Setting target file name

Note. Projects, described using `MxxRu::Cpp::CompositeTarget` class have no explicit target, so `target_root`, `target`, `target_prefix`, `target_ext/target_suffix` and `implib_path` method calls are ignored.

6.8.1 target_root method

To define a folder containing a target file, `target_root` method should be used. The placement of result is controlled by obj_placement (see 6.1.2 on page 36). If target_root wasn't defined, result would be in a folder chosen by obj_placement. If target_root is defined, result would be in a folder with name defined by `target_root`, inside a folder chosen by obj_placement.

For instance, if obj_placement stores files to current folder, then if target_root was set to `out32`, resulting file would be in `out32` subfolder of current folder.

Important. It's required to call `target_root` method before `target` method call. Then, during a `target` execution the name of a target will be created using the target_root value. If `target_root` method is called after a `target`, then already defined filename is modified. In a general case, if `target_root` and `target` are called only once, results would be identical. But, if `target_root` would be called twice (before and after `target` call), result may be different then expected.

6.8.2 target method

To define a base name of target file, `target` method should be used. Real name would be created with all platform specifics taken into account. For example, for shared library with base name threads_1.4.0 on windows platfotm, threads_1.4.0.dll would be created, and on unix platform — libthreads_1.4.0.so.

Target name, passed into `target` method, should not include any path. `target_root` method should be used to define folder where target should be created, **before** calling a `target` method.

6.8.3 target_prefix method

By default, Mxx_ru follows platform-specific rules for building resulting name of the target. For example, static and shared libraries on Unix must have prefix 'lib' (e.g. 'mylib' becomes 'libmylib'). In some cases it's required to set target name prefix explicitly. For example, for building plug-ins (thus, Nuke expects that name of shared library will be the same with name of plug-in, e.g. without 'lib' prefix). `target_prefix` method allows to do that:

```
1 MxxRu::Cpp::dll_target {
2   target 'scene_builder'
3   target_prefix '' # suppresing 'lib' prefix addition.
4   # scene_builder.so will be built.
5   ...
6 }
```

6.8.4 target_ext/target_suffix method

By default, Mxx_ru detects final file extension from target type and platform. For example, for executable file (exe_target) on Windows platform, .exe extension would be used, and at the same time on Unix platform no extension would be at all. In some cases it's required to set file extension explicitly. For example, for building plug-ins (thus, Autodesk Maya expects .mll extension for plug-ins on windows platform) or during port of Unix applications to Windows (it may be required for dll's have .so extensions). `target_ext` method allows to do that:

```
MxxRu::Cpp::exe_target {
  target 'security_monitor'
  target_ext '.mod'
  # security_monitor.mod will be built.
  ...
}
MxxRu::Cpp::dll_target {
  target 'raytracer'
  target_suffix '.mll'
  # raytracer.mll will be built.
  ...
}
```

Note. Method `target_suffix` is an alias for `target_ext` since version 1.4.2.

6.8.5 implib_path method

First of all, `implib_path` method forces Mxx_ru to build import library (on platforms supporting this feature), and secondly, it defines where this library should be placed.

`implib_path` is used for shared libraries only for a moment. If `implib_path` wasn't called, import library isn't built. [3]

6.8.6 Examples

Application name definition.

```
MxxRu::Cpp::exe_target {
    target( "hello_world" )
    ...
}
```

Placing application inside `utest` folder.

```
MxxRu::Cpp::exe_target {
    target_root( "utest" )
    target( "test_packer" )
    ...
}
```

[3] Visual C++ compiler builds import library when it sees at least one exported symbol, so import library would be built even without definition of `implib_path`. It's recommended to define `implib_path` anyway, for compatibility.

Target shared library should be placed into a current folder on Windows platform, and import library into a lib folder, but on Unix platform, target shared library should be placed into a lib folder also.

```
MxxRu::Cpp::dll_target {
    if "mswin" == toolset.tag( "target_os" )
        implib_path( "lib" )
    elsif "unix" == toolset.tag( "target_os" )
        target_root( "lib" )
    end

    target( "oess_defs.1.3.0" )
    ...
}
```

6.9 Selecting an application type (GUI/Console)

`screen_mode` method is intended for application type definition: it may be windowed or be console. By default, Mxx_ru is creating a console application — special options are defined for compiler and linker. Using a `screen_mode` method, project may be transformed into a windowed application.

`MxxRu::Cpp::SCREEN_WINDOW` constant should be passed into the `screen_mode` method for creating a windowed application, or `MxxRu::Cpp::SCREEN_CONSOLE` constant for console application.

```
MxxRu::Cpp::exe_target {
    ...
    screen_mode( MxxRu::Cpp::SCREEN_WINDOW )
    ...
}
```

screen_mode parameter is a local parameter for a project, and it's not applied nor for child projects, nor for parent projects.

Current application type can be accessed using a `mxx_screen_mode` method.

6.10 Default values for some global parameters

For such parameters, as `runtime_mode`(6.4 on page 46), `rtti_mode`(6.7 on page 47), `rtl_mode`(6.5 on page 46) and `threading_mode`(6.6 on page 46) Mxx_ru uses special default values. For example, if `runtime_mode` is not explicitly set, project would be compiled without optimization and debug information (i.e. not release and not debug). In some cases such behavior is not desirable. For example, to developer it may be more comfortable to build project in release mode by default.

Mxx_ru allows to change default values for given parameters. For this purpose the methods with `default_<parameter>` names are used. For example: `default_runtime_mode`, `default_rtti_mode`, etc.

It is supposed, that these methods will be used in `build.rb` for tuning of the whole project. But they also can be used inside project files for subprojects of the large compound

project. In this case some project files may try to change a default value for the same parameter. In this case Mxx_ru acts as follows:

- If all projects establish the same value, Mxx_ru simply uses this value;
- If the projects try to establish different default values, Mxx_ru takes only first call to `default_*` into account, ignoring all others, displaying the warning of the conflict of default values.

Because Mxx_ru uses only first value at the presence of the conflicts, it is recommended to set default values in `build.rb`, to be sure such changes were carried out in one place.

As example it is possible to show how to establish default value for `runtime_mode` (Release), `rtl_mode` (Shared RTL) and `rtti_mode` (enabled) for compound project:

```
require 'mxx_ru/cpp'

MxxRu::Cpp::composite_target( MxxRu::BUILD_ROOT ) {
  ...
  default_runtime_mode( MxxRu::Cpp::RUNTIME_RELEASE )
  default_rtl_mode( MxxRu::Cpp::RTL_SHARED )
  default_rtti_mode( MxxRu::Cpp::RTTI_ENABLED )
  ...
}
```

It is important to understand, that the changed default values are used Mxx_ru only if the appropriate mode was not changed obviously. For example, default value for `runtime_mode` is used only if `--mxx-cpp-release` and `--mxx-cpp-debug` keys were not specified in the command line.

6.11 Source files definition

6.11.1 sources_root method

By default, source files are defined in relation to project's file folder. In other words, if project alias is `interface/prj.rb`, then

```
c_source( "main.c" )
cpp_source( "io.cpp" )
```

construction will force Mxx_ru to search for `interface/main.c` and `interface/io.cpp` files. That way is handy for small projects, but sometimes there is a need to explicitly define where to search for source files:

1. When sources are placed in other branch of project file structure. For example, all project files are placed in `mxxru_prjs`, but source files are inside `src`[4]. Then `sources_root` method should be called with a single parameter:

[4]This may be handy when project allows to compile itself using different tools.

```
MxxRu::Cpp::exe_target( "mxxru_prjs/interface.rb" ) {
    ...
    sources_root( "src/interface" )

    c_source( "main.c" )
    cpp_source( "io.cpp" )
    ...
}
```

2. When project file structure is too deep. It's inconvenient to define full path for each source file. `sources_root` method may be called with two parameters: folder name and code block. In that case `sources_root` method works this way: it concatenates current source_root value with the first argument and sets result as a new sources_root parameter. After that it runs code block was passed to it and restores previous source_root value. And this even allows to call `sources_root` inside that code block:

```
MxxRu::Cpp::dll_target( "oess_1/db/prj.rb" ) {
    ...
    sources_root( "impl" ) {
        # For files in folder oess_1/db/impl.
        ...
        sources_root( "storage" ) {
            # For files in folder oess_1/db/impl/storage.
            ...
            sources_root( "impl" ) {
                # For files in folder oess_1/db/impl/storage/impl.
                ...
            }
        }
    }
    sources_root( "site" ) {
        # For files in folder oess_1/db/site.
        ...
        sources_root( "impl" ) {
            # For files in folder oess_1/db/site/impl.
            ...
        }
    }
    ...
}
```

Both methods may be combined. For example, you may call `sources_root` with one parameter for a first time, and then call it with two parameters:

```
MxxRu::Cpp::exe_target( "mxxru_prjs/interface.rb" ) {
    sources_root( "src/interface" )

    # files from src/interface folder.
    c_source( "main.c" )
    cpp_source( "io.cpp" )
```

```
7
8       sources_root( "debug" ) {
9           # files from src/interface/debug folder.
10          cpp_sources( "io_dumper.cpp" )
11          ...
12      }
13      ...
14  }
```

Current value of `sources_root` can be obtained by method `mxx_sources_root`.

6.11.2 c_source(s), cpp_source(s) methods

`c_source` and `cpp_source` methods are intended for including source files into a project for C and C++ languages correspondingly. It's important to define all files containing C code only (usually they have `.c` extension) using a `c_source` method, and files containing C++ code (usually they have `.C`, `.cc`, `.cp`, `.cpp`, `.cxx`, or `.c++` extensions) using a `cpp_source` method. Thats important because different compilers may be used for them on some platforms (for example, `gcc` and `g++` for GNU C++). On other platforms, special compiler parameters are required to tell compiler about type of code passed into it. Mxx_ru considers all these features and actively uses them.

`c_source` and `cpp_source` have two arguments: first is required and second is optional. The first is source file name. Second may contain compiler options, added to selected by Mxx_ru during a compilation of that file only.

Usually, only one argument is passed:

```
1   MxxRu::Cpp::dll_target {
2
3       required_prj( "oess_1/defs/prj.rb" )
4
5       Oess_1::setup_platform( self )
6
7       target( "oess_io" + Oess_1::VERSION )
8
9       define( "OESS_1__IO__PRJ" )
10
11      cpp_source( "stream.cpp" )
12      cpp_source( "binstream.cpp" )
13      cpp_source( "binbuffer.cpp" )
14      cpp_source( "subbinstream.cpp" )
15      cpp_source( "mem_buf.cpp" )
16      cpp_source( "fixed_mem_buf.cpp" )
17      cpp_source( "bstring_buf.cpp" )
18  }
```

Two parameters are used if some file requires specific compiler options, define symbol, for example:

```
1   ...
2   cpp_source( "interface.cpp", [ "-DLOG_LEVEL=3" ] )
```

```
cpp_source( "engine.cpp" )
...
```

`LOG_LEVEL` would be defined during compiling of `interface.cpp` only, but not for an `engine.cpp`.

Two parameter version of `c_source` and `cpp_source` methods should be used carefully, considering following factors:

- If project may be compiled with more than one compiler, different options may be required. In other words, all these options should be formed inside a project file for each compiler;
- Mxx_ru doesn't performs any check about parameters passed using these methods, and doesn't have any check about possible conflicts with parameters, already chosen by Mxx_ru. For example, if Mxx_ru considers that multithreading project with shared library requires `-MD` option for Visual C++. But there would no detection of a conflict, if `-MT` option would be passed using `c_source` or `cpp_source` methods;
- in previous Make++ versions was no way to define a single-file specific compiler options, so this feature is experimental. Because of that, there is no guaranty it wouldn't be changed somehow in a future. So be ready to rewrite calls of `c_source` and `cpp_source` methods used two arguments.

Methods `c_sources`, `cpp_sources` are analogous to just described methods, but have only one argument of `Enumerable` type. This allows to set several source file names at once (in an array, for example):

```
cpp_sources [ 'interface.cpp', 'engine.cpp' ]
```

This may be useful in combination with `Dir::glob`:

```
cpp_sources Dir.glob( 'some/prj/**/*.cpp' )
```

NOTE: When using `Dir::glob` method it's important to remember, that `Dir::glob` thinks that it's argument defines a search path from current folder, because `Dir::glob` doesn't know anything about `sources_root`. Therefore, the following code is erroneous:

```
MxxRu::Cpp::dll_target( 'some/deep/path/prj.rb' ) {
  ...
  cpp_sources Dir.glob( '**/*.cpp' )
  ...
}
```

because `Dir::glob` would search all *.cpp files starting from current folder, but not from `some/deep/path` folder. To avoid this, you may use `mxx_sources_root` method:

```
MxxRu::Cpp::dll_target( 'some/deep/path/prj.rb' ) {
  ...
  cpp_sources Dir.glob( "#{mxx_sources_root}/**/*.cpp" )
  ...
}
```

6.11.3 mswin_rc_file method

`mswin_rc_file` method is intended to define a name of resource file on Windows platform. If it was set, Mxx_ru executes a resource compiler during a project build to get a `.res` file, and then links it to the target.

`mswin_rc_file` method have two parameters: first is required and second is optional. First defines the name of `.rc` file. Second may contain file names, required to build resource file. For example, `.ico` or `.bmp` files.

During a use of `mswin_rc_file` method, remember that first parameter value is calculated considering a sources_root value. But all dependency names should be defined using a full path.

Example:

```
MxxRu::Cpp::exe_target( "interface/prj.rb" ) {
    ...
    screen_mode( MxxRu::Cpp::SCREEN_WINDOW )
    ...
    mswin_rc_file( "resources.rc",
        [ "interface/res/mainframe.ico",
            "interface/res/document.ico",
            "interface/res/toolbar.bmp",
            "interface/res/reources.rc2" ] )
    ...
}
```

Note. The value, defined by `mswin_rc_file` method is taken into account on Windows platform only.

6.12 Additional object files linking

`obj_file` method is intended for defining additional object files for a target to be linked against. By default, Mxx_ru have a list of object files, created during a compilation of source files, and uses them during a link. If project is required to include already compiled object file (for example it may be created using other programming language compiler), the name of file required should be passed into an `obj_file` method.

For example:

```
MxxRu::Cpp::exe_target {
    ...
    required_prj( "interface/asm/fast_transform/prj.rb" )
    ...
    obj_file( "interface/asm/fast_transform/linear.obj" )
    ...
}
```

Mxx_ru uses an `obj_file` method in toolset realization for saving an object file names of a project, produced during a compilation of source files.

6.13 Additional libraries linking

`lib` method is intended for defining an additional libraries, target need to link against. By default, Mxx_ru have a list of libraries required from subprojects. But often you need to define a specific (usually platform-dependent) library to link against. `lib` method should be used in that case:

```
MxxRu::Cpp::dll_target {
    ...
    if "mswin" == toolset.tag( "target_os" )
        lib( "winsock" )
    elsif "unix" == toolset.tag( "target_os" )
        lib( "socket" )
        if "bsd" != toolset.tag( "unix_port" )
            lib( "pthread" )
        end
    end
    ...
}
```

`lib` method have two arguments. The first defines library name should be passed into a linker in a way it should be passed. In other words, on Windows platform it should be the name with an extension, but on unix platforms it should be the name without an extension and `lib` prefix.

Second argument of a `lib` method defines a path linker should search for that library. By default, that argument is equal to `nil` value, i.e. no path is passed into the linker. In that case linker would perform search in standard paths for specific platform. But if second parameter is not equal to `nil`, then value defined would be passed into the linker as a search path. For example, if you need to define `/usr/local/share/mysec/lib/libtdes.3.4.a` library to link against a target on a unix platform, you need to call `lib` method that way:

```
lib( "tdes.3.4", "/usr/local/share/mysec/lib" )
```

6.13.1 Selecting static or dynamic libraries explicitly

In 1.3 version two new methods are implemented for `BinaryTarget`[5]: `lib_static` `lib_shared`. On Windows platform, they are simple aliases of `lib` method. But on Unix platform they are intended to say linker to which type of a library (static or dynamic) it should link against.

On Unix systems it's common to have the same library (let's call it "A") as a static (named `libA.a`) and shared (named `libA.so`). When library name is passed to the linker in `-l` option (`-lA`), linker chooses by itself, what type it would take. Thus, by default on linux, GCC selects shared library (i.e. `libA.so`). But, if `-static` option would be passed to the GCC, then `-lA` would take static version (`libA.a`).

In some cases developer may want to explicitly set that it's required to use exactly static (or, exaclty shared) library. To set requirement of static library, `lib_static` method was added, and to set requirement of shared library, `lib_shared` was added:

[5] It's a base class for all C/C++ targets.

```
require 'mxx_ru/cpp'

Mxx_ru::Cpp::exe_target {
  target 'some_target'
  ...
  lib 'engine' # Linker may choose or libengine.a,
               # or libengine.so.
  lib_static 'config' # Linker is forced to choose libconfig.a
                      # even if libconfig.so exists.
  lib_shared 'gui' # Linker is forced to choose libgui.so
                   # even if libgui.a exists.
  ...
}
```

For GCC on Unix a call to `lib('A')` method is expanded to `-lA`. A call to `lib_static('A')` by default is expanded to `-Wl,-Bstatic,-lA,-Bdynamic` if linker uses `-Bdynamic` mode by default. Accordingly, a call to `lib_shared('A')` would be expanded to `-Wl,-Bdynamic,-lA,-Bstatic`, if linker uses `-Bstatic` mode by default.

For GCC on Unix Mxx_ru v.1.3 assumes linker is always prefers shared libraries (i.e. uses `-Bdynamic` mode by default). If it's required to set static mode as default, then `lib_linking_mode` tag should be used with `static` value during toolset configuration:

```
export MXX_RU_CPP_TOOLSET='gcc_linux lib_linking_mode=static'
```

If Mxx_ru detects that the same library is used as both static and shared (i.e. in one place it's defined with `lib_static`, but in other place with `lib_shared` inside a project), then Mxx_ru throws an exception and stops.

After adding `lib_static`/`lib_shared` methods in Mxx_ru, the following side effect is appeared: Now, a connection of static library project with `required_prj` is equivalent to usage of `lib_static` method. Accordingly, a connection of shared library project with `required_prj` is equivalent to usage of `lib_shared` method.

```
# A/prj.rb
MxxRu::Cpp::lib_target { target 'A' ... }

# B/prj.rb
MxxRu::Cpp::dll_target { target 'B' ... }

# C/prj.rb
MxxRu::Cpp::exe_target {
  target 'C'
  required_prj 'A/prj.rb'
  required_prj 'B/prj.rb'
  ...
}
```

is equivalent to:

```
# C/prj.rb
MxxRu::Cpp::exe_target {
  target 'C'
  lib_static 'A'
  lib_shared 'B'
  ...
}
```

In 1.3 version one question was left unresolved: If user wants his application to be linked to only static versions of the libraries, Mxx_ru have no separate method to do that. The only way to do that is to define `-static` value in `link_option`. But in that case Mxx_ru can't warn if somewhere `lib_shared` method would be used.

6.13.2 Enumeration of search paths for the libraries

Mxx_ru automatically gathers folder names to search for the libraries, if these names are passed as a second parameter of `lib` method. In some cases this requirement to pass the search path to `lib` method may not be ideal. For example, it may reqiure to define the following code:

```
case toolset.tag( 'target_os' )
  when 'mswin'
    lib 'engine', 'tools/engines/lib'
    lib 'parser', 'tools/parsers/lib'
  when 'unix'
    lib 'engine', '/usr/local/lib'
    lib 'parser', '/usr/local/lib'
end
```

A situation with `lib` method (as well as with `lib_static` and `lib_shared`) becomes more worse, if the library may be placed in one of several folders (for example, `/usr/lib`, `/usr/local/lib`), but exactly isn't known where it is. In this case it's unclear which path should be defined in `lib` method.

To solve this problems, `lib_path` method was added into Mxx_ru, which allows to define search path for the libraries. Using it, an example above can be rewriten as the following:

```
case toolset.tag( 'target_os' )
  when 'mswin'
    lib_path 'tools/engines/lib'
    lib_path 'tools/parsers/lib'
  when 'unix'
    lib_path '/usr/local/lib'
end

lib 'engine'
lib 'parser'
```

Also, `lib_paths` method exists, which have `Enumerable` argument:

```
case toolset.tag( 'target_os' )
  when 'mswin'
    lib_paths [ 'tools/engines/lib', 'tools/parsers/lib' ]
  when 'unix'
    lib_paths [ '/usr/local/lib', '/usr/lib', '/usr/share/lib' ]
end

lib 'engine'
lib 'parser'
```

Method `lib_paths` can be combined with the features of standard Ruby library and algorithmic features of Ruby language itself. For example, the following fragment defines as search paths all subfolders that have *.lib files inside:

```
lib_paths Dir['**/*.lib'].inject([]) { |r,f| r << File.dirname(f) }.uniq
```

6.14 Optimization mode selection

Optimization mode may be selected using `optimization` method. Mxx_ru considers optimization mode during assemble of compiler options for release mode (see 6.1.5 on page 41). Optimization by speed is used by default.

These constants should be used to define optimization mode: `MxxRu::Cpp::OPTIM_SIZE` (Optimization by size of code generated) and `MxxRu::Cpp::OPTIM_SPEED` (optimization by speed of code generated).

Optimization mode is a local project parameter, so defined optimization mode have no effect nor child, nor parent projects.

6.15 Functions for setting local, global and upspread parameters

As can be seen in 6.1.6 on page 42, functions to define local, upspread and global project parameters are usually look like this:

```
<parameter>( a_value, a_type )
global_<parameter>( a_value )
```

where `<parameter>` — is the name of parameter. First function is used to define local or upspread parameter. The value of the parameter would be a local if `a_type` argument would have `MxxRu::Cpp::Target::OPT_LOCAL` value, or wouldn't be defined. If `a_type` argument would have the `MxxRu::Cpp::Target::OPT_UPSPREAD` value, then the value of the parameter would be automatically spread to all projects, requiring this project. Function with a `global_` prefix is used to define a global parameter, applied to all projects.

Functions, intended for defining a local, upspread and global parameters are described in this section

6.15.1 include_path, global_include_path

Functions `include_path` and `global_include_path` are used to define search path(s) for header files.

6.15.2 define, global_define

Functions `define` and `global_define` are used to set preprocessor define symbols, used during compilation of C/C++ files.

6.15.3 compiler_option, global_compiler_option

Functions `compiler_option, global_compiler_option` are used to define additional compiler options for both C and C++ files.

6.15.4 c_compiler_option, global_c_compiler_option

Functions `c_compiler_option, global_c_compiler_option` are used to define additional compiler options for C files only.

6.15.5 cpp_compiler_option, global_cpp_compiler_option

Functions `cpp_compiler_option, global_cpp_compiler_option` are used to define additional compiler options for C++ files only.

6.15.6 linker_option, global_linker_option

Functions `linker_option, global_linker_option` are used to define additional linker options.

6.15.7 librarian_option, global_librarian_option

Functions `librarian_option, global_librarian_option` are used to define additional librarian options, used to create a static libraries.

6.15.8 Resource compiler on Microsoft Windows platform

mswin_rc_include_path, global_mswin_rc_include_path

Functions `mswin_rc_include_path`, `global_mswin_rc_include_path` are used to define header search path(s) for resource compiler.

mswin_rc_define, global_mswin_rc_define

Functions `mswin_rc_define`, `global_mswin_rc_define` are used to set preprocessor define symbols, defined during resource compilation.

mswin_rc_option, global_mswin_rc_option

Functions `mswin_rc_option`, `global_mswin_rc_option` are used to pass additional options to resource compiler.

mswin_rlink_option, global_mswin_rlink_option

Functions `mswin_rlink_option`, `global_mswin_rlink_option` are used to pass additional options to resource linker.

6.16 Manifest support for Visual C++ 8/9/10

In Visual C ++ 8.0/9.0/10.0 the next innovation of Microsoft was realized: support of .NET manifests for EXE and DLL targets, which use Shared RTL library from Visual C++. The manifest is a XML file, describing exact libraries including their versions necessary for the application. The manifest file should be or is written manually by developer, or generated by linker during an application link. After that manifest should be or is placed near to EXE/DLL file (additional file with name of a kind ¡target¿.manifest, for example, calculator.exe.manifest or complex.matrix.dll.manifest), or is built in EXE/DLL file with a special tool mt.exe (Manifest Tool). Thus the manifest is located in an EXE/DLL file as a resource and this resource should have one of special identifiers:

- 1 (`CREATEPROCESS_MANIFEST_RESOURCE_ID`) for EXE file;
- 2 (`ISOLATIONAWARE_MANIFEST_RESOURCE_ID`) for a DLL, which is statically linked to the application during a link process using an import library;
- 3 (`ISOLATIONAWARE_NOSTATICIMPORT_MANIFEST_RESOURCE_ID`) for a DLL, which is dynamically loaded through LoadLibrary call.

Since the version 1.1, the support of work with the manifests in Visual C++ 8.0/9.0/10.0 is implemented. Mxx_ru allows:

- to specify rules of work with the manifests by default and these rules will be distributed to all projects, which doesn't have their own manifest rules specified;
- to specify rules of work with the manifests for the concrete project (default rules will not be distributed to such project);
- to force linker to generate the manifest during a link process (if it is not done, the developer should independently give the manifest for his EXE/DLL file);
- to use Manifest Tool for embedding the manifest into an EXE/DLL file (as with an automatic choice of the appropriate identifier of resources, and with the obviously given identifier).

6.16.1 Installation of the rules of work with the manifests

Installation of the rules of work with the manifests is carried out with the help of a `MxxRu::Cpp::Toolsets::Vc8::manifest` method. As parameter this method receives Hash with values describing rules of work with the manifest.

To set rules for the concrete project, it is necessary to call a `MxxRu::Cpp::Toolsets::Vc8::manifest` method in a project file of the given project, having a `:target` parameter set:

```
MxxRu::Cpp::exe_target {
  ...
  target( 'calculator' )
  MxxRu::Cpp::Toolsets::Vc8::manifest(
      :target => self,
      :autogen => :to_default_file )
  ...
}
```

Such description specifies that for `calculator.rb` project the manifest should be generated (through `:target => self` construction). The manifest will be generated by a linker and will be placed in a `calculator.exe.manifest` file (it is set by `:autogen => :to_default_file`).

If `:target` parameter is not defined during a `MxxRu::Cpp::Toolsets::Vc8::manifest` call, Mxx_ru will consider that the default manifest is set.

```
MxxRu::Cpp::composite_target( MxxRu::BUILD_ROOT ) {
  ...
  MxxRu::Cpp::Toolsets::Vc8::manifest(
      :autogen => :to_default_file )
  ...
}
```

Such description forces Mxx_ru to create manifest with a linker and place the manifests in files with names of a `<target>.manifest` kind for all projects, in which there are no own adjustments for work with the manifests.

The default manifest description may be done in any project. Notice, however, that Mxx_ru will not show any warning if there are several different descriptions would appear. Therefore it is recommended to do this description in one place — in `build.rb` file.

To cancel actions of rules by default it is necessary to call `MxxRu::Cpp::Toolsets::Vc8::manifest` with `nil` value:

```
MxxRu::Cpp::Toolsets::Vc8::manifest( nil )
```

6.16.2 The name of the manifest file for generation of the manifest by a linker

The `:autogen` parameter of a `MxxRu::Cpp::Toolsets::Vc8::manifest` method specifies that manifest file should be generated by a linker. But it is necessary to specify to the linker in what file the generated manifest should be written down. If `:to_default_file` is given as `:autogen` value, then Mxx_ru generates manifest file name by itself (`.manifest` is added to the full target name). Also, as `:autogen` value, file name may be defined. In that case, manifest is written down to the file given. For example, this:

```
MxxRu::Cpp::Toolsets::Vc8::manifest(
    :target => self,
    :autogen => :to_default_file )
```

would force Mxx_ru to generate manifest file name by itself. Whereas that:

```
MxxRu::Cpp::Toolsets::Vc8::manifest(
    :target => self,
    :autogen => 'my.manifest' )
```

would force Mxx_ru to write generated manifest file to `my.manifest` file.

6.16.3 Manifest Tool

If it is required that the manifest should be built-in as a resource in resulting EXE/DLL file, then `:mt` parameter should be defined during a `MxxRu::Cpp::Toolsets::Vc8::manifest` call.

For example, construction:

```
MxxRu::Cpp::Toolsets::Vc8::manifest(
  :target => self,
  :autogen => :to_default_file,
  :mt => {} )
```

specifies, that manifest is generated by linker, the name of manifest file should be generated by Mxx_ru, manifest should be built-in into application by Manifest Tool, and after that generated manifest file should be removed. If developer wants to store manifest file, the description should be changed this way:

```
MxxRu::Cpp::Toolsets::Vc8::manifest(
  :target => self,
  :autogen => :to_default_file,
  :mt => { :keep_manifest_file => true } )
```

The manifest, defined this way, will be built-in into EXE/DLL file with resource identifier, automatically set by Mxx_ru. For EXE files "1" is used as identifier, for DLL files — "2". If there is a need to set identifier explicitly, it may be done this way:

```
MxxRu::Cpp::Toolsets::Vc8::manifest(
  :target => self,
  :autogen => :to_default_file,
  :mt => { :resource_id => :process_manifest } )
```

In that case "1" would be used as a resource identifier. For identifier set to "2", `:resource_id` should be set to `:isolationaware_manifest` value:

```
MxxRu::Cpp::Toolsets::Vc8::manifest(
  :target => self,
  :autogen => :to_default_file,
  :mt => { :resource_id => :isolationaware_manifest } )
```

For identifier set to "3" — `:isolationaware_nostaticimport_manifest`:

```
MxxRu::Cpp::Toolsets::Vc8::manifest(
  :target => self,
  :autogen => :to_default_file,
  :mt => { :resource_id => :isolationaware_nostaticimport_manifest } )
```

If manifest is generated into a user specified file name, then there is no need to put manifest file twice: Mxx_ru will extract the file name needed from `:autogen` parameter:

```
MxxRu::Cpp::Toolsets::Vc8::manifest(
  :target => self,
  :autogen => 'hello_world.exe.manifest',
  :mt => {} )
```

In that case manifest would be built-in from automatically generated `hello_world.exe.manifest` file.

However, if manifest file is not automatically generated, but given by a developer, then file name should be defined explicitly in a `:mt` parameter:

```
MxxRu::Cpp::Toolsets::Vc8::manifest(
  :target => self,
  :mt => { :manifest => 'hello_world.exe.manifest' } )
```

In that case Mxx_ru assumes `hello_world.exe.manifest` file already exists, it is built in EXE file and not removed after embedding.

Different parameters for `:mt` may vary. For example, embedding of automatically generated manifest with resource identifier set to "3" and preserving the generated manifest file is defined that way:

```
MxxRu::Cpp::Toolsets::Vc8::manifest(
  :target => self,
  :autogen => :to_default_file,
  :mt => { :resource_id => :isolationaware_nostaticimport_manifest,
      :keep_manifest_file => true } )
```

6.17 Mac OS bundle support

Note. I haven't expirience with programming for Mac OS. So the following information could be a bit inaccurate. Please take it with care.

Mac OS has a special file type — *bundle*[6]. From Mxx_ru's point of view, bundle is a kind of shared library but with important distinctions:

- bundle should have `.bundle` extension instead of `.so`;
- bundle should be linked with `-bundle` argument for linker (instead of `-dynamic_lib`).

[6] `http://developer.apple.com/documentation/CoreFoundation/Conceptual/CFBundles/CFBundles.html`

A new type of target `MacOSBundleTarget` and auxiliary function `macos_bundle_target` have been added in version 1.4.10 for support Mac OS bundles. With help of them a bundle could be build using such project type:

```
require 'mxx_ru/cpp'

Mxx_ru::Cpp::macos_bundle_target {
  target 'my_test_bundle'
  c_source 'my_test_bundle.c'
}
```

In version 1.4.10 `MacOSBundleTarget` has special meaning only form `gcc_darwin` toolset. For other toolsets that type of target is regarded as ordinal shared library. So if you try to compile above sample with `vc7` toolset you will get ordinal `my_test_bundle.dll` dynamic link library.

6.18 Custom tool names

If the following tags defined in `MXX_RU_CPP_TOOLSET` environment variable then the names which specified by those tags will be used instead of the standard tools' names.

If tags `c_compiler_name` and `cpp_compiler_name` are defined in `MXX_RU_CPP_TOOLSET` then a tool whose name specified by `c_compiler_name` will be used for C files compilation, and for C++ files — a tool whose name specified by `cpp_compiler_name`. If tags `c_compiler_name` and `compiler_name` are defined then a tool with name in tag `c_compiler_name` will be used for C files, and for C++ files — a tool whose name specified by `compiler_name`. If only tag `compiler_name` is specified then that tool will be used for C and for C++ files. If, for example, only tag `c_compiler_name` is specified then that tool will be used only for C files, and C++ files will be compiled by the standard compiler.

An example:

```
export MXX_RU_CPP_TOOLSET="gcc_linux compiler_name=my-gcc linker_name=my-gcc"
```

bcc_win32_5	Borland C++ 5.* for Microsoft Windows
c89_etk_nsk	c89 for HP NonStop from eToolkit for Microsoft Windows
c89_nsk	89 for HP NonStop
icc_win	Intel C++ for Windows
gcc_cygwin	GNU C/C++ for Cygwin
gcc_darwin	GNU C/C++ for Mac OS X
gcc_linux	GNU C/C++ for Linux
gcc_mingw	Minimalist GNU C/C++ for Windows
gcc_sparc_solaris	GNU C/C++ for Solaris on SPARC
vc7	Visual C++ 7.* for Microsoft Windows
vc8	Visual C++ 8.* for Microsoft Windows
vc9	Visual C++ 9.* for Microsoft Windows
vc10	Visual C++ 10.* for Microsoft Windows
vc11	Visual C++ 11.* for Microsoft Windows
vc12	Visual C++ 12.* for Microsoft Windows
vc14	Visual C++ 14.* for Microsoft Windows

Table 6.5: Supported toolsets for Mxx_ru.

compiler_name	C and C++ files compiler name.
c_compiler_name	C files compiler name.
cpp_compiler_name	C++ files compiler name.
linker_name	Linker name.
librarian_name	Librarian name.
import_librarian_name	Name of librarian for import library creation (for example, it is actial for Borland C++).
rc_name	Resource compiler name.
mt_name	Manifest tool name (for Visual C++ 8/9/10).

Table 6.6: The tags for specifying custom tool names.

Chapter 7

Additional features

7.1 mxx-show-cmd argument

By default, Mxx_ru doesn't show any information about actions it performs — it works like a black box. But sometimes it may be useful to know what commands are running. They would appear in stdout if `--mxx-show-cmd` argument is passed to Rube interpreter. For example, a build of a sample from 4.4 on page 21 using a GNU C++ compiler in Cygwin environment, `--mxx-show-cmd` argument would bring this output:

```
ruby build.rb --mxx-show-cmd
<<< g++ -c -o say/o/say.o -I. say/say.cpp >>>
<<< ar -r  lib/libsay.a say/o/say.o >>>
<<< g++ -c -o inout/o/inout.o -DINOUT_PRJ -I. inout/inout.cpp >>>
<<< g++ -shared --shared-libgcc -o ./libinout.so -Llib inout/o/inout.o \
-lstdc++ -lsay -Wl,--out-implib=lib/libinout.a,--export-all-symbols >>>
<<< g++ -c -o main/o/main.o -I. main/main.cpp >>>
<<< g++ --shared-libgcc -o ./exe_dll_lib.exe -Llib main/o/main.o -lstdc++ \
-linout  >>>
```

7.2 mxx-keep-tmps argument

To create command strings for some tools on some platforms, creation of *response*-files is required. It's a temporary text files with values of parameters inside. In that case, Mxx_ru creates temporary files with unique names, executes a tools required and removes that files after processing of project file is done. For example, a build of a sample from 4.4 on page 21 using a Visual C++ compiler would bring this output:

```
ruby build.rb --mxx-show-cmd
<<< cl @tmpmxx_ru.3804.1 >>>
say.cpp
<<< lib @tmpmxx_ru.3804.2 >>>
<<< cl @tmpmxx_ru.3804.3 >>>
inout.cpp
<<< link @tmpmxx_ru.3804.4 >>>
   Creating library lib/inout.lib and object lib/inout.exp
<<< cl @tmpmxx_ru.3804.5 >>>
```

```
main.cpp
<<< link @tmpmxx_ru.3804.6 >>>
```

Where files with names like `tmpmxx_ru.[0-9]+.[0-9]+` — are temporary files, created by Mxx_ru.

Sometimes, during a debug of projects or new toolsets is useful to save temporary files for following analysis. It's possible to do using a `--mxx-keep-tmps` argument passed into Ruby interpreter:

```
ruby build.rb --mxx-keep-tmps
```

7.3 mxx-show-tmps argument

`--mxx-show-tmps` argument is analogue to `--mxx-show-cmd` argument, but for temporary response-files. Mxx_ru puts contents of temporary files to stdout if `--mxx-show-tmps` is passed to Ruby interpreter. For example, a build of a sample from 4.4 on page 21 using a Visual C++ compiler with `--mxx-show-cmd` and `--mxx-show-tmps` arguments would bring this output:

```
<<<[tmpmxx_ru.896.1]     -c -TP -Fosay/o/say.obj -nologo -ML -I. -GX \
say/say.cpp>>>
<<< cl @tmpmxx_ru.896.1 >>>
say.cpp
<<<[tmpmxx_ru.896.2]     /NOLOGO /OUT:lib/say.lib say/o/say.obj>>>
<<< lib @tmpmxx_ru.896.2 >>>
<<<[tmpmxx_ru.896.3]     -c -TP -Foinout/o/inout.obj -nologo -ML -LD \
-DINOUT_PRJ -DINOUT_MSWIN -I. -GX inout/inout.cpp>>>
<<< cl @tmpmxx_ru.896.3 >>>
inout.cpp
<<<[tmpmxx_ru.896.4]     /DLL /NOLOGO /SUBSYSTEM:CONSOLE /OUT:./inout.dll \
/IMPLIB:lib/inout.lib /LIBPATH:lib inout/o/inout.obj say.lib  >>>
<<< link @tmpmxx_ru.896.4 >>>
   Creating library lib/inout.lib and object lib/inout.exp
<<<[tmpmxx_ru.896.5]     -c -TP -Fomain/o/main.obj -nologo -ML -I. -GX \
main/main.cpp>>>
<<< cl @tmpmxx_ru.896.5 >>>
main.cpp
<<<[tmpmxx_ru.896.6]     /NOLOGO /SUBSYSTEM:CONSOLE /OUT:./exe_dll_lib.exe \
/LIBPATH:lib main/o/main.obj inout.lib  >>>
<<< link @tmpmxx_ru.896.6 >>>
```

7.4 mxx-dry-run argument

If `--mxx-dry-run` argument is passed to Ruby interpreter, Mxx_ru would perform only imitation of project build. That means all projects are processed, all command strings are created, including response-files creation, but tools itself are not executed.

It's useful to use this argument during a project files debug in combination with `--mxx-show-cmd`, `--mxx-show-tmps` and `--mxx-keep-tmps` arguments.

7.5 Exceptions

Ruby is an object-oriented programming language, with exception mechanisms widely used in. For example, exception is thrown if Ruby interpreter sees a syntax error in a script. Mxx_ru itself also uses exceptions for informing developer about errors encountered.

One of Mxx_ru features is that in selected architecture for project files it's impossible to handle all types of exceptions by Mxx_ru. As a result, for example, because of syntax error in a project file, interpreter may print that stack-trace:

```
mxx_ru/abstract_target.rb:187:in ‘require’: ./say/prj.rb:9: \
syntax error (SyntaxError)
    from mxx_ru/abstract_target.rb:187:in ‘required_prj’
    from ./inout/prj.rb:8
    from ./inout/prj.rb:4:in ‘instance_eval’
    from mxx_ru/cpp/target.rb:1256:in ‘instance_eval’
    from mxx_ru/cpp/target.rb:1256:in ‘initialize’
    from ./inout/prj.rb:4:in ‘new’
    from ./inout/prj.rb:4
    from mxx_ru/abstract_target.rb:187:in ‘require’
     ... 11 levels...
    from mxx_ru/cpp/composite.rb:21:in ‘instance_eval’
    from mxx_ru/cpp/composite.rb:21:in ‘initialize’
    from build.rb:4:in ‘new’
    from build.rb:4
```

This amount of details may shock for a first time. But with a time this problem goes away, firstly because amount of syntax errors is reduced with experience, and, as a second, it allows to diagnose problems inside Mxx_ru itself.

But if Mxx_ru can catch an exception, then it does it and shows only description of exception:

```
<<< cl @tmpmxx_ru.2324.1 >>>
say.cpp
say\say.cpp(14) : fatal error C1075: end of file found before the left \
brace ’{’ at ’say\say.cpp(11)’ was matched
<<<[Mxx_ru::Build_ex]   Build error: ’cl @tmpmxx_ru.2324.1’ returns ’512’>>>
```

7.6 Adding make-rules into a project

The basic idea of Mxx_ru consists in simplifying the most of typical actions required to be performed during a project build. Because of that, Mxx_ru provides a set of templates, used to fill data in. But sometimes capability provided by templates are not enough. For example, in current Mxx_ru version no support of localization in Qt-applications. If Qt project would require, for instance, to generate .qm file from ready .ts file, it would be impossible to do using current Mxx_ru templates only. It would be a good solution to build a feature required into Mxx_ru for using it in a future Qt-projects, but it's clear that rare developer, using Mxx_ru to build his own projects, would want to develop Mxx_ru itself.

In that cases it's only one method remains — to use `MxxRu::MakestyleGenerator` class, allowing to include an arbitrary make-rule into the project:

```
...
require 'mxx_ru/makestyle_generator'
...
generator(
    MxxRu::MakestyleGenerator.new(
        "etc/wms_ctl_1.ru.qm", "etc/wms_ctl_1.ru.ts",
        "lrelease etc/wms_ctl_1.ru.ts -qm etc/wms_ctl_1.ru.qm" ) )
```

In example above, make-rule is created to generate an `etc/wms_ctl_1.ru.qm` file (target of make-rule) from `etc/wms_ctl_1.ru.ts` file (dependency of target of make-rule) using a `lrelease` tool from Qt toolkit. When build of project is run, `MxxRu::MakestyleGenerator` is checking .qm file for existence. If it's not exists or it's "modified" attribute is earlier then .ts file, `lrelease` tool is executed. In the other words, the logic of usual make-rule is performed. During a clean operation, `etc/wms_ctl_1.ru.qm` file is removed.

But, unlike usual make-files, rules, created using a `MxxRu::MakestyleGenerator` are executed before all other actions in project file. And, in addition to, they are run in exact order they are written in a project file. Because of that, this example:

```
generator(
    MxxRu::MakestyleGenerator.new( "c", "b", "make_c" ) )
generator(
    MxxRu::MakestyleGenerator.new( "b", "a", "make_b" ) )
```

would not work if `b` file is not exists, since first generator doesn't know anything about any other generators in a project. And that's an exact behavior was planned during a `MxxRu::MakestyleGenerator` creation — Mxx_ru is not a usual make, and all complex operations should be performed using a Mxx_ru tools created for them. And `MxxRu::MakestyleGenerator` is the only guarantee that if no such tools exists at a moment, project still can be built.

A constructor of a `MxxRu::MakestyleGenerator` class expecting 4 arguments (the last one is optional):

a_target_files names of target files for make-rule.

a_dependencies names of dependencies for make-rule.

a_build_cmds A list of commands for building target files.

a_clean_cmds A list of commands for cleaning up. By default, target files would be deleted.

If only one value would be passed into a constructor argument, it may be passed using a string (as in example above). If several values are required to be passed, they should be defined as a vector:

```
generator(
    MxxRu::MakestyleGenerator.new(
        # Target file names.
        [ "a", "b", "c" ],
        # One dependency for all targets.
```

```
6           "d",
7           # Build of a targets commands.
8           [
9               "make_a",
10              "make_b_c"
11          ],
12          # Cleanup commands.
13          [
14              "destroy_a",
15              "destroy_b",
16              "destroy_c"
17          ]
18      ) )
```

During a build of target files `MxxRu::MakestyleGenerator` runs commands from `a_build_cmds` argument in specified order. If some command returns exit code other then zero, exception is thrown. During a cleanup process, exit codes of commands from `a_clean_cmds` argument are ignored.

Chapter 8

Unit-testing support

8.1 Unit-testing with binary applications

8.1.1 Definition of unit-test application

Unit-test application is a binary application containing unit-test code responsible to test some units. The success of unit-test is detected by exit code returned by unit-test application. If unit-test application returns zero, the tests are successful.

8.1.2 The basic idea

For unit-test support it's necessary to create two project files: First is responsible for building unit-test application, and second is responsible to run it and analyze it's result.

It's supposed that project have a composite project file, describing all subprojects. Unit-test project files should be also included into that list. In result it turns out that during a build of composite project, unit-test application is also built and executed. If unit-test is completed successfully, composite project build process would be continued. Otherwise build process would be interrupted.[1].

8.1.3 Unit-test target class

In Mxx_ru object of `MxxRu::BinaryUnittestTarget` class is required to include unit-test into a project build process. File `mxx_ru/binary_unittest` should be included in order to use that class:

```
1 require 'mxx_ru/binary_unittest'
2
3 MxxRu::setup_target(
4     MxxRu::BinaryUnittestTarget.new(
5         "some/project/prj.ut.rb",
6         "some/project/prj.rb" )
7 )
```

That class gets all it's parameters from it's constructor: it's own alias and project name, responsible to build unit-test application.

[1] That is the difference of unit-test vs. regression-test in Mxx_ru: It's required all unit-tests to be successful.

8.1.4 Example

Project file of unit-test application:

```
require 'mxx_ru/cpp'

MxxRu::Cpp::exe_target( "test/active_group/prj.rb" ) {

    required_prj( "threads_1/dll.rb" )
    required_prj( "so_4/prj.rb" )

    target_root( "unittest" )
    target( "test.active_group" )

    cpp_source( "main.cpp" )
}
```

Unit-test project file:

```
require 'mxx_ru/binary_unittest'

MxxRu::setup_target(
    MxxRu::BinaryUnittestTarget.new(
        "test/active_group/prj.ut.rb",
        "test/active_group/prj.rb" )
)
```

Composite project file:

```
require 'mxx_ru/cpp'

MxxRu::Cpp::composite_target( Mxx_ru::BUILD_ROOT ) {
    global_include_path( "." )

    required_prj( "so_4/prj.rb" )
    ...
    required_prj( "test/active_group/prj.ut.rb" )
    ...
}
```

Composite project may be run as usual:

```
ruby build.rb
```

But during a build process next message would appear:

```
running unit test: unittest/test.active_group.exe...
```

If not all unit-tests would be successful, result of project build may look like this:

```
main.cpp
running unit test: unittest/test.active_group.exe...
thread group #0: a_receiever_1 a_receiever_1::a_1 a_receiever_2::a_1
thread group #1: a_receiever_2
[2004.09.24 16:58:29.669152] so_4/ret_code.cpp:40:\
test/active_group/main.cpp:347: 10000 [Invalid threads groups]


unit test 'unittest/test.active_group.exe' FAILED! 256
<<<[Mxx_ru::Build_ex]   Build error: 'unittest/test.active_group.exe'\
 returns '256'>>>
```

Negative unit-test target class

Sometimes it is a lot easier to write unit test that alwais fails. For example this kind of test helps to check programs/libraries/classes which must call *std::abort()* in some cases.

Since v.1.6.3 Mxx_ru supports this kind of unit-tests by `MxxRu::NegativeBinaryUnittestTarget`:

```
1 require 'mxx_ru/binary_unittest'
2
3 MxxRu::setup_target(
4     MxxRu::NegativeBinaryUnittestTarget.new(
5         "some/project/prj.ut.rb",
6         "some/project/prj.rb" )
7 )
```

This class does exactly the same actions that `MxxRu::BinaryUnittestTarget`. The only difference is the checking of the exit code for unit-test application. `MxxRu::BinaryUnittestTarget` requires that exit code is 0 (exactly 0). But `MxxRu::NegativeBinaryUnittestTarget` requires that exit code is not 0.

8.2 Running unit-tests using comparison of text files

In some cases, it's too difficult to create a binary unit-test application, controlling test results and returning a zero exit code if successful (that applications is required for `MxxRu::BinaryUnittestTarget` class, described in 8.1 on page 72). For example, it's too difficult to test a text-parsing libraries that way. More conveniently to create a text application, getting one source file as a parameter, and saving result as another text file. Then, for each of source files it's *reference* file of a result. And the whole library testing would consists in execution of test application with prepared source file as an input and comparing it's result with a reference.

That testing scenario also may be applied in cases when testing unit's result depends from source data. It would be even better if simple procedure for adding/replacing source data, without recompiling of a unit-test application. For instance, for libraries performing calculations, encoding/decoding, creating/checking of cryptographic signature e.t.c. That

kind of testing is also applicable into a cases when success of test may be determined by comparison of log-files.

For support of testing that way, Mxx_ru gives `MxxRu::TextfileUnittestTarget` class. It may be used in a way similar to `MxxRu::BinaryUnittestTarget` [2]: it's necessary to create two project files — first is responsible for building unit-test application, and second is responsible to run it with different parameters and analyze it's result.

8.2.1 MxxRu::TextfileUnittestTarget class

`MxxRu::TextfileUnitestTarget` class is intended for defining a targets, running one unit-test application with different parameters and comparing results after each run with reference results. In order to use that class `mxx_ru/textfile_unittest` file should be included into a project:

```
require 'mxx_ru/textfile_unittest'

MxxRu::setup_target(
    MxxRu::TextfileUnittestTarget.new(
        "some/project/prj.ut.rb",
        "some/project/prj.rb" ) {

        # Description of tests sequence.
        ...
    }
)
```

Each execution of unit-test application is defined with a `launch` method. First argument of that method is string of parameters for unit-test application. Second argument is a vector of files to be compared. In other words, it's allowed for a test application to produce more then one file as a result.

In a `build` method of `MxxRu::TextfileUnittestTarget` class Mxx_ru executes build of test application at first. On success, a sequence of tests is run in their definition order. For each successful execution (zero exit code was returned), a consecutive comparison of result and reference files is performed. If all pairs of files are equal, next execution is run.

8.2.2 Example

Unit-test project file:

```
require 'mxx_ru/textfile_unittest'

MxxRu::setup_target(
    MxxRu::TextfileUnittestTarget.new(
        "prj.ut.rb",
        "prj.rb" ) {

        launch( "out_0.txt 0",
            [ pair( "out_0.txt", "etalons/out_0.txt" ) ] )
```

[2] See 8.1.2 on page 72

```
10
11          launch( "out_1.txt 1",
12              [ pair( "out_1.txt", "etalons/out_1.txt" ) ] )
13
14          launch( "out_128.txt 128",
15              [ pair( "out_0.txt", "etalons/out_0.txt" ),
16                  pair( "out_1.txt", "etalons/out_1.txt" ),
17                  pair( "out_128.txt", "etalons/out_128.txt" ) ] )
18      }
19  )
```

The result of unit-test run:

```
$ ruby prj.ut.rb
running unit test: ./test.exe...
    launching './test.exe out_0.txt 0'...
        comparing 'out_0.txt' and 'etalons/out_0.txt'
    launching './test.exe out_1.txt 1'...
        comparing 'out_1.txt' and 'etalons/out_1.txt'
    launching './test.exe out_128.txt 128'...
        comparing 'out_0.txt' and 'etalons/out_0.txt'
        comparing 'out_1.txt' and 'etalons/out_1.txt'
        comparing 'out_128.txt' and 'etalons/out_128.txt'
```

If during a comparison of files mismatches are detected, result may look like this:

```
$ ruby prj.ut.rb
running unit test: ./test.exe...
    launching './test.exe out_0.txt 0'...
        comparing 'out_0.txt' and 'etalons/out_0.txt'
    launching './test.exe out_1.txt 1'...
        comparing 'out_1.txt' and 'etalons/out_1.txt'
    launching './test.exe out_128.txt 128'...
        comparing 'out_0.txt' and 'etalons/out_0.txt'
        comparing 'out_1.txt' and 'etalons/out_1.txt'
        comparing 'out_128.txt' and 'etalons/out_128.txt'
<<<[Mxx_ru::Build_ex]   Build error: './test.exe out_128.txt 128' \
returns 'Error during comparing files 'out_128.txt', 'etalons/out_128.txt':\
 Build error: './test.exe out_128.txt 128' returns 'Mismatch found in \
line 120. Line in 'out_128.txt' is 'Hello, World!
'. Line in 'etalons/out_128.txt' is 'Hello, world
'''>>>
```

8.2.3 Features

`MxxRu::TextfileUnittestTarget` class performs a simple per string comparison of files in case sensitive manner. A comparison is interrupted on first mismatch.

In a `clean` method `MxxRu::TextfileUnittestTarget` class removes all files, defined as output (file names defined in a first argument of `pair` method). `clean` method is called when `--mxx-clean` argument was passed into Ruby interpreter.

Chapter 9

Qt4 generator

9.1 Introduction

This chapter describing a C++ files generator, required if Qt4 library `http://www.trolltech.com` is used. The described generator was tested on Qt version 4.4.3.

At use of Qt library there is a necessity of C++ files generation with Qt4 tools in the following cases:

1. When a class inherited from QObject is defined in a header file, and slots or signals are defined in that class. In that case source file should be created from header file, containing implementation of slots and signals. For generation, *moc* tool should be used. Usually `moc_a.cpp` file would be generated from `a.h` file. In other words, extension is changed to `.cpp` and `moc_` prefix is added to the name of file.

2. When a class inherited from QObject is defined in a source file. In that case, additional source file should be created and included into the main source file using `#include` directive:

```
1 ...
2 class   MyWidget : public QWidget { ... }
3
4 #include "mywidget.moc"
5 ...
```

   For generation, *moc* tool is used. Usually `a.moc` file would be generated from `a.cpp` file. In other words, extension is changed to `.moc`.

3. From `.ui` file, produced by Qt Designer, header file should be created. *uic* is used for generation. For example, from `a.ui` a file with name `ui_a.h` will be generated.

4. From `.qrc` file either a `.cpp` or a `.rcc` file should be created. *rcc* tool is used for generation. In the case of a `.cpp` file generation the result of generation will automatically be added to the list of project's source files.

In additional to the cases described above Qt4 generator allows selection of Qt4 modules for a projects. For example:

```
require 'mxx_ru/cpp/qt4'

MxxRu::Cpp::exe_target {
  ...
  qt = MxxRu::Cpp::Qt4.new( self )
  # The project needs Gui, Network and Script modules.
  qt.use_modules QT_GUI_LIB, QT_NETWORK_LIB, QT_SCRIPT_LIB
  ...
}
```

And Qt4 generator allows building of translation (`.ts`) files form source and `.ui` files. For example:

```
require 'mxx_ru/cpp/qt4'

MxxRu::Cpp::exe_target {
  ...
  qt = MxxRu::Cpp::Qt4.new( self )
  # Project should have translation to Russian.
  qt.ts 'translations/client_ru.ts'
  ...
}
```

9.2 Qt4 generator usage

9.2.1 Adding definitions required to a project file

Qt4 generator definition is in `mxx_ru/cpp/qt4.rb` file, and should be included into a project file using a `require` directive:

```
require 'mxx_ru/cpp/qt4'

MxxRu::setup_target( ... )
```

9.2.2 Adding definitions in presence of pkg-config

In the case when there is `pkg-config` tool in the system (for example on Debian GNU/Linux) it is possible to use pkg-config for detection of Qt4-related files locations and compiler/linker options.

To use pkg-config it is necessary to include `mxx_ru/cpp/qt4_via_pkg_config.rb` instead of `mxx_ru/cpp/qt4.rb`:

```
require 'mxx_ru/cpp'
require 'mxx_ru/cpp/qt4_via_pkg_config'

MxxRu::setup_project( ... )
```

The work scheme of `mxx_ru/cpp/qt4_via_pkg_config.rb` is:

- at first, the presence of pkg-config is checked by `pkg-config --version` command;
- if pkg-config is present in the system then call of Qt4.module leads to pkg-config invocations with arguments `--cflags` and `--libs` to detect all information required (include paths, library paths, compiler and linker options);
- if pkg-config isn't present then the work of `mxx_ru/cpp/qt4_via_pkg_config.rb` is the same as work of `mxx_ru/cpp/qt4.rb`.

For example suppose that we are working on Debian GNU/Linux and we need to use QtGui module. We specify in the project file:

```
require 'mxx_ru/cpp'
require 'mxx_ru/cpp/qt4_via_pkg_config'
...
  qt = generator MxxRu::Cpp::Qt4.new( self )
  qt.use_module QT_GUI
...
```

During the call to `qt.use_module QT_GUI` pkg-config is being invoked with argument `--cflags`:

```
pkg-config --cflags
```

The result of such invocation could be like the following:

```
-DQT_SHARED -I/usr/include/qt4/QtGui
```

Those options will be added as compiler options for the projects.

Then pkg-config is invoked with argument `--libs`:

```
pkg-config --libs
```

That could lead to:

```
-L/usr/lib/qt4/QtGui -lQtGui
```

And those options will be set as linker options for the projects.

On the systems without pkg-config (e.g. Windows) Qt-generator will work without pkg-config invocations. So it is safe to use `qt4_via_pkg_config` even on systems without pkg-config.

9.2.3 Creation of Qt4 generator

Qt4 generator is implemented in `MxxRu::Cpp::Qt4` class. To use it, object of that class should be created in a project:

```
require 'mxx_ru/cpp/qt4'

MxxRu::Cpp::exe_target {
  ...
  qt = generator( MxxRu::Cpp::Qt4.new( self ) )
  ...
}
```

The constructor of `Qt4` class have two arguments: a pointer to target-object, where generator would be used, and a set of preprocessor symbols, required to compile a project. By default, second argument is equal to `['QT_DLL', 'QT_THREAD_SUPPORT']`. In other words, if Qt is used in DLL mode with multithreading support, second argument may be omitted. For all other cases second argument should contain all preprocessor symbols required for Qt version used. For example, for statically linked Qt, but with multithreading support:

```
require 'mxx_ru/cpp/qt4'

MxxRu::Cpp::exe_target {
  ...
  qt = generator( MxxRu::Cpp::Qt4.new( self, [ 'QT_THREAD_SUPPORT' ] ) )
  ...
}
```

Note. The support of those preprocessor symbols is implemented for compatibility with Qt3 support in earlier versions of Mxx_ru. Qt4 seems not need such defines.

9.2.4 Header files definition to generate signal/slot implementation sources

Header files, used to generate source files with *moc* tool should be defined using `h2moc` method:

```
require 'mxx_ru/cpp/qt4'

MxxRu::Cpp::exe_target {
  ...
  qt = generator( MxxRu::Cpp::Qt4.new( self ) )
  ...
  qt.h2moc( "mywidget.hpp" )
}
```

The names defined are searched from current value of sources_root (see 6.11.1 on page 51). In example above `mywidget.hpp` file should be in `some/project` folder. And results of *moc* tool would also be placed in that folder[1].

From files defined with `h2moc` method, source files are built, with added `moc_` prefix and extension changed to `.cpp`[2]. So from `some/project/mywidget.hpp` file, `some/project/moc_mywidget.cpp` file would be generated.

9.2.5 Source files definition to generate signal/slot implementation sources

Source files (`.cpp`), used to generate signal/slot implementation source files (`.moc`) with *moc* tool should be defined using `cpp2moc` method:

[1] Taking value of moc_result_subdir into account (see 9.2.9 on page 83)

[2] If it's not changed with `cpp_ext` attribute (see 9.2.12 on page 84)

```
require 'mxx_ru/cpp/qt4'

MxxRu::Cpp::exe_target {
  ...
  qt = generator( MxxRu::Cpp::Qt4.new( self ) )
  ...
  qt.cpp2moc( "mywidget.cpp" )
}
```

The names defined are searched from current value of sources_root (see 6.11.1 on page 51). In example above `mywidget.hpp` file should be in `some/project` folder. And results of *moc* tool would also be placed in that folder[3].

From files defined with `cpp2moc` method, source files are built, with extension changed to `.cpp`[4]. So from `some/project/mywidget.cpp` file, `some/project/mywidget.moc` file would be generated.

9.2.6 UI files definition

Qt Designer user interface files, saved as `.ui`-files, are defined using a `ui` method:

```
require 'mxx_ru/cpp/qt4'

MxxRu::Cpp::exe_target {
  ...
  qt = generator( MxxRu::Cpp::Qt4.new( self ) )
  ...
  qt.ui( "myform.ui" )
}
```

The names defined are searched from current value of sources_root (see 6.11.1 on page 51). In example above `myform.ui` file should be in `some/project` folder. And results of *ui* tool would also be placed in that folder.

From files defined with `ui` method, header files are built, with prefix `ui_` and extension changed to `.h`[5]. As result, from `some/project/myform.ui` would be generated file `some/project/ui_myform.h`[6].

9.2.7 Resource files definition

Qt4 uses resource files with `.qrc` extension. Such files should be translated into `.cpp`-files (textual form) or into `.rcc`-files (binary form). If textual form is used than the resulting `.cpp`-files should be added to the list of project's source files. The `rcc` tool is used for resource files processing. Mxx_ru supports both forms of resources.

For textual form `qrc2cpp` method should be used:

[3] Taking value of moc_result_subdir into account (see 9.2.9 on page 83)
[4] If it's not changed with `cpp_ext` attribute (see 9.2.12 on page 84)
[5] If it's not changed with `hpp_ext` attribute (see 9.2.13 on page 84)
[6] Taking value of uic_result_subdir into account (see 9.2.10 on page 83)

```
...
qt = generator( MxxRu::Cpp::Qt4.new( self ) )

sources_root( 'sample' ) {
  qt.qrc2cpp 'basic.qrc'
}
...
```

In that case `sample/basic.cpp` file will be generated and will be added to the list of project's source files. The extension depends from `cpp_ext` attribute (9.2.12 on page 84).

For binary form `qrc2rcc` method should be used:

```
MxxRu::Cpp::exe_target {
  target 'bin32/sample'

  qt = generator( MxxRu::Cpp::Qt4.new( self ) )

  sources_root( 'plugins' ) {
    qt.qrc2rcc 'plugins.qrc'
  }
...
}
```

In that case `bin32/plugins.rcc` file will be generated. Please take into account that for binary resource form the result files is placed into the same directory with projects target.

9.2.8 TS files definition

Qt4 can make `.ts`-files via `lupdate` tool. Those files are defined using `ts` method:

```
MxxRu::Cpp::exe_target {
  ...
  qt = MxxRu::Cpp::Qt4.new( self )
  qt.ts 'translations/client_ru.ts'
  qt.ts 'translations/client_fr.ts'
  ...
}
```

During the build Qt4-generator will run `lupdate` for each `.ts`-file defined. For example, if there is `gui/client/prj.rb` with the above contents then there will be two invocation of `lupdate` in the form:

```
lupdate -slient gui/client -ts gui/client/translations/client_ru.ts
lupdate -slient gui/client -ts gui/client/translations/client_fr.ts
```

It means that `lupdate` is forced to scan the directory with project file (and subdirectories below it). It means also that name of `.ts`-file is relative to project file.

9.2.9 Output folder for moc tool

By default, work results of moc tool are located in the same folder where source data were. In some cases it's not so convenient. For example, during a moving to new Qt version, when all old generated files should be removed. It would be easier to do if all generated files would be in separate folder.

In a `Mxx_ru::Cpp::Qt4` class `moc_result_subdir` attribute exists, responsible for determining where to put moc results. If it is equal to `nil`(the default value), then all generated files are placed near the sources they are generated from. Otherwise, files would be placed into defined subfolder of source folder. For example:

```
1 qt = generator( MxxRu::Cpp::Qt4.new( self ) )
2 qt.moc_result_subdir = "moc"
3 qt.h2moc( "h/mywidget.hpp" )
```

From `h/mywidget.hpp` file `h/moc/moc_mywidget.cpp` file would be generated.

The value of `moc_result_subdir` only matters for files defined using a `h2moc` and `cpp2moc` methods.

9.2.10 Output folder for uic tool

As well as in previous case, during the run of uic, by default results are stored in the same folder with source .ui file. If it isn't desirable, then `uic_result_subdir` method allows to set subfolder of a folder with .ui file, where uic would store it's results. For example:

```
1 qt = generator( MxxRu::Cpp::Qt4.new( self ) )
2 qt.uic_result_subdir = "uic"
3 qt.ui( "src/dlg.ui" )
```

would generate `src/uic/ui_dlg.h`. And `src/uic` path would be automatically included to project's `include_path` (6.15.1 on page 60).

9.2.11 Output folder for rcc tool

By default, textual form resources are located in the same folder where source data were. In some cases it's not so convenient. In a `Mxx_ru::Cpp::Qt4` class `qrc_result_subdir` attribute exists, responsible for determining where to put `.qrc` to `.cpp` translation results. If it is equal to `nil` (the default value), then all generated files are placed near the sources they are generated from. Otherwise, files would be placed into defined subfolder of source folder. For example:

```
1 qt = generator( MxxRu::Cpp::Qt4.new( self ) )
2 qt.qrc_result_subdir = "res"
3 qt.qrc2cpp "plugins/images.qrc"
```

From `plugins/images.qrc` file `plugins/res/images.cpp` file would be generated.

The value of `qrc_result_subdir` only matters for files defined using `qrc2cpp` method.

9.2.12 Extension for source files generated

By default, during a source files generation with moc and ui tools, `.cpp` extension is used. It may be changed by changing a `cpp_ext` attribute value:

```
1 qt = generator( MxxRu::Cpp::Qt4.new( self ) )
2 qt.cpp_ext = ".CC";
3 qt.h2moc( "h/mywidget.H" )
```

From `h/mywidget.H` file `h/mywidget.CC` file would be generated.

9.2.13 Extension for header files generated

By default, during a header files generation with ui tool, `.h` extension is used. It may be changed by changing a `hpp_ext` attribute value:

```
1 qt = generator( MxxRu::Cpp::Qt4.new( self ) )
2 qt.hpp_ext = ".H";
3 qt.ui( "myform.ui" )
```

From `myform.ui` file `ui_myform.H` file would be generated.

9.2.14 Extension for moc files generated

By default, during a moc files generation with moc tool, `.moc` extension is used. It may be changed by changing a `moc_ext` attribute value:

```
1 qt = generator( MxxRu::Cpp::Qt4.new( self ) )
2 qt.moc_ext = ".cpp.moc";
3 qt.cpp2moc( "mywidget.cpp" )
```

From `mywidget.cpp` file `mywidget.cpp.moc` file would be generated.

9.2.15 The file name of moc tool

The file name for moc tool is located in `moc_name` attribute. By default, it's equal to `moc`. Changing the value of that attribute allows to set another file name, used to run moc tool:

```
1 qt = generator( MxxRu::Cpp::Qt4.new( self ) )
2 qt.moc_name = "/usr/src/custom-qt.4.5/bin/moc"
```

9.2.16 The file name of uic tool

The file name for uic tool is located in `uic_name` attribute. By default, it's equal to `uic`. Changing the value of that attribute allows to set another file name, used to run uic tool:

```
1 qt = generator( MxxRu::Cpp::Qt4.new( self ) )
2 qt.uic_name = "/usr/src/custom-qt.4.5/bin/uic"
```

9.2.17 The file name of rcc tool

The file name for rcc tool is located in `rcc_name` attribute. By default, it's equal to `rcc`. Changing the value of that attribute allows to set another file name, used to run rcc tool:

```
1 qt = generator( MxxRu::Cpp::Qt4.new( self ) )
2 qt.rcc_name = "/usr/src/custom-qt.4.5/bin/rcc"
```

9.2.18 The file name of lupdate tool and custom lupdate options

The file name for lupdate tool is located in `lupdate_name` attribute. By default, it's equal to `lupdate`. Changing the value of that attribute allows to set another file name, used to run lupdate tool:

```
1 qt = generator( MxxRu::Cpp::Qt4.new( self ) )
2 qt.rcc_name = "/usr/src/custom-qt.4.5/bin/lupdate"
```

The options for lupdate tool are contained in `lupdate_options` attribute. By default they are `['-noobsolete', '-slient']`. It is possible to specify those options by modifying `lupdate_options` value:

```
1 qt = generator( MxxRu::Cpp::Qt4.new( self ) )
2 # Remove option -noobsolete and keep other options intact.
3 qt.lupdate_options = qt.lupdate_options.delete( '-noobsolete' )
4 ...
5 # Completely change options for lupdate.
6 qt.lupdate_options = [ '-pluralonly', '-extensions ui,hpp,cpp' ]
```

Chapter 10

Project file stub generators

10.1 Introduction

A simple idea is used as the base for project file stub generators: as long as too much similar fragments are need to be duplicated at the time of writing project files then it is possible to move this duty to a stub generator. A skeleton of a project file is created by a generator and missing information will be included into it.

For example, a generator had been run to create a project file for “Hello, World!”:

```
mxxrugen cpp-exe -t hello_world -o prj.rb
```

and stub file `prj.rb` was created as result:

```
require 'rubygems'

gem 'Mxx_ru', '>= 1.3.0'

require 'mxx_ru/cpp'

MxxRu::Cpp::exe_target {

  # Define your target name here.
  target 'hello_world'

  # Enumerate one or more required projects.
  #required_prj 'some project'

  # Enumerate your C/C++ files here.
  #c_source 'C source file'
  #cpp_source 'C++ source file'

}
```

that was modified later by excluding unnecessary details:

```
require 'rubygems'

gem 'Mxx_ru', '>= 1.3.0'
```

```
4
5  require 'mxx_ru/cpp'
6
7  MxxRu::Cpp::exe_target {
8
9    target 'hello_world'
10
11   cpp_source 'hello_world.cpp'
12
13 }
```

Since Mxx_ru is intended to support different kinds of project, the stub generation is built in such a way as to generate different kinds of stubs in the same way. For this purpose a generator utility, mxxrugen, and a set of templates for various project's types have been added into Mxx_ru. A name of template is specified to mxxrugen at start (*cpp-exe* in the example above) with an optional set of template's parameters. The mxxrugen utility searches template and uses it for the stub generation. Such schema allows extension of list of available templates as Mxx_ru will grow.

10.2 The mxxrugen generator

The mxxrugen utility is automaticaly installed into `bin` directory of Ruby interpreter during Mxx_ru RubyGem installation. After that mxxrugen can be launched from the command line like a Ruby interpreter itself.

Synopsis:

```
mxxrugen [<mxxrugen-options>] template-name [<template-options>]
```

The mxxrugen utility supports the following options:

- `--help` print short help for mxxrugen options;
- `-I, --include-path PATH` an additional path for searching templates. By default the mxxrugen searches in the path `<mxxru>/generators`, where `<mxxru>` — is a placement of Mxx_ru;
- `-l, --list` print list of templates found.

The option `--list` is intended for determination of list of available templates. For example:

```
> mxxrugen --list
cpp-composite (c:/ruby/lib/ruby/gems/1.8/gems/Mxx_ru-1.4.0/lib/mxx_ru/generators)
cpp-dll (c:/ruby/lib/ruby/gems/1.8/gems/Mxx_ru-1.4.0/lib/mxx_ru/generators)
cpp-exe (c:/ruby/lib/ruby/gems/1.8/gems/Mxx_ru-1.4.0/lib/mxx_ru/generators)
cpp-lib (c:/ruby/lib/ruby/gems/1.8/gems/Mxx_ru-1.4.0/lib/mxx_ru/generators)
cpp-lib-collection (c:/ruby/lib/ruby/gems/1.8/gems/Mxx_ru-1.4.0/lib/mxx_ru/generators)
```

shows that five templates are available: *cpp-composite*, *cpp-dll*, *cpp-exe*, *cpp-lib*, *cpp-lib-collection* and all them are a part of Mxx_ru version 1.4.0.

The option `--help` forces the mxxrugen to print only mxxrugen own otions. Templates can have their own lists of options. A concrete template must be launched with the `--help` option to get to know a list of this template specific options:

```
> mxxrugen cpp-lib-collection --help
Stubs for C/C++ projects generator
LibCollection target generator

Usage:
mxxrugen [<mxxrugen-options>] cpp-lib-collection [<options>]

    -t, --target NAME                Target name
    -o, --output-file FILE           Output file name
    -h, --help                       Show this message

> mxxrugen cpp-dll --help
Stubs for C/C++ projects generator
DLL target generator

Usage:
mxxrugen [<mxxrugen-options>] cpp-dll [<options>]

    -t, --target NAME                Target name
    -o, --output-file FILE           Output file name
        --implib-path PATH           Import library path name
    -h, --help                       Show this message
```

Notice. A support of the `--help` option is a feature of standard Mxx_ru templates. Non-standard templates which are included by the `--include-path` may not follow this convention.

10.3 Standard templates for C/C++ projects

10.3.1 cpp-composite, cpp-dll, cpp-exe, cpp-lib, cpp-lib-collection

Those templates is intended for generation of project files of the corresponding types (see table 10.1)

cpp-composite	MxxRu::Cpp::CompositeTarget
cpp-dll	MxxRu::Cpp::DllTarget
cpp-exe	MxxRu::Cpp::ExeTarget
cpp-lib	MxxRu::Cpp::LibTarget
cpp-lib-collection	MxxRu::Cpp::LibCollection

Table 10.1: The correspondence between the names of standard C/C++ templates and the types of C/C++ projects.

Those templates have single implementation and because of that support the following command line format:

```
<template-name> [<path-name>] [<template-options>]
```

where ¡*template-options*¿ are:

- `-t, --target NAME` optional option which specifies project's target name;
- `-o, --output-file FILE` optional option which specifies file's name where the generated stub must be saved. If that name is not specified then the generated stub is printed to the standard output;
- `--implib-path PATH` optional option which specifies path's name where the import library is resided. This option is used only for templates *cpp-dll* and *cpp-exe*;
- `-h, --help` shows short usage help.

If the option `--target` is specified at the generator start then its value is used as target's name:

```
mxxrugen cpp-exe -t hello.world
```

that produces:

```
...
MxxRu::Cpp::exe_target {

  # Define your target name here.
  target 'hello.world'
...
}
```

If the option `--target` is missed but path's name is specified (the `path-name` argument) then target's name is produced form path's name (all slashes are substituted by points):

```
mxxrugen cpp-exe some/my/hello/world
```

that produces:

```
...
MxxRu::Cpp::exe_target {

  # Define your target name here.
  target 'some.my.hello.world'
...
}
```

If neither target's name, nor path's name is specified at the generator start that the notice about necessarity of manual inclusion of those names is inserted into the generated stub. For example:

```
mxxrungen cpp-dll
```

generates:

```
...
MxxRu::Cpp::dll_target {

  # Define your target name here.
  target 'your target name'

  # Define your import library path here.
  implib_path 'your import library path'
...
}
```

10.3.2 cpp-build-root

This template is intended for generation of stubs for project build root files (which usually have name 'build.rb').

Synopsis:

```
cpp-build-root [<template-options>]
```

where *¡template-options¿* are:

- `-o, --output-file FILE` optional option which specifies file's name where the generated stub must be saved. If that name is not specified then the generated stub is printed to the standard output.

10.4 Standard templates for unit-tests

10.4.1 bin-unittest

This template is intended for generation of stubs for unit-test of `MxxRu::BinaryUnittestTarget` (8.1 on page 72) type.

Synopsis:

```
bin-unittest [<path-name>] [<template-options>]
```

where *¡template-options¿* are:

- `-o, --output-file FILE` optional option which specifies file's name where the generated stub must be saved. If that name is not specified then the generated stub is printed to the standard output.

If the *path-name* argument is specified at the generator start:

```
mxxrugen bin-unittest some/my/test
```

then that name is used in the body of the generated stub:

```
require 'rubygems'

gem 'Mxx_ru', '>= 1.3.0'

require 'mxx_ru/cpp'

# Set path to your unit test projet here.
path = 'some/my/test'

MxxRu::setup_target(
  MxxRu::BinaryUnittestTarget.new(
    # Set name of your unit test project here.
    "#{path}/prj.ut.rb",
    # Set name of your project to be tested here.
    "#{path}/prj.rb" ) )
```

If the path name is missed but the `--output-file` is specified:

```
mxxrugen bin-unittest -o hello/hello.test.rb
```

then the path to output file is used in the generation as if it had been specified as *¡path-name¿* argument:

```
require 'rubygems'

gem 'Mxx_ru', '>= 1.3.0'

require 'mxx_ru/cpp'

# Set path to your unit test projet here.
path = 'hello'

MxxRu::setup_target(
  MxxRu::BinaryUnittestTarget.new(
    # Set name of your unit test project here.
    "#{path}/hello.test.rb",
    # Set name of your project to be tested here.
    "#{path}/prj.rb" ) )
```

Also in the case of `--output-file` option the generated stub is saved into the file specified, and nothing is shown to the standard output.

10.5 The usage of the stub generator from text editors

10.5.1 VIM

The stub generator can be used from VIM by one of the following methods:

1. The stub generator is run as an external utility and their output is saved into a concrete file. After that this file is loaded and edited as usual:

```
:!mxxrugen cpp-exe -t my.project -o some/my/project/prj.rb
:e some/my/project/prj.rb
<...editing of the project file...>
:w
```

2. A new file is started and the generation result is inserted into it by the help from command `:read !{cmd}`. This method is based on the fact that the result of generation is shown on the standard output by default. The standard output is redirected into the current editor buffer by the editor:

```
:e some/my/project/prj.rb
<...new file is opened...>
:r!mxxrugen cpp-exe -t my.project
<...editing of the project file...>
:w
```

10.6 The creation of custom templates

A template is a Ruby programm which must conforms to the following conditions:

- resides in the subdirectory with the name of template (for example the template *cpp-dll* resides in subdirectory `cpp-dll`);
- have the name `g.rb`;

If a template requires some parameters from the user it is prefered to use command line arguments for that.

It is desirable that a template handles the `-h, --help` option.

To use a custom template it is necessary to specify the path to custom template's directory into the `--include-path` option:

```
mxxrugen -I ~/my-templates custom-lib
```

In such case mxxrugen checks the existance of the file `~/my-templates/custom-lib/g.rb` and, if that file exists, runs it. If the *custom-lib* doesn't exist in the path `~/my-templates` then mxxrugen continues the search of it between the standard templates.

Appendix A

Qt3 generator

A.1 Introduction

This chapter describing a C++ files generator, required if Qt3 library `http://www.trolltech.com` is used. The described generator was tested on Qt version 3.3.3.

At use of Qt library there is a necessity of C++ files generation with Qt tools in three cases:

1. When a class inherited from QObject is defined in a header file, and slots or signals are defined in that class. In that case source file should be created from header file, containing implementation of slots and signals. For generation, *moc* tool should be used. Usually `moc_a.cpp` file would be generated from `a.h` file. In other words, extension is changed to `.cpp` and `moc_` prefix is added to the name of file.

2. When a class inherited from QObject is defined in a source file. In that case, additional source file should be created and included into the main source file using `#include` directive:

```
1 ...
2 class   MyWidget : public QWidget { ... }
3
4 #include "mywidget.moc"
5 ...
```

   For generation, *moc* tool is used. Usually `a.moc` file would be generated from `a.cpp` file. In other words, extension is changed to `.moc`.

3. From `.ui` file, produced by Qt Designer, header and source files should be created. *uic* tool should be used for generation. And then, from generated header file, additional source file for slots and signals implementation should be created with *moc* tool. Usually, from `a.ui` file, `a.hpp`, `a.cpp` and `moc_a.cpp` files are created.

Qt generator from Mxx_ru allows to work with all three cases described above.

A.2 Qt generator usage

A.2.1 Adding definitions required to a project file

Qt generator definition is in a `mxx_ru/cpp/qt.rb` file, and should be included into a project file using a `require` directive:

```
require 'mxx_ru/cpp/qt'

MxxRu::setup_target( ... )
```

A.2.2 Creation of Qt generator

Qt generator is implemented in `MxxRu::Cpp::QtGen` class. To use it, object of that class should be created in a project:

```
require 'mxx_ru/cpp/qt'

MxxRu::Cpp::exe_target {
  ...
  qt = generator( MxxRu::Cpp::QtGen.new( self ) )
  ...
}
```

The constructor of `QtGen` class have two arguments: a pointer to target-object, where generator would be used, and a set of preprocessor symbols, required to compile a project. By default, second argument is equal to `['QT_DLL', 'QT_THREAD_SUPPORT']`. In other words, if Qt is used in DLL mode with multithreading support, second argument may be omitted. For all other cases second argument should contain all preprocessor symbols required for Qt version used. For example, for statically linked Qt, but with multithreading support:

```
require 'mxx_ru/cpp/qt'

MxxRu::Cpp::exe_target {
  ...
  qt = generator( MxxRu::Cpp::QtGen.new( self, [ 'QT_THREAD_SUPPORT' ] ) )
  ...
}
```

A.2.3 Header files definition to generate signal/slot implementation sources

Header files, used to generate source files with *moc* tool should be defined using `h2moc` method:

```
require 'mxx_ru/cpp/qt'

MxxRu::Cpp::exe_target {
```

```
4     ...
5     qt = generator( MxxRu::Cpp::QtGen.new( self ) )
6     ...
7     qt.h2moc( "mywidget.hpp" )
8   }
```

The names defined are searched from current value of sources_root (see 6.11.1 on page 51). In example above `mywidget.hpp` file should be in `some/project` folder. And results of *moc* tool would also be placed in that folder[1].

From files defined with `h2moc` method, source files are built, with added `moc_` prefix and extension changed to `.cpp`[2]. So from `some/project/mywidget.hpp` file, `some/project/moc_mywidget.cpp` file would be generated.

A.2.4 Source files definition to generate signal/slot implementation sources

Source files (`.cpp`), used to generate signal/slot implementation source files (`.moc`) with *moc* tool should be defined using `cpp2moc` method:

```
1   require 'mxx_ru/cpp/qt'
2
3   MxxRu::Cpp::exe_target {
4     ...
5     qt = generator( MxxRu::Cpp::QtGen.new( self ) )
6     ...
7     qt.cpp2moc( "mywidget.cpp" )
8   }
```

The names defined are searched from current value of sources_root (see 6.11.1 on page 51). In example above `mywidget.hpp` file should be in `some/project` folder. And results of *moc* tool would also be placed in that folder[3].

From files defined with `cpp2moc` method, source files are built, with extension changed to `.cpp`[4]. So from `some/project/mywidget.cpp` file, `some/project/mywidget.moc` file would be generated.

A.2.5 UI files definition

Qt Designer user interface files, saved as `.ui`-files, are defined using a `ui` method:

```
1   require 'mxx_ru/cpp/qt'
2
3   MxxRu::Cpp::exe_target {
4     ...
5     qt = generator( MxxRu::Cpp::QtGen.new( self ) )
6     ...
```

[1] Taking value of moc_result_subdir into account (see A.2.6 on page 96)
[2] if it's not changed with `cpp_ext` attribute (see A.2.8 on page 97)
[3] Taking value of moc_result_subdir into account (see A.2.6 on page 96)
[4] if it's not changed with `cpp_ext` attribute (see A.2.8 on page 97)

```
7   qt.ui( "myform.ui" )
8 }
```

The names defined are searched from current value of sources_root (see 6.11.1 on page 51). In example above `myform.ui` file should be in `some/project` folder. And results of *ui* tool would also be placed in that folder.

From files defined with `ui` method, header files are built, with extension changed to `.hpp`[5], and source files are built also, with extension changed to `.cpp`[6]. The names of generated header files are passed to `h2moc` method for generating implementation for signal/slot mechanisms. As a result, from `some/project/myform.ui` file, these files would be generated: `some/project/myform.hpp`, `some/project/myform.cpp`, `some/project/moc_myform.cpp`[7].

A.2.6 Output folder for moc tool

By default, work results of moc tool are located in the same folder where source data were. In some cases it's not so convenient. For example, during a moving to new Qt version, when all old generated files should be removed. It would be easier to do if all generated files would be in separate folder.

In a `Mxx_ru::Cpp::QtGen` class `moc_result_subdir` attribute exists, responsible for determining where to put moc results. If it is equal to `nil`(the default value), then all generated files are placed near the sources they are generated from. Otherwise, files would be placed into defined subfolder of source folder. For example:

```
1 qt = generator( MxxRu::Cpp::QtGen.new( self ) )
2 qt.moc_result_subdir = "moc"
3 qt.h2moc( "h/mywidget.hpp" )
```

From `h/mywidget.hpp` file `h/moc/moc_mywidget.cpp` file would be generated.

The value of `moc_result_subdir` only matters for files defined using a `h2moc` and `cpp2moc` methods.

A.2.7 Output folder for uic tool

As well as in previous case, during the run of uic, by default results are stored in the same folder with source .ui file. If it isn't desirable, then `uic_result_subdir` method allows to set subfolder of a folder with .ui file, where uic would store it's results. For example:

```
1 qt = generator( MxxRu::Cpp::QtGen.new( self ) )
2 qt.uic_result_subdir = "uic"
3 qt.ui( "src/dlg.ui" )
```

would generate `src/uic/dlg.hpp`, `src/uic/dlg.cpp` and `src/uic/moc_dlg.cpp`. And `src/uic` path would be automatically included to project's `include_path`(6.15.1 on page 60).

[5] if it's not changed with `hpp_ext` attribute (see A.2.9 on page 97)
[6] if it's not changed with `cpp_ext` attribute (see A.2.8 on page 97)
[7] Taking value of uic_result_subdir into account (see A.2.7 on page 96)

A.2.8 Extension for source files generated

By default, during a source files generation with moc and ui tools, `.cpp` extension is used. It may be changed by changing a `cpp_ext` attribute value:

```
1 qt = generator( MxxRu::Cpp::QtGen.new( self ) )
2 qt.cpp_ext = ".CC";
3 qt.h2moc( "h/mywidget.H" )
```

From `h/mywidget.H` file `h/mywidget.CC` file would be generated.

A.2.9 Extension for header files generated

By default, during a header files generation with ui tool, `.hpp` extension is used. It may be changed by changing a `hpp_ext` attribute value:

```
1 qt = generator( MxxRu::Cpp::QtGen.new( self ) )
2 qt.hpp_ext = ".h";
3 qt.ui( "myform.ui" )
```

From `myform.ui` file `myform.h` file would be generated.

A.2.10 Extension for moc files generated

By default, during a moc files generation with moc tool, `.moc` extension is used. It may be changed by changing a `moc_ext` attribute value:

```
1 qt = generator( MxxRu::Cpp::QtGen.new( self ) )
2 qt.moc_ext = ".cpp.moc";
3 qt.cpp2moc( "mywidget.cpp" )
```

From `mywidget.cpp` file `mywidget.cpp.moc` file would be generated.

A.2.11 The file name of moc tool

The file name for moc tool is located in `moc_name` attribute. By default, it's equal to `moc`. Changing the value of that attribute allows to set another file name, used to run moc tool:

```
1 qt = generator( MxxRu::Cpp::QtGen.new( self ) )
2 qt.moc_name = "/usr/src/qt.4.0.b2/bin/moc"
```

A.2.12 The file name of uic tool

The file name for uic tool is located in `uic_name` attribute. By default, it's equal to `uic`. Changing the value of that attribute allows to set another file name, used to run uic tool:

```
1 qt = generator( MxxRu::Cpp::QtGen.new( self ) )
2 qt.uic_name = "/usr/src/qt.4.0.b2/bin/uic"
```