INTERVALE

125083, Москва, ул. 8-го Марта, 10/12
Тел./Факс: (095) 212-8238, 212-9975
http://www.intervale.ru

Mxx_ru

Краткое руководство

Е. Охотников

Оглавление

Глава 1

Введение

1.1 Что такое Mxx_ru

Mxx_ru — это написанный на языке Ruby[1] инструмент для поддержки кроссплатформенной компиляции и сборки проектов, в первую очередь, C/C++ проектов. В чем-то Mxx_ru является аналогом make, но вместо make-правил использует механизм шаблонов. Проектный файл в Mxx_ru представляет из себя небольшую программу на языке Ruby в которой используется уже готовый класс (шаблон) из состава Mxx_ru. Все, что нужно сделать программисту – это создать экземпляр необходимого класса и вызвать у него соответствующие методы. Все остальное делает Mxx_ru.

1.2 Возможности

Mxx_ru позволяет работать с проектами на разных языках программирования. При этом предоставляемая Mxx_ru функциональность зависит от языка программирования.

Для языков C/C++ Mxx_runпредоставляет следующие возможности:

- использование одного и того же проектного файла для разных компиляторов и на разных платформах;
- автоматическое отслеживание C/C++ зависимостей с помощью встроенного анализатора исходных текстов;
- готовые шаблоны для построения исполнимых (.exe)-файлов, динамических (.dll, .so) и статических (.lib, .a) библиотек;
- готовый шаблон для композитных (состоящих из нескольких самостоятельных подпроектов) проектов;
- поддержку компиляции ресурсных файлов на платформе Microsoft Windows;
- поддержку понятия “генератора исходных текстов” и ряд готовых генераторов, например, для Qt.

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

1.3 Немного истории

Mxx_ru является развитием инструмента Make++, разработанного Евгением Охотниковым[2]. Первоначально Make++ был создан на основе wmake из состава Watcom C++. Затем Make++ был полностью переписан на C++ и задачей Make++ было формирование make-файлов для конкретных компиляторов на основании простого текстового описания проекта. Этот вариант оказался плохо адаптируемым под новые инструменты и платформы. Поэтому был создан следующий вариант Make++, который был одновременно и интерпретатором собственного языка, и инструментом make. Но и этот вариант имел проблемы с новыми инструментами, особенно когда требовалось не только управлять компиляцией, но и генерировать часть исходных текстов (например, moc-файлы при использовании Qt[3]). В результате был создан следующий, четвертый вариант Make++, который был, главным образом, интерпретатором собственного языка программирования. А функциональность make-модуля была доступна в этом языке в виде API функций.

Четвертый вариант Make++ широко использовался компанией Intervale. Но со временем он так же достиг предела своих возможностей. В частности, переход на новые платформы и компиляторы становился все сложнее и сложнее, т.к. язык четвертой Make++ предоставлял очень мало алгоритмических возможностей. После оценки трудозатрат на различные варианты преодоления проблем четвертой версии Make++ компания Intervale приняла решение создать очередную, пятую версию Make++.

Основная идея пятой версии осталось такой же, как и в четвертой версии: проектный файл — это небольшая программа на скриптовом, интерпритируемом языке. Функциональность make-модуля и других вспомогательных модулей (например, анализатора C/C++ зависимостей) доступна в виде API функций и классов. Только в качестве языка программирования выбран скриптовый язык Ruby.

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

Глава 2

Инсталляция

2.1 Требования

Для использования Mxx_ru необходим Ruby версии 1.6 или выше. Загрузить Ruby для различных платформ можно с `http://www.ruby-lang.org`. Простой инсталлятор One-Click Ruby Installer для Microsoft Windows доступен на `http://rubyforge.org/projects/rubyinstaller`.

2.2 Получение Mxx_ru

Загрузить Mxx_ru можно либо с сайта компании Intervale (`http://www.intervale.ru`), либо с сайта Евгения Охотникова (`http://eao197.narod.ru`).

2.3 Инсталляция Mxx_ru

После того, как архив с Mxx_ru загружен, необходимо проинсталлировать Mxx_ru одним из следующих способов:

- распаковать содержимое архива Mxx_ru в каталог с библиотеками Ruby. На платформе Microsoft Windows это, обычно, `c:/ruby/lib/ruby/site-ruby`. На платформе Unix: `/usr/local/lib/ruby/site-ruby`. После этого Mxx_ru будет автоматически доступен Ruby;
- распаковать содержимое архива Mxx_ru в какой-нибудь удобный для разработчика каталог и указать имя этого каталога в переменной среды RUBYLIB. Например, на платформе Microsoft Windows:

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

  Или на платформе Unix:

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

После этого необходимо настроить Mxx_ru для того языка и тех инструментов, которые используются для разработки.

2.3.1 Настройка Mxx_ru для работы с C/C++ проектами

Для работы с C/C++ проектами необходимо для Mxx_ru установить переменную среды MXX_RU_CPP_TOOLSET. Значение этой переменной среды должно иметь вид:

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

где `<file>` – это имя .rb-файла из состава Mxx_ru, который отвечает за создания объекта C/C++-toolset-a. Например:

mxx_ru/cpp/toolsets/bcc_win32_5 компилятор Borland C++ 5.* на платформе Microsoft Windows;

mxx_ru/cpp/toolsets/vc7 компилятор Visual C++ 7.* на платформе Microsoft Windows;

mxx_ru/cpp/toolsets/gcc_unix компилятор GNU C++ на платформе Unix (включая FreeBSD и Linux);

mxx_ru/cpp/toolsets/gcc_sparc_solaris компилятор GNU C++ на платформе SPARC Solaris;

mxx_ru/cpp/toolsets/gcc_cygwin компилятор GNU C++ на платформе Cygwin в Microsoft Windows.

mxx_ru/cpp/toolsets/c89_nsk компилятор c89 на платформе HP NonStop в среде Open System Services.

Пары значений tag, value будут установлены в качестве тегов выбранного C/C++ toolset-a.

Примеры:

```
export MXX_RU_CPP_TOOLSET=mxx_ru/cpp/toolsets/gcc_unix unix=linux arch=x86
```

```
set MXX_RU_CPP_TOOLSET=mxx_ru/cpp/toolsets/vc7
```

```
set MXX_RU_CPP_TOOLSET=mxx_ru/cpp/toolsets/bcc_win32_5
```

Глава 3

Несколько простых примеров

3.1 Простой exe-файл

Пусть необходимо скомпилировать один C++ файл и получить exe-файл:

```
1  #include <iostream>
2  #include <string>
3
4  void
5  say_hello( std::ostream & to )
6    {
7      to << "Hello!" << std::endl;
8    }
9
10 void
11 say_bye( std::ostream & to )
12   {
13     to << "Bye!" << std::endl;
14   }
15
16 int
17 main()
18   {
19     say_hello( std::cout );
20
21     std::cout << "Simple exe main..." << std::endl;
22
23     say_bye( std::cout );
24
25     return 0;
26   }
```

Для этого потребуется следующий Mxx_ru-проектный файл (с именем prj.rb):

```
1  require 'mxx_ru/cpp'
2
3  Mxx_ru::setup_target(
4    Mxx_ru::Cpp::Exe_target.new( "prj.rb" ) {
```

```
5       target( "simple_exe" )
6
7       cpp_source( "main.cpp" )
8     }
9   )
```

Для компиляции проекта необходимо запустить интерпретатор ruby.rb и указать ему имя проектного файла:

```
ruby prj.rb
```

В результате, на платформе mswin будет построен файл simple_exe.exe, а на платформе unix — simple_exe.

3.2 exe-файл и простая статическая библиотека

Пусть теперь необходимо разбить приведенный выше пример на статическую библиотеку, предоставляющую функции say_hello(), say_bye(), и основной модуль, который будет пользоваться услугами этой библиотеки.

Для статической библиотеки создаются файлы:

- заголовочный say.hpp

```
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
```

- файл реализации say.cpp

```
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   }
```

- проектный файл say.rb

```
require 'mxx_ru/cpp'

Mxx_ru::setup_target(
  Mxx_ru::Cpp::Lib_target.new( "say.rb" ) {
    target( "say" )

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

Для основного модуля создаются файлы:

- файл реализации main.cpp

```
#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;
  }
```

- проектный файл prj.rb

```
require 'mxx_ru/cpp'

Mxx_ru::setup_target(
  Mxx_ru::Cpp::Exe_target.new( "prj.rb" ) {
    target( "exe_and_lib" )

    required_prj( "say.rb" )

    include_path( "." )

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

Для компиляции всего проекта, включая статическую библиотеку и основной модуль необходимо указать интерпретатору Ruby имя проектного файла prj.rb. Mxx_ru автоматически скомпилирует библиотеку say и слинкует с ней основной модуль.

Необходимо отметить, что в проектном файле prj.rb не делается никаких указаний относительно имени библиотеки say. Более того, основной модуль даже не знает, является ли библиотека статической или динамической. Основной модуль просто указывает, что ему необходим проект, описанный в проектном файле say.rb. За все остальное отвечает Mxx_ru — компиляция подчиненного проекта, определение имени получившейся библиотеки, линкование основного модуля с библиотекой say.

3.3 exe-файл, динамическая и статическая библиотеки

Пусть теперь требуется усложнить два предыдущих примера — требуется создать динамически-загружаемую библиотеку, предоставляющую класс inout_t. Задачей данного класса является печать строки “Hello!” в конструкторе и строки “Bye!” в деструкторе. Объекты класса inout_t можно будет объявлять в функциях для того, чтобы на стандартный поток ошибок печатались сообщения в входе/выходе в/из функции.

Для своей работы библиотека inout будет использовать модифицированный вариант библиотеки say — в функции say_hello, say_bye добавляется еще один аргумент. В результате библиотека say будет состоять из следующих файлов:

- заголовочный say.hpp

```
#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
```

- файл реализации say.cpp

```
#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;
  }
```

- проектный файл say.rb

```
require 'mxx_ru/cpp'

Mxx_ru::setup_target(
  Mxx_ru::Cpp::Lib_target.new( "say.rb" ) {
    target( "say" )

    include_path( "." )

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

Библиотека inout состоит из следующих файлов:

- заголовочный inout.hpp

```
#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
```

- файл реализации inout.cpp

```
#include <say.hpp>

#include <inout.hpp>

```

```
5  inout_t::inout_t(
6    const std::string & method )
7    : m_method( method )
8    {
9      say_hello( std::cout, method );
10   }
11
12 inout_t::~inout_t()
13   {
14     say_bye( std::cout, m_method );
15   }
```

- проектный файл inout.rb

```
1  require 'mxx_ru/cpp'
2
3  Mxx_ru::setup_target(
4    Mxx_ru::Cpp::Dll_target.new( "inout.rb" ) {
5      target( "inout" )
6      implib_path( "." )
7
8      required_prj( "say.rb" )
9
10     include_path( "." )
11
12     define( "INOUT_PRJ" )
13
14     if "mswin" == toolset.tag( "target_os" )
15       define( "INOUT_MSWIN" )
16     end
17
18     cpp_source( "inout.cpp" )
19   }
20 )
```

Основной модуль состоит из файлов:

- файл реализации main.cpp

```
1  #include <iostream>
2
3  #include <inout.hpp>
4
5  void
6  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   }
```

- проектный файл prj.rb

```
1  require 'mxx_ru/cpp'
2
3  Mxx_ru::setup_target(
4    Mxx_ru::Cpp::Exe_target.new( "prj.rb" ) {
5      target( "exe_dll_lib" )
6
7      required_prj( "inout.rb" )
8
9      include_path( "." )
10
11     cpp_source( "main.cpp" )
12   }
13 )
```

Важно отметить, что основной модуль напрямую зависит только от подпроекта inout. Неявная зависимость основного модуля от библиотеки say отслеживается Mxx_ru. Для таких платформ, как mswin и os2 эта неявная зависимость не столь важна, как на unix-платформах. В частности, при линковании основного модуля на платформах unix нужно будет указывать линкеру не только библиотеку inout, но и библиотеку say. В случае с Mxx_ru основной модуль об этом знать не должен — Mxx_ru в состоянии самостоятельно извлечь из подпроектов всю необходимую информацию и распорядиться ей в зависимости от используемой платформы.

3.4 exe-файл, динамическая, статическая библиотеки и исходные файлы в разных каталогах

В предыдущем примере есть два серьезных недостатка:

1. Все файлы (заголовочные, исходные, проектные, результаты компиляции и линковки) находятся в одном каталоге. Так, после компиляции проекта g++ на платформе Cygwin в каталоге размещаются следующие файлы:

```
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
```

Очевидно, что если количество файлов в проекте будет увеличиваться, то ориентироваться в такой каше из файлов разных типов будет все сложнее и сложнее.

2. В каждом из проектных файлов есть инструкция: `include_path(".")`. Было бы удобнее, если бы существовало одно место, в котором можно было бы задать общие для всей группы подпроектов параметры.

Для преодоления первого недостатка нужно разместить все подпроекты в собственных подкаталогах одного общего каталога (корневого каталога проекта). Например, пусть будет следующая иерархия каталогов и файлов:

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

Каталог lib предназначен для размещения библиотек (статических и библиотек импорта для dll).

3.4.1 Подпроект say

Подпроект say состоит из следующих файлов:

- заголовочный say.hpp

```
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
```

- файл реализации say.cpp

```
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   }
```

  Отдельно нужно обратить внимание на то, что заголовочный файл say.hpp загружается как `<say/say.hpp>`;

- проектный файл prj.rb

```
1  require 'mxx_ru/cpp'
2
3  Mxx_ru::setup_target(
4    Mxx_ru::Cpp::Lib_target.new( "say/prj.rb" ) {
5      target_root( "lib" )
6      target( "say" )
7
8      cpp_source( "say.cpp" )
9    }
10 )
```

3.4.2 Подпроект inout

Подпроект main состоит из следующих файлов:

- заголовочный inout.hpp

```
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
```

- файл реализации inout.cpp

```
1  #include <say/say.hpp>
2
3  #include <inout/inout.hpp>
4
5  inout_t::inout_t(
6    const std::string & method )
7    : m_method( method )
8    {
9      say_hello( std::cout, method );
10   }
11
12 inout_t::~inout_t()
13   {
14     say_bye( std::cout, m_method );
15   }
```

Здесь так же следует обратить внимание, что для загрузки заголовочных файлов из какого-либо подпроекта, в директиве `#include` указывается имя подкаталога этого проекта: `<say/say.hpp>`, `<inout/inout.hpp>`.

- проектный файл prj.rb

```
1  require 'mxx_ru/cpp'
2
3  Mxx_ru::setup_target(
```

```
4   Mxx_ru::Cpp::Dll_target.new( "inout/prj.rb" ) {
5     target( "inout" )
6     implib_path( "lib" )
7
8     required_prj( "say/prj.rb" )
9
10    define( "INOUT_PRJ" )
11
12    if "mswin" == toolset.tag( "target_os" )
13      define( "INOUT_MSWIN" )
14    end
15
16    cpp_source( "inout.cpp" )
17  }
18 )
```

3.4.3 Подпроект main

Подпроект main состоит из следующих файлов:

- файл реализации main.cpp

```
1  #include <iostream>
2
3  #include <inout/inout.hpp>
4
5  void
6  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   }
```

- проектный файл prj.rb

```
1  require 'mxx_ru/cpp'
2
3  Mxx_ru::setup_target(
```

```
4   Mxx_ru::Cpp::Exe_target.new( "main/prj.rb" ) {
5     target( "exe_dll_lib" )
6
7     required_prj( "inout/prj.rb" )
8
9     cpp_source( "main.cpp" )
10  }
11 )
```

3.4.4 Файл build.rb

Файл build.rb — это специальный проектный файл. Он должен располагаться в корневом каталоге проекта, в том каталоге, из которого запускается интерпретатор Ruby. В файле build.rb должны быть заданы все параметры, глобальные для всего проекта.

Если проект состоит из нескольких подпроектов, как в данном случае, то build.rb как правило, является проектом-композитом.

Для данного примера build.rb имеет вид:

```
1 require 'mxx_ru/cpp'
2
3 Mxx_ru::setup_target(
4   Mxx_ru::Cpp::Composite_target.new( Mxx_ru::BUILD_ROOT ) {
5     required_prj( "main/prj.rb" )
6
7     global_include_path( "." )
8   }
9 )
```

3.4.5 Компиляция всего проекта

Для компиляции примера необходимо войти в корневой каталог примера (в котором находится файл build.rb) и запустить интерпретатор Ruby с файлом build.rb в качестве параметра. В результате компиляции g++ на платформе Cygwin в корневом каталоге окажутся файлы:

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

а в каталоге lib файлы:

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

3.4.6 Компиляция только одного подпроекта

Для компиляции только одного подпроекта необходимо находясь в корневом каталоге примера (т.е. там, где расположен build.rb) запустить интерпретатор Ruby, указав ему имя проектного файла и аргумент `--mxx-cpp-1`. Например, для компиляции только inout:

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

В этом случае будет обработан только сам подпроект inout без своих подпроектов. Т.е. если подпроект say не был перед этим скомпилирован, то линковка inout завершиться неудачей из-за того, что нет библиотеки say.

3.5 Операция clean

По-умолчанию, Mxx_ru выполняет построение проекта (в случае C/C++ это компиляция и линкование). Но Mxx_ru так же поддерживает и обратную операцию – *clean*, т.е. удаление всего, что было построено. Для C/C++ операция clean приводит к уничтожению всех объектных файлов, библиотек, исполнимых файлов и других результатов компиляции и линкования.

Для выполнения операции clean необходимо запустить интерпретатор ruby, передать ему имя проектного файла и указать аргумент `--mxx-clean`:

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

Операция clean распространяется как на сам проект, имя проектного файла которого передано Ruby-интерпретатору, так и на все подпроекты. Если для C/C++ проектов необходимо выполнить очистку только одного проекта, без его подпроектов, то интерпретатору Ruby нужно передать имя проектного файла этого проекта и аргументы `--mxx-clean`, `--mxx-cpp-1`.

```
ruby build.rb --mxx-clean --mxx-cpp-1
```

Глава 4

Основная идея

Основная идея Mxx_ru заключается в том, что каждый проект определяет *target* (цель), которая должна быть построена. Каждая цель описывается экземпляром класса, производного от `Mxx_ru::Abstract_target`. Каждая цель описывается в одном проектном файле. Каждая цель должна иметь уникальный псевдоним (*prj_ alias*). В Mxx_ru псевдонимом цели является имя проектного файла, в котором описывается цель.

Задачей проектного файла является создание экземпляра объекта-цели и передача этого объекта в Mxx_ru с помощью функции `Mxx_ru::setup_target`. Поэтому проекты в Mxx_ru, как правило, имеют вид:

```
1 require 'mxx_ru/<что-то>'
2
3 Mxx_ru::setup_target(
4   <какой-то класс>.new( <имя проектного файла> ) {
5     <настройка параметров проекта>
6   }
7 )
```

4.1 Подробнее о Mxx_ru::Abstract_target

Класс `Mxx_ru::Abstract_target` является базовым классом для всех целей. Он определяет два самых важных метода: `build` для построения цели и `clean` для очистки цели. В самом классе `Mxx_ru::Abstract_target` эти методы считаются абстрактными (чистыми виртуальными, в терминологии C++), т.е. должны быть переопределены в производных классах. Но, поскольку в Ruby нет понятия чистого виртуального метода, то попытка вызова методов `build`, `clean` из класса `Mxx_ru::Abstract_target` приведет к порождению исключения `Mxx_ru::Abstract_method_ex`.

Получить псевдоним цели можно с помощью метода `prj_alias`.

4.1.1 Имена производимых целью файлов

Как правило, цель определяется для того, чтобы получить один результирующий файл. Например, исполнимый файл для C++ проектов, DVI-файл для LaTeX проектов или jar-файл для Java проектов. Но, в общем случае, цель может приводить к построению нескольких файлов — это зависит от типа проекта.

Сколько бы файлов не производила цель, все их полные имена должны быть переданы в базовый класс `Mxx_ru::Abstract_target` посредством метода `mxx_add_full_target_name`. Получить имена всех производимых целью файлов можно с помощью метода `mxx_full_target_names`.

Важно, чтобы в имена производимых файлов не включались имена промежуточных результатов построения цели. Например, для C/C++ проектов такими промежуточными результатами являются объектные файлы. Их имена не должны быть доступны через `mxx_full_target_names`.

4.1.2 Необходимые подпроекты

Часто бывает, что цель нуждается в файлах, производимых другой целью из другого проектного файла. Например, для линкования exe-файла могут потребоваться lib-файлы, создаваемые другими подпроектами. В этих случая проект (цель) нуждается в подпроектах (подчиненных проектах, подчиненных целях).

Подпроектом должен быть проектный Mxx_ru-файл, созданный по обычным правилам формирования проектных Mxx_ru-файлов. Подпроекты указываются при помощи метода `required_prj`:

```
1  require 'mxx_ru/cpp'
2
3  Mxx_ru::setup_target(
4    Mxx_ru::Cpp::Composite_target.new( Mxx_ru::BUILD_ROOT ) {
5
6      global_include_path( "." )
7
8      required_prj( "oess_1/stdsn/prj.rb" )
9      required_prj( "oess_1/scheme/prj.rb" )
10     required_prj( "oess_1/util_cpp_serializer/prj.rb" )
11     required_prj( "oess_1/file/prj.rb" )
12     required_prj( "oess_1/db/prj.rb" )
13     required_prj( "oess_1/util_ent_enum/prj.rb" )
14     required_prj( "oess_1/util_slice_create/prj.rb" )
15     required_prj( "oess_1/tlv/prj.rb" )
16   }
17 )
```

Метод `required_prj` не только сохраняет в описании проекта имя его подпроекта — в нем происходит загрузка и обработка файла подпроекта.

Получить имена всех подпроектов можно с помощью метода `mxx_required_prjs`.

Обработка подпроектов может зависить от типа проекта, но исходя из здравого смысла, можно ожидать, что в методе `build` сначала вызываются методы `build` для всех подпроектов. Например, в C++ проектах так и происходит.

4.1.3 Генераторы исходных текстов

В некоторых случаях требуется из файлов определенного типа при помощи специальных инструментов сгенерировать исходные файлы. Например, из .y-файлов с помощью уасс генерируется исходный код синтаксических анализаторов.

Различные генераторы работают с разными файлами, получают различные наборы параметров и производят разное количество различных исходных файлов. Например, инструмент uic из Qt может производить как заголовочные, так и файлы реализации.

Поэтому в Mxx_ru используется обобщенное понятие генератора. Есть базовый класс `Mxx_ru::Abstract_generator`, который определяет два метода: `build` и `clean`. Все генераторы должны быть производными от этого класса.

Поскольку различные генераторы требуют различных способов работы с ними, то Mxx_ru никак не специфицирует других интерфейсов генераторов. Единственное требование: генераторы должны быть переданы в `Mxx_ru::Abstract_target` с помощью метода `generator`.

За запуск генераторов отвечает конкретный класс, производный от `Mxx_ru::Abstract_target`. Например, классы целей для C++ проектов запускают генераторы после того, как обработают все подпроекты и перед тем, как начнется компиляция самого проекта.

Т.к. Mxx_ru не определяет интерфейса, с помощью которого любому генератору можно дать указания по генерации, то обычно использование конкретного генератора выглядит следующим образом:

```
require 'mxx_ru/<что-то>'

require '<файл с описанием генератора>'

Mxx_ru::setup_target(
  Mxx_ru::<какой-то класс цели>.new( <псевдоним> ) {
    ...
    gen = generator( <класс генератора>.new( <параметры> ) )
    gen.<какой-то метод>( <параметры> )
    ...
  }
)
```

Например:

```
require  'mxx_ru/cpp'

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

Mxx_ru::setup_target(
  Mxx_ru::Cpp::Dll_target.new( "oess_1/db/prj.rb" ) {

    ...
    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" )
```

```
18
19          cpp_source( "slice_unit.cpp" )
20          ddl_cpp_generator.ddl_file( "slice_unit.ddl" )
21
22          cpp_source( "slice_stream_item.cpp" )
23          ddl_cpp_generator.ddl_file( "slice_stream_item.ddl" )
24
25          cpp_source( "db_slicemap.cpp" )
26          cpp_source( "std_db_slicemap.cpp" )
27
28          cpp_source( "db_struct.cpp" )
29          cpp_source( "std_db_struct.cpp" )
30        }
31      }
32      ...
33    }
34  )
35
```

Либо, учитывая, что метод `generator` возвращает переданный ему объект-генератор, можно вызывать методы генератора не сохраняя ссылку на него в отдельной переменной:

```
1  require  'mxx_ru/cpp'
2
3  require 'oess_1/util_cpp_serializer/gen'
4
5  Mxx_ru::setup_target(
6    Mxx_ru::Cpp::Exe_target.new( "test/stdsn/shptr/prj.rb" ) {
7
8      target( "test.stdsn.shptr" )
9
10     required_prj( "oess_1/defs/prj.rb" )
11     required_prj( "oess_1/io/prj.rb" )
12
13     generator( Oess_1::Util_cpp_serializer::Gen.new( self ) ).
14       ddl_file( "main.ddl" )
15
16     cpp_source( "main.cpp" )
17   }
18 )
```

Получить список всех установленных генераторов можно с помощью метода `mxx_generators`.

Еще один способ применения генераторов описан в 6.6 на стр. 50

4.2 Подробнее о Mxx_ru::setup_target

Mxx_ru поддерживает карту целей — псевдонимам (т.е. именам проектных файлов) сопоставлены объекты-цели. Функция `Mxx_ru::setup_target` предназначена для занесения очередной цели в эту карту.

Кроме того, функция `Mxx_ru::setup_target` автоматически запускает построение или очистку проекта, если переданная в `Mxx_ru::setup_target` цель является самой верхней (т.е. цель создается в проектном файле, имя которого передано интерпретатору Ruby)[1].

При обработке самой верхней цели Функция `Mxx_ru::setup_target` проверяет наличие аргумента `--mxx-clean`. Если этот аргумент указан, то у цели вызывается метод `clean`. В противном случае у цели вызывается метод `build`.

[1] В действительности самой верхней целью является цель, для которой конструктор класса `Mxx_ru::Abstract_Target` был вызван самым первым.

Глава 5

Mxx_ru для C/C++ проектов

5.1 Введение

Mxx_ru предоставляет набор готовых средств для поддержки компиляции C/C++ проектов. Эти средства описаны в файле `mxx_ru/cpp`, который необходимо загружать директивой `require`:

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

Для поддержки C/C++ в Mxx_ru выделены важные понятия: *toolset* и *obj_placement*.

Mxx_ru предоставляет готовые классы для таких типов целей, как exe-файл, dll-файл (с поддержкой библиотек импорта), lib-файл, композитный проект.

5.1.1 Понятие toolset

Toolset — это набор инструментов, применяемых для компиляции проекта. Проще говоря, toolset определяет тип, версию и другие особенности конкретного компилятора на конкретной платформе.

Mxx_ru уже адаптирован к ряду компиляторов. В частности поддерживаются компиляторы Borland C++ (версий 5.* на платформе mswin); Visual C++ (версий 7.* и 6.* на платформе mswin); GNU C++ на платформах unix, mswin (через порты mingw и cygwin); c89 для HP NonStop (native-версия и кросс-компилятор из eToolkit для mswin).

Для корректной работы Mxx_ru тип toolset должен быть указан Mxx_ru во время настройки (см. 5.2 на стр. 34).

Toolset представляется объектом класса, производного от `Mxx_ru::Cpp::Toolset`. Объект-toolset создается Mxx_ru и используется для всего проекта (т.е. для проекта, имя файла которого указано интерпретатору Ruby и для всех его подпроектов). Объект-toolset нельзя заменить. Получить доступ к объекту-toolset можно через метод `toolset()`.

Имена toolset

Каждый toolset имеет собственное имя, доступное через метод `name` класса `Mxx_ru::Cpp::Toolset`. Это имя может использоваться для настройки проекта на кон-

кретный компилятор. Например:

```
Mxx_ru::Cpp::Exe_target.new( "prj.rb" ) {
  ...
  if "vc" == toolset.name
    # Адаптация к Visual C++.
    cpp_source( "mswin/vc/exception_handler.cpp" )
  elif "gcc" == toolset.name
    # Адаптация к GNU C++.
    cpp_source( "gcc/exception_handler.cpp" )
  end
  ...
}
```

Определенные на данный момент имена компиляторов перечислены в таблице 5.1.

bcc	Компилятор Borland C++ (платформа mswin).
c89_nsk	Компилятор c89 для HP NonStop (платформы HP NonStop и mswin).
gcc	Компилятор GNU C/C++ (платформы unix, mswin).
vc	Компилятор Visual C++ (платформа mswin).

Таблица 5.1: Имена реализованых toolset.

Теги toolset

Т.к. toolset адаптирован к конкретному компилятору на конкретной платформе, то в проекте можно использовать toolset для настройки проекта на данную платформу. Для этих целей в toolset есть понятие тегов. Тег — это уникальны текстовый ключ, которому сопоставлено текстовое значение. Теги устанавливаются самими toolset, а так же их можно указать при настройке Mxx_ru.

Для получения значения тега предназначен метод `tag(a_name, a_default=nil)`. Если в toolset установлен тег с таким именем, то возвращается его значение. В противном случае возвращается значение аргумента `a_default`. Если тег не установлен и `a_default==nil`, то порождается исключение `Mxx_ru::Cpp::Toolset::Unknown_tag_ex`.

Ряд тегов считаются обязательными, они должны быть определены для всех toolset на всех платформах (см. таблицу 5.2).

host_os	Операционная система, на которой производится компиляция.
target_os	Операционная система, для которой производится компиляция.

Таблица 5.2: Обязательные теги для всех toolset.

Значения тегов host_os и target_os могут различаться в случае использования кросс-компилятора, например, c89 для HP NonStop.

Операционные системы идентифицируются перечисленными в таблице 5.3 именами.

На платформе unix должен быть определен обязательный тег *unix_port*, который определяет конкретный тип unix. Рекомендуется использовать перечисленные в таблице 5.4 значения[1].

[1] Другие значения будут добавляться по мере адаптации Mxx_ru к другим платформам.

mswin	32-битовый вариант Microsoft Windows.
tandem_oss	Подсистема OSS (Open System Services) для HP NonStop.
unix	Различные варианты Unix.

Таблица 5.3: Имена операционных систем.

bsd	FreeBSD/NetBSD/OpenBSD.
cygwin	Cygwin на Microsoft Windows.
linux	GNU/Linux.
solaris	Sun Solaris.

Таблица 5.4: Имена вариантов Unix.

При настройке Mxx_ru могут быть указаны и другие теги, которые необходимы проекту.

Пример работы с тегами:

```
Mxx_ru::Cpp::Dll_target.new( "my.rb" ) {
  # Под Unix помещаем результат компиляции в подкаталог lib.
  if "unix" == toolset.tag( "target_os" )
    target_root( "lib" )
  end
  target( "my" )

  # Если используется архитектура процессора, отличная от Intel x86,
  # то ее имя должно быть указано в собственном теге "arch".
  if "x86" == toolset.tag( "arch", "x86" )
    ...
  else
    ...
  end
  ...
}
```

5.1.2 Понятие obj_placement

Mxx_ru позволяет управлять размещением результатов компиляции и линковки C/C++ проекта. Для этих целей используется понятие *obj_placement* — объекта производного от `Mxx_ru::Cpp::Obj_placement` класса. Этот объект указывает Mxx_ru куда помещать объектные файлы, куда библиотеки, куда скомпилированные ресурсные файлы и т.д.

Проект может сам указать необходимый ему тип obj_placement. Для этого предназначен метод `obj_placement`. Например:

```
Mxx_ru::Cpp::Dll_target.new( "my.rb" ) {
  ...
  obj_placement(
    Mxx_ru::Cpp::Runtime_subdir_obj_placement.new(
      "output" ) )
```

```
6   ...
7 }
```

Проект может назначить глобальный obj_placement при помощи функции `global_obj_placement`. В этом случае действие нового obj_placement будет распространяться на все подпроекты.

Попытка установить одновременно и локальный и глобальный obj_placement является ошибкой и приведет к исключению. Рекомендуется устанавливать только глобальный obj_placement в самом верхнем, обычно композитном, проекте.

В состав Mxx_ru входят два штатных класса для obj_placement:

- `Mxx_ru::Cpp::Source_subdir_obj_placement`, который помещает все объектные файлы (.obj, .o) в указанный подкаталог того каталога, в котором находится исходный файл. По-умолчанию, используется подкаталог `o`. Например, если есть исходный файл `src/win/init.cpp`, то объектный файл для него будет размещен в `src/win/o`. Если нужного подкаталога нет, то он будет создан. Таким же образом обрабатываются скомпилированные ресурсные файлы на платформе mswin. Размещение остальных результатов компиляции (exe, lib, dll, библиотеки импорта и т.д.) определяются на основании заданных в проекте значений относительно текущего каталога.

- `Mxx_ru::Cpp::Runtime_subdir_obj_placement`, который помещает все результаты компиляции в каталог, имя которого зависит от типа runtime. При этом в каталоге-приемнике формируется исходное дерево каталогов.

Например, если есть дерево каталогов:

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

и используется `Source_subdir_obj_placement`, то после компиляции дерево каталогов примет вид:

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

Если же используется `Runtime_subdir_obj_placement` с подкаталогом `output` в качестве корневого и с режимом компиляции debug, то получится следующе дерево каталогов:

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

При этом все результаты компиляции, включая exe, dll, lib и т.д., будут помещены в `output/debug`.

Если obj_placement не задан явно, то используется `Source_subdir_obj_placement`.

5.1.3 Цели для C/C++ проектов

Для C/C++ проектов в Mxx_ru существуют следующие классы целей:

- `Mxx_ru::Cpp::Exe_target` для построения exe-файлов;
- `Mxx_ru::Cpp::Lib_target` для построения lib-файлов (статических библиотек);
- `Mxx_ru::Cpp::Dll_target` для построения dll-файлов (динамически-загружаемых библиотек и библиотек импорта для них);
- `Mxx_ru::Cpp::Composite_target` для композитных проектов.

Для C/C++ проекта в проектном файле необходимо создать и передать в функцию `Mxx_ru::setup_target` объект одного из этих классов.

Все классы имеют конструкторы, которые могут получать блок кода (специфическая особенность языка Ruby). Поэтому для описания проекта не обязательно создавать свой класс, производный от одного из указанных выше классов. Достаточно создать объект одного из существующих классов и передать в его конструктор блок, в котором производится настройка проекта:

```
1 Mxx_ru::Cpp::Exe_target.new( ... ) { ... }
```

Примечание. Для того, чтобы Ruby считал блок в фигурных скобках параметром конструктора необходимо, чтобы открывающая фигурная скобка этого блока располагалась на той же строке, что и закрывающая круглая скобка вызова конструктора. Например:

```
1 # Правильно.
2 Mxx_ru::Cpp::Exe_target.new( "my.rb" ) { target( "my" ) c_source( "my.c" ) }
3
4 # Правильно.
```

```
5  Mxx_ru::Cpp::Exe_target.new( "my.rb" ) { target( "my" )
6    c_source( "my.c" ) }
7
8  # Правильно.
9  Mxx_ru::Cpp::Exe_target.new( "my.rb" ) {
10   target( "my" )
11   c_source( "my.c" )
12 }
13
14 # Не правильно!
15 Mxx_ru::Cpp::Exe_target.new( "my.rb" )
16   {
17     target( "my" )
18     c_source( "my.c" )
19   }
20
21 # Правильно. Обратный слэш в конце строки означает,
22 # что на следующей строке находится продолжение данной строки.
23 Mxx_ru::Cpp::Exe_target.new( "my.rb" ) \
24   {
25     target( "my" )
26     c_source( "my.c" )
27   }
```

5.1.4 Порядок выполнения build/clean

При построении C/C++ целей выполняются следующие действия:

1. Выполняется построение всех подпроектов.
2. Выполняется вызов метода `build` у всех генераторов исходного кода.
3. Запускается анализатор C/C++ зависимостей.
4. Выполняется компиляция всех C файлов.
5. Выполняется компиляция всех C++ файлов.
6. Если на платформе mswin определены ресурсы, то выполняется компиляция ресурсов.
7. Если типом цели является lib, то строится статическая библиотека. В случае exe и dll осуществляется линковка exe- или dll-файла соответственно. Если для dll определено расположение библиотеки импорта, то строится библиотека импорта.

При очистке C/C++ целей выполняются следующие действия:

1. Выполняется очистка всех подпроектов.
2. Выполняется вызов метода `clean` у всех генераторов исходного кода.
3. Удаляются объектные файлы для всех C файлов.
4. Удаляются объектные файлы для всех C++ файлов.

5. Если на платформе mswin определены ресурсы, то удаляются скомпилированные ресурсные файлы.

6. Если типом цели является lib, то удаляется статическая библиотека. В случае exe и dll удаляются exe- и dll-файлы. Если для dll определено расположение библиотеки импорта, то библиотека импорта так же удаляется.

Примечание. Класс `Composite_target` выполняет построение и очистку только подпроектов.

5.1.5 Режимы runtime

Mxx_ru позволяет строить C/C++ проекты в трех режимах runtime:

- debug. В этом режиме используются отладочные библиотеки runtime. Компилятору и линкеру указываются опции для генерации и включения в результирующую цель отладочной информации;
- default. В этом режиме используются release-библиотеки runtime, но ни компилятору, ни линкеру не задаются какие-либо опции компилятора. Т.е. получившуюся цель нельзя отлаживать, но на компиляцию уходит меньше времени, чем на компиляцию с включенной оптимизацией. Макрос NDEBUG не определяется (т.е. будут работать assert-ы). Этот режим используется по-умолчанию, если не указаны режимы debug, release;
- release. В этом режиме используются release-библиотеки runtime. Компилятору и линкеру указываются опции для включения оптимизации. Определяется макрос NDEBUG.

Указать режим runtime можно двумя способами:

1. В проектном файле при помощи функции `runtime_mode`. Например:

```
1 Mxx_ru::Cpp::Exe_target( "prj.rb" ) {
2   runtime_mode( Mxx_ru::Cpp::RUNTIME_RELEASE )
3   ...
4 }
```

2. В командной строке посредством аргументов `--mxx-cpp-release` или `--mxx-cpp-debug`. Например:

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

5.1.6 Локальные, глобальные и распространяемые параметры проекта

Описание C/C++ проекта в Mxx_ru состоит в перечислении таких параметров, как имя результирующего файла, имена исходных файлов и необходимых библиотек, необходимых define-символов, необходимых опций компилятора, линкера и т.д. Часть этих

параметров могут относиться только к самому проекту. Например, имена исходных файлов. Но часть параметров, т.к. define-символы и опции компилятора, разделяются на три типа:

Локальные параметры распространяются только на сам проект. Не оказывают никакого воздействия ни на подпроекты, ни на проекты, которые используют данный проект в качестве подпроекта.

Пример установления локального define-символа:

```
Mxx_ru::Cpp::Dll_target.new( "my_dll/prj.rb" ) {
  target( "my_dll" )
  implib_path( "lib" )

  cpp_source( "my_dll/impl.cpp" )

  defines( "MY_DLL_PRJ" )
}
```

Установленный таким образом символ MY_DLL_PRJ будет доступен только при компиляции `my_dll/impl.cpp`.

Глобальные параметры распространяются на все проекты.

Глобальные параметры задаются с помощью функций с именами вида `global_<имя>`, где <имя> — это имя функции для установки локального параметра того же назначения. Например: global_define, global_include_path, global_compiler_option и т.д.

Пример установки глобального пути поиска заголовочных файлов для композитного проекта:

```
Mxx_ru::Cpp::Composite_target.new( Mxx_ru::BUILD_ROOT ) {

  global_include_path( "." )

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

В этом случае при компиляции всех файлов из проектов `engine/prj.rb` и `interface/prj.rb` в качестве пути поиска заголовочных файлов будет использован текущий каталог.

Пример установки глобальной опции компилятора.

```
Mxx_ru::Cpp::Dll_target.new( "packer/prj.rb" ) {
  ...
  # Должен использоваться режим выравнивания в 4 байта.
  if "vc" == toolset.name
    global_compiler_option( "-Zp4" )
  end
```

```
7   ...
8 }
```

В этом случае при компиляции всех файлов всех проектов, которые оказались в одном общем проекте с `packer/prj.rb` компилятору Visual C++ будет передаваться опция `-Zp4`.

Распространяемые (upspread) параметры распространяются только на сам проект и на те проекты, которые используют данный проект в качестве подпроекта (вне зависимости от степени вложенности).

Upspread-параметры устанавливаются теми же функциями, что и локальные параметры, но с указанием в качестве второго аргумента константы `Mxx_ru::Cpp::Target::OPT_UPSPREAD`.

Пример установки upspread define-символа и upspread пути поиска заголовочных файлов:

```
1  Mxx_ru::Cpp::Lib_target.new( "pcre/prj.rb" ) {
2    target_root( "lib" )
3    target( "pcre.4.5.0" )
4
5    c_source( "get.c" )
6    c_source( "maketables.c" )
7    c_source( "pcre.c" )
8    c_source( "study.c" )
9
10   define( "PCRE_STATIC", Mxx_ru::Cpp::Target::OPT_UPSPREAD )
11   define( "SUPPORT_UTF8", Mxx_ru::Cpp::Target::OPT_UPSPREAD )
12
13   include_path( "pcre", Mxx_ru::Cpp::Target::OPT_UPSPREAD )
14 }
```

В этом случае при компиляции всех проектов, которые прямо или косвенно используют проект `pcre/prj.rb` в качестве подпроекта (т.е. указывают его имя, либо имя использующего его проекта в `required_prj`) в качестве define-символов будут определены `PCRE_STATIC` и `SUPPORT_UTF8`, а в качестве пути для поиска заголовочных файлов будет использоваться подкаталог `pcre`.

5.1.7 Аргумент mxx-cpp-1

Если в командной строке интерпретатору Ruby указан аргумент `--mxx-cpp-1`, то операции build и clean не распространяются на подпроекты.

Аргумент `--mxx-cpp-1` удобно применять, когда выполняются работы над одним подпроектом в составе большого композитного проекта. Запуск построения всего композита может занимать много времени, что расточительно, если вносимые в подпроект изменения не требуют перекомпиляции и перелинковки остальных подпроектов. В этом случае интерпретатору Ruby в командной строке указывается не `build.rb`, а имя проектного файла и аргумент `--mxx-cpp-1`:

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

Аргумент `--mxx-cpp-1` может использоваться не только для операции *build*, но и для операции *clean* — в этом случае будут удалены только файлы, относящиеся непосредственно к указанного проекту.

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

5.1.8 Аргумент mxx-cpp-no-depends-analyzer

Аргумент `--mxx-cpp-no-depends-analyzer`[2] указывает Mxx_ru не проводить поиск C/C++ зависимостей путем анализа исходных и заголовочных файлов. По умолчанию такой анализ производится, что может значительно замедлять процесс компиляции. Иногда (например, при полной перекомпиляции проекта) поиском зависимостей можно пренебречь. В таких случаях интерпретатору Ruby можно указать агрумент `--mxx-cpp-no-depends-analyzer`:

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

5.2 Настройка Mxx_ru для работы с C/C++ проектами

Для использования возможностей Mxx_ru для работы с C/C++ проектами необходимо определить переменную среды `MXX_RU_CPP_TOOLSET`. Значение этой переменной среды должно иметь вид:

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

где `<file>` – это имя .rb-файла из состава Mxx_ru, который отвечает за создания объекта-toolset. Поддерживаемые toolset перечислены в таблице 5.5. Пары значений tag, value будут установлены в качестве тегов выбранного toolset.

mxx_ru/cpp/toolsets/bcc_win32_5	Borland C++ 5.* для Microsoft Windows
mxx_ru/cpp/toolsets/c89_etk_nsk	c89 для HP NonStop из eToolkit для Microsoft Windows
mxx_ru/cpp/toolsets/c89_nsk	c89 для HP NonStop
mxx_ru/cpp/toolsets/gcc_cygwin	GNU C/C++ для Cygwin
mxx_ru/cpp/toolsets/gcc_mingw	Minimalist GNU C/C++ для Windows
mxx_ru/cpp/toolsets/gcc_unix	GNU C/C++ для Unix
mxx_ru/cpp/toolsets/vc7	Visual C++ 7.* для Microsoft Windows

Таблица 5.5: Список поддерживаемых Mxx_ru toolset.

Примеры:

```
export MXX_RU_CPP_TOOLSET=mxx_ru/cpp/toolset/gcc_unix unix=linux arch=x86
```

```
set MXX_RU_CPP_TOOLSET=mxx_ru/cpp/toolset/vc7
```

[2] Добавлен в версии 1.0.9.

5.3 Получение доступа к toolset

Получить объект-toolset из проекта можно обратившись к методу `toolset` класса `Mxx_ru::Cpp::Target`. Например:

```
Mxx_ru::Cpp::Exe_target.new( "my/prj.rb" ) {
  ...
  if "vc" == toolset.name
    ...
  end
  ...
}
```

Примечание. Т.к. `toolset` является не статическим методом, его можно вызывать только для созданного объекта-цели. Например, как показано выше, в блоке кода, переданного в конструктор в качестве аргумента.

5.4 Установка режима runtime в проектном файле

Установить режим runtime в проектном файле можно при помощи метода `runtime_mode`. Значения режима runtime задаются константами: `Mxx_ru::Cpp::RUNTIME_DEBUG`, `Mxx_ru::Cpp::RUNTIME_DEFAULT`, `Mxx_ru::Cpp::RUNTIME_RELEASE`. По-умолчанию используется режим default.

Режим runtime является глобальным параметром. Он распространяется на все проекты сразу. Попытка двух проектов установить разные значения runtime приведет к порождению исключения.

Получить текущее значение режима runtime можно с помощью метода `mxx_runtime_mode`.

См. так же 5.1.5 на стр. 31.

5.5 Установка типа runtime library

На платформах, которые поддерживают динамически-загружаемые библиотеки, некоторые toolset позволяют выбрать один из двух типов runtime library (rtl): static (код rtl влинковывается в само приложение) и shared (код rtl находится в отдельной dll, к которой линкуется приложение). Если приложение использует собственные dll и передает C++ объекты созданные в одной dll для уничтожения в другую dll, то такое приложение должно использовать shared вариант rtl (в этом случае все dll имеют общий heap).

Установить тип rtl можно при помощи метода `rtl_mode`. Тип rtl задается с помощью констант: `Mxx_ru::Cpp::RTL_DEFAULT` (проекту все равно, какой тип rtl будет использоваться), `Mxx_ru::Cpp::RTL_STATIC` (требуется использовать статический вариант rtl), `Mxx_ru::Cpp::RTL_SHARED` (требуется использовать разделяемый вариант rtl). По-умолчанию значение rtl_mode установлено в `Mxx_ru::Cpp::RTL_DEFAULT` (т.н. default режим).

Тип rtl является глобальным параметром. Он распространяется на все проекты сразу. Попытка двух проектов установить разные значения типа rtl приведет к порождению исключения.

Если тип rtl не был установлен (т.е. остался default-режим), то тип используемой rtl определяется toolset. Как правило, используется rtl, который компилятор подставляет по-умолчанию, если ему не дано никаких явных указаний на этот счет.

Пример:

```
Mxx_ru::Cpp::Dll_target.new( "my/prj.rb" ) {
  target_root( "lib" )
  target( "my" )

  rtl_mode( Mxx_ru::Cpp::RTL_SHARED )
  ...
}
```

Получить текущее значение типа rtl можно с помощью метода `mxx_rtl_mode`.

5.6 Установка режима многопоточности

На некоторых платформах, например, mswin, требуется явно указывать, нуждается ли приложение в поддержке многопоточности или нет. Для установления режима многопоточности в Mxx_ru используется метод `threading_mode`. Режим многопоточности определяется с помощью констант `Mxx_ru::Cpp::THREADING_DEFAULT` (проект не предъявляет никаких требований к режиму многопоточности, является однопоточным, но может работать и в многопоточном приложении), `Mxx_ru::Cpp::THREADING_MULTI` (проект требует поддержки многопоточности), `Mxx_ru::Cpp::THREADING_SINGLE` (проект расчитан только на однопоточность, не может использоваться совместно с многопоточными проектами). По-умолчанию режим многопоточности выставлен в `Mxx_ru::Cpp::THREADING_DEFAULT`.

Если режим многопоточности отличается от `Mxx_ru::Cpp::THREADING_MULTI` и компилятор нуждается в явном указании режима многопоточности, то используется однопоточный режим.

Режим многопоточности является глобальным параметром. Он распространяется на все проекты сразу. Попытка двух проектов установить разные значения режима многопоточности приведет к порождению исключения.

```
Mxx_ru::Cpp::Dll_target.new( "threads_1/dll.rb" ) {
  target( "threads_1.4.0" )

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

Получить текущее значение режима многопоточности можно с помощью метода `mxx_threading_mode`.

Примечание. Для некоторых компиляторов, например, Visual C++, имена используемых rtl-библиотек выбираются исходя из типа rtl и режима многопоточности. И не для всех сочетаний возможных типов rtl и режима многопоточности компилятор может иметь библиотеки. Так, Visual C++ не имеет библиотек для однопоточной shared rtl.

5.7 Установка режима RTTI

В некоторых компиляторах режим RTTI (Run-Time Type Identification) по-умолчанию отключен (например, Visual C++) или может быть отключен (например, GNU C++). Если проект нуждается в RTTI, например, для безопасного использования *dynamic_cast*, то проект должен явно включить режим RTTI. Напротив, если проект не нуждается в RTTI и для проекта критично время исполнения и/или объем результирующего кода, то режим RTTI может быть отключен.

Для включения/выключения режима RTTI предназначен метод `rtti_mode`, который принимает в качестве значений константы `Mxx_ru::Cpp::RTTI_DEFAULT` (проекту все равно, используется ли RTTI или нет), `Mxx_ru::Cpp::RTTI_ENABLED` (проекту необходим режим RTTI) и `Mxx_ru::Cpp::RTTI_DISABLED` (проекту необходимо отсутствие RTTI).

Режим RTTI является глобальным параметром. Он распространяется на все проекты сразу. Попытка двух проектов установить разные значения режима RTTI приведет к порождению исключения.

```
1 Mxx_ru::Cpp::Dll_target.new( "threads_1/dll.rb" ) {
2   target( "threads_1.4.0" )
3
4   rtl_mode( Mxx_ru::Cpp::RTL_SHARED )
5   threading_mode( Mxx_ru::Cpp::THREADING_MULTI )
6   rtti_mode( Mxx_ru::Cpp::RTTI_ENABLED )
7   ...
8 }
```

Получить текущее значение режима RTTI можно с помощью метода `mxx_rtti_mode`.

5.8 Указание имени результирующего файла цели

Примечание. Для проектов, которые описываются с помощью `Mxx_ru::Cpp::Composite_target` нет результирующего файла, поэтому обращения к методам `target_root`, `target`, `implib_path` игнорируются.

5.8.1 Метод target_root

Метод `target_root` позволяет задать имя каталога, в который должен быть помещен результирущий файл (exe, dll или lib). Размещение результата контролируется с помощью obj_placement (см. 5.1.2 на стр. 27). Если не задан target_root, то результат будет помещен в каталог, который выберет obj_placement. Если же установлен target_root, то результат будет помещен в подкаталог с именем, указанным в `target_root`, в каталоге, который выберет obj_placement.

Например, пусть obj_placement располагает exe-файлы в текущий каталог. Тогда, если установить в target_root значение `out32`, то obj_placement поместит результирующий файл в подкаталог `out32` текущего каталога.

Важно. Желательно, чтобы метод `target_root` был вызван перед методом `target`. Тогда, при обращении к `target` имя результирующего файла будет автоматически создано с использованием значения target_root. Если же метод `target_root` вызывается

после `target`, то модифицируется уже назначенное имя результирующего файла. В общем случае, если `target_root` и `target` вызываются по одному разу, то результат получается один и тот же. Но, если `target_root` вызвать два раза (до и после обращения к `target`, то результат может отличаться от ожидаемого.

5.8.2 Метод target

Метод `target` позволяет задать базовое имя для результирующего файла. Реальное имя результата будет создано с учетом особенностей целевой платформы. Например, для dll с базовым именем threads_1.4.0 на платформе mswin будет создано реально имя threads_1.4.0.dll, а на платформе unix — libthreads_1.4.0.so.

Указываемое методу `target` имя не должно содержать имен каталогов. Если требуется дать указания относительно размещения результирующего файла, то следует использовать метод `target_root` **перед** обращением к `target`.

5.8.3 Метод implib_path

Метод `implib_path`, во-первых, предписывает Mxx_ru построить библиотеку импорта для dll (на тех платформах, на которых такое понятие актуально) и, во-вторых, определяет расположение библиотеки импорта.

В настоящий момент `implib_path` используется только для dll. Если `implib_path` не вызывается, то библиотека импорта не строится.

5.8.4 Примеры

Простое указание имени для exe-файла.

```
Mxx_ru::Cpp::Exe_target.new( "hello_world.rb" ) {
  target( "hello_world" )
  ...
}
```

Размещение exe-файла в подкаталоге `utest`.

```
Mxx_ru::Cpp::Exe_target.new( "packer/utest/prj.rb" ) {
  target_root( "utest" )
  target( "test_packer" )
  ...
```

На платформе mswin результирующая dll помещается в текущий каталог, а библиотека импорта в каталог lib. На платформе unix результирующая библиотека помещается в каталог lib.

```
Mxx_ru::Cpp::Dll_target.new( "oess_1/defs/prj.rb" ) {
  if "mswin" == toolset.tag( "target_os" )
    implib_path( "lib" )
  elsif "unix" == toolset.tag( "target_os" )
    target_root( "lib" )
  end

```

```
8   target( "oess_defs.1.3.0" )
9   ...
10  }
```

5.9 Установка типа приложения (консоль/GUI)

Метод `screen_mode` предназначен для указания типа приложения: консольное или оконное. По-умолчанию, Mxx_ru расчитывает на создание консольного приложения — компилятору и линкеру выставляются специальные опции. Вызвав метод `screen_mode` можно перевести проект в режим создания оконного (GUI) приложения.

В метод `screen_mode` необходимо передавать константу `Mxx_ru::Cpp::SCREEN_WINDOW` для оконного приложения или константу `Mxx_ru::Cpp::SCREEN_CONSOLE` для консольного приложения.

```
1  Mxx_ru::Cpp::Exe_target.new( "interface/prj.rb" ) {
2    ...
3    screen_mode( Mxx_ru::Cpp::SCREEN_WINDOW )
4    ...
5  }
```

Параметр screen_mode является локальным для проекта, он не распространяется ни на подчиненные проекты, ни на проекты, которые используют данный проект.

Получить текущий тип приложения можно посредством метода `mxx_screen_mode`.

5.10 Указание исходных файлов

5.10.1 Метод sources_root

По-умолчанию, Mxx_ru считает, что имена исходных файлов задаются относительно того каталога, в котором находится файл-проект. Т.е., если в качестве псевдонима проекта указано имя `interface/prj.rb`, то конструкция:

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

приведет к тому, что Mxx_ru будет искать файлы `interface/main.c` и `interface/io.cpp`. Такой подход позволяет легко перечислять в проектном файле содержимое небольшого проекта. Но может быть два случая, когда требуется указать Mxx_ru иное место для поиска исходных файлов:

1. Когда исходные тексты расположены в другой ветви файловой структуры проекта, нежели проектный файл. Например, все проектные файлы сложного проекта находятся в каталоге `mxxru_prjs`, а исходные файлы в подкаталоге `src`[3]. Тогда следует использовать метод `sources_root` с одним параметром:

[3] Такой вариант может оказаться удобным, если проект позволяет компиляцию различными инструментами.

```
Mxx_ru::Cpp::Exe_target.new( "mxxru_prjs/interface.rb" ) {
  ...
  sources_root( "src/interface" )

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

2. Когда файловая структура проекта имеет большую глубину. В этом случае неудобно для каждого исходного файла указывать полный путь к нему. В таких случаях можно воспользоваться вызовом метода `sources_root` с двумя параметрами: именем каталога и блоком кода. В этом случае метод `sources_root` работает очень просто: он конкатенирует текущее значение параметра sources_root с первым аргументом и получившийся результат устанавливается в качестве нового значения параметра sources_root. Затем метод `sources_root` запускает переданный ему блок кода, а по завершению блока кода восстанавливает значение sources_root в исходное. Причем такое поведение позволяет использовать вызовы `sources_root` внутри блока-параметра:

```
Mxx_ru::Cpp::Dll_target.new( "oess_1/db/prj.rb" ) {
  ...
  sources_root( "impl" ) {
    # Для файлов из каталога oess_1/db/impl.
    ...
    sources_root( "storage" ) {
      # Для файлов из каталога oess_1/db/impl/storage.
      ...
      sources_root( "impl" ) {
        # Для файлов из каталога oess_1/db/impl/storage/impl.
        ...
      }
    }
  }
  sources_root( "site" ) {
    # Для файлов из каталога oess_1/db/site.
    ...
    sources_root( "impl" ) {
      # Для файлов из каталога oess_1/db/site/impl.
      ...
    }
  }
  ...
}
```

Оба подхода можно комбинировать. Например, сначала вызвать `sources_root` с одним параметром, а затем использовать `sources_root` с двумя параметрами:

```
Mxx_ru::Cpp::Exe_target.new( "mxxru_prjs/interface.rb" ) {
  sources_root( "src/interface" )
```

```
3
4     # Эти файлы из каталога src/interface.
5     c_source( "main.c" )
6     cpp_source( "io.cpp" )
7
8     sources_root( "debug" ) {
9       # Эти файлы из каталога src/interface/debug.
10      cpp_sources( "io_dumper.cpp" )
11      ...
12    }
13    ...
14  }
```

5.10.2 Методы c_source, cpp_source

Методы `c_source` и `cpp_source` предназначены для указания имен C и C++ файлов, входящих в проект. Важно, чтобы файлы, которые содержат только C-код (как правило, имеющие расширение `.c`) перечислялись с помощью метода `c_source`, а файлы с C++-кодом (которые на разных платформах могут иметь расширения `.C`, `.cc`, `.cp`, `.cpp`, `.cxx`, `.c++` и т.д.) перечислялись с помощью метода `cpp_source`. Дело в том, что на некоторых платформах от типа файла зависит имя используемого компилятра (например, `gcc` и `g++` для GNU C++). На некоторых платформах есть возможность выставлять специальные ключи компилятору для того, чтобы компилятор воспринимал файл как C- или C++-код вне зависимости от расширения файла. Mxx_ru учитывает все эти особенности и активно их использует.

Методы `c_source`, `cpp_source` получают два аргумента: первый, обязательный, указывает имя исходного файла. Второй, не обязательный, может содержать опции компилятора, которые будут добавлены к выбранным Mxx_ru опциям при компиляции только этого файла.

Как правило, используется вариант с одним параметром:

```
1  Mxx_ru::setup_target(
2    Mxx_ru::Cpp::Dll_target.new( "oess_1/io/prj.rb" ) {
3
4      required_prj( "oess_1/defs/prj.rb" )
5
6      Oess_1::setup_platform( self )
7
8      target( "oess_io" + Oess_1::VERSION )
9
10     define( "OESS_1__IO__PRJ" )
11
12     cpp_source( "stream.cpp" )
13     cpp_source( "binstream.cpp" )
14     cpp_source( "binbuffer.cpp" )
15     cpp_source( "subbinstream.cpp" )
16     cpp_source( "mem_buf.cpp" )
17     cpp_source( "fixed_mem_buf.cpp" )
18     cpp_source( "bstring_buf.cpp" )
19   }
```

```
20 )
```

Вариант с двумя параметрами может использоваться, если для какого-то файла требуется установить специфические опции компилятора. Например, отдельный символ препроцессора:

```
1 ...
2 cpp_source( "interface.cpp", [ "-DLOG_LEVEL=3" ] )
3 cpp_source( "engine.cpp" )
4 ...
```

в этом случае символ препроцессора `LOG_LEVEL` будет определен только при компиляции `interface.cpp`, но не будет определен для `engine.cpp`.

Применять методы `c_source`, `cpp_source` следует с осторожностью, учитывая следующие факторы:

- если проект может быть скомпилирован несколькими компиляторами на разных платформах, то необходимо учитывать, что одни и те же параметры должны будут задаваться разными опциями для разных компиляторов. Т.е. в проекте придется формировать набор опций для каждого из компиляторов;
- Mxx_ru не отслеживает переданных в `c/cpp_source` параметров и не проверяет, насколько они соответствуют тем параметрам, которые выбирает сам Mxx_ru. Например, если Mxx_ru считает, что для многопоточного приложения с разделяемой runtime-библиотекой для Visual C++ нужно выставить опцию `-MD`, а для конкретного файла будет выставлена опция `-MT` (многопоточная статическая runtime-библиотека), то Mxx_ru не сможет диагностировать этот конфликт;
- в предыдущих версиях Make++ не было возможности задавать специфических для отдельного исходного файла опций компилятра. Поэтому такая поддержка в первой версии Mxx_ru является экспериментом. Может быть, его результат приведет к тому, что специфические для отдельного файла опции будут задаваться по-другому. Например, с указанием типа опции (определение символа препроцессора, установка уровня оптимизации, установка величины выравнивания данных и т.д.). Поэтому следует учитывать, что при переходе на новые версии Mxx_ru может потребоваться переписать проектные файлы, в которых используются вызовы методов `c_source`, `cpp_source` с двумя аргументами.

5.10.3 Метод mswin_rc_file

Метод `mswin_rc_file` позволяет установить имя ресурсного файла на платформе mswin. Если имя ресурсного файла установлено, то при компиляции проекта Mxx_ru запускает компилятор ресурсов для получения `.res` файла, а при линковке результирующего exe- или dll-файла использует соответствующее средства toolset-а для включения скомпилированных ресурсов в результирующий файл.

Метод `mswin_rc_file` получает два параметра: первый, обязательный, указывает имя `.rc`-файла. Второй, необязательный, может содержать имена файлов, от которых зависит ресурсный файл. Например, имена файлов `.ico`, `.bmp` и т.д.

При использовании метода `mswin_rc_file` необходимо помнить, что значение первого параметра (имени `.rc`-файла) вычисляется с учетом текущего значения sources_root. Но имена зависимостей, которые передаются во втором параметре, должны быть указаны полностью.

Пример:

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

Примечание. Значение, установленное методом `mswin_rc_file` учитывается только, если целевой платформой является mswin.

5.11 Указание дополнительных объектных файлов

Метод `obj_file` позволяет указать проекту дополнительные объектные файлы для линковки результирующего файла. По-умолчанию, Mxx_ru формирует имена объектных файлов, получаемых в результате компиляции исходных файлов, и использует их имена при линковании. Если проекту требуется включить в проект уже готовый объектный файл (например, получаемый путем компиляции исходного файла на другом языке программирования), то имя этого объектного файла нужно включить с помощью метода `obj_file`. Например:

```
Mxx_ru::Cpp::Exe_target.new( "interface/prj.rb" ) {
  ...
  required_prj( "interface/asm/fast_transform/prj.rb" )
  ...
  obj_file( "interface/asm/fast_transform/linear.obj" )
  ...
}
```

Сам Mxx_ru использует метод `obj_file` в реализации toolset для сохранения в проекте имен объектных файлов, получаемых в результате компиляции исходных файлов.

5.12 Указание дополнительных библиотек

Метод `lib` позволяет указать проекту дополнительные библиотеки для линкования результирующего файла. По-умолчанию, Mxx_ru извлекает имена необходимых для линкования библиотек из подпроектов. Но часто бывает необходимо указать проекту имя конкретной, как правило, платформо-зависимой библиотеки. В этом случае следует использовать метод `lib`:

```
Mxx_ru::Cpp::Dll_target.new( "engine/prj.rb" ) {
  ...
  if "mswin" == toolset.tag( "target_os" )
    lib( "winsock.lib" )
  elsif "unix" == toolset.tag( "target_os" )
    lib( "socket" )
    if "bsd" != toolset.tag( "unix_port" )
      lib( "pthread" )
    end
  end
  ...
}
```

Метод `lib` получает два аргумента. Первый задает имя библиотеки в том виде, в каком это имя следует указать линкеру в командной строке. Т.е., на платформе mswin это должно быть имя с расширением. А на платформах unix — имя без расширения и префикса `lib`.

Второй аргумент метода `lib` задает путь, в котором линкеру следует искать библиотеку. По-умолчанию, этот аргумент равен `nil`, т.е. линкеру не дается никаких указаний. В этом случае линкер будет искать библиотеку в стандартных для конкретной платформы путях. Если же второй параметр отличен он `nil`, то указанное значение будет передано линкеру в качестве пути для поиска библиотек. Например, если на платформе unix требуется указать линкеру использовать библиотеку `/usr/local/share/mysec/lib/libtdes.3.4.a`, то следует вызвать метод `lib` следующим образом:

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

5.13 Указание режима оптимизации

Метод `optimization` позволяет установить необходимый режим оптимизации. Mxx_ru учитывает режим оптимизации при формировании опций компилятора в режиме release (см. 5.1.5 на стр. 31). По-умолчанию, используется оптимизации по скорости выполнения сгенерированного кода.

Режим оптимизации задается константами: `Mxx_ru::Cpp::OPTIM_SIZE` (оптимизация по размеру генерируемого кода) и `Mxx_ru::Cpp::OPTIM_SPEED` (оптимизации по скорости генерируемого кода).

Режим оптимизации является локальным для проекта параметром. Установленное в проекте значение режима оптимизации не оказывает никакого воздействия ни на подчиненные проекты, ни на проекты, которые используют в качестве подчиненного данный проект.

5.14 Функции для работы с локальными, распространяемыми и глобальными параметрами

Как показано в 5.1.6 на стр. 31, для установки локальных, распространяемых и глобальных параметров проекта используются функции вида:

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

где `<параметр>` — это название параметра. Функция первого вида используется для установки локального или распространяемого параметра. Значение параметра будет локальным, если параметр `a_type` будет иметь значение `Mxx_ru::Cpp::Target::OPT_LOCAL`, либо будет опущен. Если параметр `a_type` будет иметь значение `Mxx_ru::Cpp::Target::OPT_UPSPREAD`, то значение параметра будет автоматически распространяться и на все использующие данный проект проекты. Функция с префиксом `global_` используется для установки глобального значения параметра.

В данном разделе описываются функции, которые позволяют задавать локальные, распространяемые и глобальные значения параметров проекта.

5.14.1 include_path, global_include_path

Функции `include_path`, `global_include_path` позволяют установить пути, в которых компилятор будет искать заголовочные файлы.

5.14.2 define, global_define

Функции `define`, `global_define` позволяют установить символы препроцессора, которые будут определены при компиляции исходных C- и C++-файлов.

5.14.3 compiler_option, global_compiler_option

Функции `compiler_option, global_compiler_option` позволяют задать дополнительные опции компилятора, которые будут использоваться как для компиляции C, так и для компиляции C++-файлов.

5.14.4 c_compiler_option, global_c_compiler_option

Функции `c_compiler_option, global_c_compiler_option` позволяют задать дополнительные опции компилятора, которые будут использоваться для компиляции только C-файлов.

5.14.5 cpp_compiler_option, global_cpp_compiler_option

Функции `cpp_compiler_option, global_cpp_compiler_option` позволяют задать дополнительные опции компилятора, которые будут использоваться для компиляции только C++-файлов.

5.14.6 linker_option, global_linker_option

Функции `linker_option, global_linker_option` позволяют задать дополнительные опции линкера, которые будут использоваться для линковки exe- и dll-файлов.

5.14.7 librarian_option, global_librarian_option

Функции `librarian_option, global_librarian_option` позволяют задать дополнительные опции библиотекаря, которые будут использоваться для создания статических библиотек.

5.14.8 Компилятор ресурсов на платформе mswin

mswin_rc_include_path, global_mswin_rc_include_path

Функции `mswin_rc_include_path`, `global_mswin_rc_include_path` позволяют задать пути поиска заголовочных файлов для компилятора ресурсов.

mswin_rc_define, global_mswin_rc_define

Функции `mswin_rc_define`, `global_mswin_rc_define` позволяют задать символы препроцессора, которые будут определены при компиляции ресурсов.

mswin_rc_option, global_mswin_rc_option

Функции `mswin_rc_option`, `global_mswin_rc_option` позволяют задать дополнительные опции компилятору ресурсов.

mswin_rlink_option, global_mswin_rlink_option

Функции `mswin_rlink_option`, `global_mswin_rlink_option` позволяют задать дополнительные опции линкеру ресурсов.

Глава 6

Дополнительные возможности и особенности

6.1 Аргумент mxx-show-cmd

По-умолчанию, Mxx_ru в процессе своей работы не отображает информации о своих действиях — работает по принципу черного ящика. В больших проектах или при отладке проектных файлах может быть интересно или полезно знать, какие именно команды запускаются. Если передать интерпретатору Ruby аргумент `--mxx-show-cmd`, то на стандартный поток вывода будут отображаться названия запускаемых команд и их параметры. Например, при компиляции примера из 3.4 на стр. 13 компилятором GNU C++ под Cygwin, аргумент `--mxx-show-cmd` приведет к следующей печати:

```
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  >>>
```

6.2 Аргумент mxx-keep-tmps

Для формирования командных строк некоторых инструментов на некоторых платформах требуется создание т.н. *response*-файлов. Это временные текстовые файлы, которые содержат значения параметров для запускаемого инструмента. В этих случаях Mxx_ru создает временные файлы с уникальными именами, осуществляет запуск необходимых инструментов и затем, после окончания обработки проектного файла, уничтожает созданные временные файлы. Например, компиляция примера из 3.4 на стр. 13 компилятром Visual C++ приводит к запуску следующих команд:

```
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 >>>
```

Здесь файлы с именами `tmpmxx_ru.[0-9]+.[0-9]+` — это временные файлы, созданные Mxx_ru.

Иногда, при отладке проектов или новых toolset бывает полезно сохранить временные файлы для последующего анализа. Достигается это указанием интерпретатору Ruby аргумента `--mxx-keep-tmps`:

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

6.3 Аргумент mxx-show-tmps

Аргумент `--mxx-show-tmps` является аналогом аргумента `--mxx-show-cmd`, но для временных response-файлов. Если этот аргумент передан интрепритатору Ruby, то Mxx_ru после формирования очередного временного файла отображает на стандартный поток вывода содержимое этого файла. Так, компиляция примера из 3.4 на стр. 13 компилятором Visual C++ с аргументами `--mxx-show-cmd`, `--mxx-show-tmps` приведет к следующему результату:

```
<<<[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 >>>
```

6.4 Аргумент mxx-dry-run

Если интерпретатору Ruby передан аргумент `--mxx-dry-run`, то Mxx_ru выполняет имитацию построения проекта. Т.е. обрабатываются все проекты, формируются командные строки для запуска всех инструментов, включая создание response-файлов, но сами инструменты не запускаются.

Этот аргумент удобно использовать для отладки проектных файлов и новых toolset в сочетании с аргументами `--mxx-show-cmd`, `--mxx-show-tmps`, `--mxx-keep-tmps`.

6.5 Исключения

Язык Ruby является объектно-ориентированным языком, в котором широко используется механизм исключений. Например, когда интерпретатор Ruby встречает в скрипте синтаксическую ошибку, то генерируется исключение. Сам Mxx_ru так же использует исключения для информирования об обнаруженных им ошибках.

Особенностью использования Mxx_ru является то, что в выбранной архитектуре проектных файлов не все иключения можно перехватить средствами Mxx_ru. В результате, например, из-за синтаксической ошибке в проектном файле, интерпретатор может выдать подробный 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
```

Такая подробность может первое время шокировать. Но со временем это перестает быть проблемой, во-первых, потому, что с опытом уменьшается количество синтаксических ошибок и, во-вторых, это позволяет легко диагностировать проблемы в самом Mxx_ru.

Если же Mxx_ru в состоянии перехватить исключение, то Mxx_ru делает это и отображает на стандартный поток ошибок только описание перехваченного исключения. Например:

```
<<< 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'>>>
```

6.6 Подключение в проект make-правил

Основная идея Mxx_ru состоит в том, чтобы упростить наиболее типовые действия при компиляции проектов. Поэтому Mxx_ru предоставляет набор готовых шаблонов, в которые необходимо лишь подставить собственные данные. Однако, бывают случаи когда возможностей шаблонов не хватает. Например, в текущей версии Mxx_ru нет поддержки средств локализации Qt-приложений. Если в Qt-проекте потребуется, скажем, сгенерировать .qm-файл из готового .ts-файла, то окажется, что в существующих шаблонах Mxx_ru нет инструмента для выполнения именно этого действия.

Хорошим выходом из этой ситуации стало бы создание для Mxx_ru необходимой возможности с тем, чтобы ее можно было использовать затем и в других Qt-проектах. Но очевидно, что не каждый использующий Mxx_ru разработчик захочет дорабатывать Mxx_ru. В таких случаях остается только воспользоваться классом генератора `Mxx_ru::Makestyle_generator`, который позволяет подключить в проект произвольное make-правило:

```
1 ...
2 require 'mxx_ru/makestyle_generator'
3 ...
4 generator(
5   Mxx_ru::Makestyle_generator.new(
6     "etc/wms_ctl_1.ru.qm", "etc/wms_ctl_1.ru.ts",
7     "lrelease etc/wms_ctl_1.ru.ts -qm etc/wms_ctl_1.ru.qm" ) )
```

В данном примере создается make-правило по генерации файла `etc/wms_ctl_1.ru.qm` (цель make-правила) из файла `etc/wms_ctl_1.ru.ts` (зависимость для цели make-правила) при помощи команды `lrelease` из состава Qt. Когда запускается построение проекта, содержащего данное правило, генератор `Mxx_ru::Makestyle_generator` проверяет существование файла .qm. Если его нет, или он изменялся раньше, чем файл .ts, то запускается `lrelease`. Т.е. выполняется логика обычного make-правила в обычном make. При выполнении операции clean (очистки проекта) файл `etc/wms_ctl_1.ru.qm` уничтожается.

Однако, в отличии от обычных make-файлов, заданные с помощью генератора `Mxx_ru::Makestyle_generator` правила выполняются до основных действий проектного файла. И, кроме того, выполняются строго в том порядке, в котором были созданы генераторы. Поэтому следующий пример:

```
1 generator(
2   Mxx_ru::Makestyle_generator.new( "c", "b", "make_c" ) )
3 generator(
4   Mxx_ru::Makestyle_generator.new( "b", "a", "make_b" ) )
```

не будет работать при отсутствии файла `b`, т.к. первый генератор просто не будет знать, что файл `b` строится по правилам второго генератора. И это именно то поведение, которое задумывалось при реализации `Mxx_ru::Makestyle_generator` — Mxx_ru не является традиционным make, и все сложные операции должны выполняться посредством специально созданных для этих операций инструментов Mxx_ru. А генератор

`Mxx_ru::Makestyle_generator` выступает лишь в качестве спасательного круга когда таких инструментов еще нет.

Конструктор класса `Mxx_ru::Makestyle_generator` получает четыре аргумента (последний аргумент опциональный):

a_target_files имена файлов-целей make-правила.

a_dependencies имена файлов зависимостей для файлов-целей.

a_build_cmds список команд для построения файлов-целей.

a_clean_cmds список команд для удаления файлов-целей при выполнении операции clean. Если этот аргумент не указан, то файлы-цели удаляются автоматически.

Если аргументу конструктора `Mxx_ru::Makestyle_generator` требуется передать всего одно значение, то его можно передать в виде строки (как было показано выше). Если же аргументу требуется передать несколько значений, то их нужно указывать как вектор:

```
generator(
  Mxx_ru::Makestyle_generator.new(
    # Имена файлов целей.
    [ "a", "b", "c" ],
    # Единственная зависимость для всех целей.
    "d",
    # Команды для построения целей.
    [
      "make_a",
      "make_b_c"
    ],
    # Команды для удаления целей.
    [
      "destroy_a",
      "destroy_b",
      "destroy_c"
    ]
  ) )
```

При выполнении построения файлов-целей генератор `Mxx_ru::Makestyle_generator` поочередно запускает команды из аргумента `a_build_cmds`. Если очередная команда возвращает ненулевой код возврата, то построение целей прерывается исключением. При выполнении операции clean коды возвратов команд из `a_clean_cmds` игнорируются.

Глава 7

Поддержка unit-тестинга

7.1 Unit-тестинг для исполняемых двоичных приложений

7.1.1 Определение unit-test приложения

Unit-test исполняемым двоичным приложением называется приложение, в котором находится код unit-тестов и которое необходимо запустить для прохождения unit-тестов. Успешность прохождения unit-тестов определяется по коду завершения приложения. Если приложение возвращает 0, то unit-тесты считаются успешно пройденными.

7.1.2 Идея

Для поддержки unit-test необходимо создать два проектных файла: один из них отвечает за построение unit-test приложения, второй отвечает за его запуск и анализ кода возврата.

Предполагается, что для проекта существует композитный проектный файл, в котором указываются все проекты. Среди этих проектов так же указываются проекты unit-test-ов. В результате получается, что в процессе построения композита автоматически запускается построение unit-test приложения, после чего unit-test запускается. Если unit-test отрабатывает нормально, то процесс построения композита продолжается. В противном случае построение композита завершается[1].

7.1.3 Класс для цели unit-test

В состав Mxx_ru входит класс `Mxx_ru::Binary_unittest_target`, экземпляр которого нужно создать для того, чтобы включить unit-test в процесс построения проекта. Для использования этого класса необходимо включить файл `mxx_ru/binary_unittest`:

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

[1] Именно этим в Mxx_ru unit-test отличается от regression-test: для успешного построения проекта необходимо чтобы успешно отрабатывали все unit-test-ы.

Данный класс получает все параметры в своем конструкторе: собственный псевдоним и имя проекта, который отвечает за построение unit-test приложения.

7.1.4 Пример

Проектный файл unit-test приложения:

```
require 'mxx_ru/cpp'

Mxx_ru::setup_target(
  Mxx_ru::Cpp::Exe_target.new( "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" )
  }
)

```

Проектный файл для unit-test:

```
require 'mxx_ru/binary_unittest'

Mxx_ru::setup_target(
  Mxx_ru::Binary_unittest_target.new(
    "test/active_group/prj.ut.rb",
    "test/active_group/prj.rb" )
)
```

Общий композитный проектный файл:

```
require 'mxx_ru/cpp'

Mxx_ru::setup_target(
  Mxx_ru::Cpp::Composite_target.new( Mxx_ru::BUILD_ROOT ) {
    global_include_path( "." )

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

Запускается композитный проект как обычно:

```
ruby build.rb
```

Но в процессе построения проекта будет выдано сообщение:

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

Если же unit-тесты не будут пройдены успешно, то результат построения проекта может выглядеть, например, так:

```
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'>>>
```

7.2 Unit-тестинг в виде сравнения текстовых файлов

В некоторых случая сложно создать двоичное unit-test приложение которое полностью контролирует прохождение тестов и возвращает нулевой код возврата в случае успеха (именно на такие unit-test приложения расчитан класс `Mxx_ru::Binary_unittest_target`, описанный в 7.1 на стр. 52). Например, таким способом сложно тестировать библиотеки разбора каких-либо входных файлов. В таких случаях удобно создать одно тестовое приложение, которое в командной строке получает имя входного файла в качестве аргумента и помещает результат своей работы в выходной файл. Тогда для каждого из входных файлов можно подготовить *эталонный* файл результата. И тестирование библиотеки будет заключаться в последовательном запуске тестового приложения с каждым из подготовленных тестовых входных данных с последующим сравнением произведенных выходных файлов с эталонными файлами.

Такой сценарий тестирования можно применять не только в случаях библиотек разбора чего-либо. Но и в случаях, когда результат работы тестируемого кода зависит от входящих данных. И при этом желательно, чтобы была простая процедура добавления и/или замены входных данных, без изменения исходного кода unit-test приложения. Например, для библиотек, выполняющих какие-либо расчеты, шифрование/дешифрование, вычисление/проверку криптографической подписи и т.д. Такое тестирование можно применять даже в случае, когда корректность работы кода проверяется по отладочным печатям, которые тестируемый код осуществляет в log-файл (например, так можно проверять, правильные ли ветви вычислений выбираются в том или ином случае).

Для поддержки такого типа тестирования Mxx_ru предоставляет класс `Mxx_ru::Textfile_unittest_target`. Идея его использования аналогична идеи использования класса `Mxx_ru::Binary_unittest_target`[2]: необходимо создать два проектных файла, первый из которых будет управлять построением unit-test приложения, а второй — запуском тестового приложения с различными параметрами и сравнением результатов работы для каждого тестового случая.

7.2.1 Класс Mxx_ru::Textfile_unittest_target

Класс `Mxx_ru::Textfile_unitest_target` предназначен для описания целей, которые запускают одно unit-test приложение с разными параметрами и сравнивают результаты работы после каждого запуска с эталонными результатами. Для использования этого класса необходимо подключить в проект файл `mxx_ru/textfile_unittest`:

```
require 'mxx_ru/textfile_unittest'

Mxx_ru::setup_target(
  Mxx_ru::Textfile_unittest_target.new(
    "some/project/prj.ut.rb",
    "some/project/prj.rb" ) {

    # Описание последовательности запуска тестов.
    ...
  }
)
```

Каждый запуск тестового приложения описывается с помощь метода `launch`. Первым параметром этого метода является строка с параметрами для тестового приложения. Вторым аргументом является вектор описаний подлежащих сравнению файлов. Т.е. допускаются случаи, когда приложение производит более одного выходного файла.

В своем методе `build` класс `Mxx_ru::Textfile_unittest_target` сначала выполняет построение тестового приложения. Если тестовое приложение построено успешно, то последовательно выполняются все описанные запуски тестового приложения. Если очередной запуск прошел успешно, т.е. приложение возвратило нулевой код, то производится последовательное сравнение всех описанных пар файлов. Если все пары файлов совпадают, то выполняется следующий запуск приложения и т.д.

7.2.2 Пример

Проектный файл для unit-test:

```
require 'mxx_ru/textfile_unittest'

Mxx_ru::setup_target(
  Mxx_ru::Textfile_unittest_target.new(
    "prj.ut.rb",
    "prj.rb" ) {

```

[2] См. 7.1.2 на стр. 52

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

Результат запуска unit-test:

```
$ 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'
```

Если при сравнении файлов обнаруживаются несовпадения, то результат запуска unit-test может выглядеть, например, так:

```
$ 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
'''>>>
```

7.2.3 Особенности

Класс `Mxx_ru::Textfile_unittest_target` выполняет простое построчное сравнение файлов с учетом регистра символов. Сравнение прерывается при обнаружении первого несовпадения.

В методе `clean` класс `Mxx_ru::Textfile_unittest_target` удалят файлы, которые указаны как выходные файлы (т.е. файлы, имена которых задаются первым параметром при обращении к методу `pair`, удаляются при указании интерпретатору Ruby агрумента `--mxx-clean`).

Глава 8

Генератор для Qt

8.1 Введение

В данной главе разсказывается о генераторе C++ файлов, необходимых при использовании библиотеки Qt `http://www.trolltech.com`. Описываемый генератор был протестирован на Qt версии 3.3.3.

При использовании Qt в трех случаях возникает необходимость генерирования C++ файлов с помощью входящих в состав Qt инструментов:

1. Когда в заголовочном файле описывается класс, производный от QObject и в этом классе определены слоты и/или сигналы. В этом случае из заголовочного файла необходимо создать исходный файл, в котором будет находится реализация слотов и сигналов. Для генерации используется утилита *moc*. Обычно из файла `a.h` генерируется файл `moc_a.cpp` (т.е. расширение заменяется на `.cpp`, а к имени добавляется префикс `moc_`).

2. Когда производный от QObject класс описывается в исходном файле. В этом случае должен быть создан дополнительный исходный файл, который подключается в основной исходный файл посредством директивы `#include`:

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

   Для генерации используется утилита *moc*. Обычно из файла `a.cpp` генерируется файл `a.moc` (т.е. расширение заменяется на `.moc`).

3. Из файла `.ui`, полученного с помощью Qt Designer, необходимо получить заголовочный и исходный файлы. Для генерации используется утилита *uic*. Из полученного заголовочного файла с помощью утилиты *moc* нужно так же получить еще один исходный файл с реализацией слотов и сигналов. Обычно из файла `a.ui` строятся файлы `a.hpp`, `a.cpp`, `moc_a.cpp`.

Генератор для Qt, который входит в состав Mxx_ru, позволяет обрабатывать перечисленные выше случаи.

8.2 Использование генератора для Qt

8.2.1 Подключение необходимых описаний к проектному файлу

Описание генератора для Qt содержится в файле `mxx_ru/cpp/qt.rb`, который подключается в проектный файл посредством `require`:

```
require 'mxx_ru/cpp/qt'

Mxx_ru::setup_target( ... )
```

8.2.2 Создание генератора для Qt

Генератор для Qt реализован в виде класса `Mxx_ru::Cpp::Qt_gen`. Чтобы создать генератор для Qt необходимо создать в проекте объект этого класса:

```
require 'mxx_ru/cpp/qt'

Mxx_ru::setup_target(
  Mxx_ru::Cpp::Exe_target.new( "some/project/prj.rb" ) {
    ...
    qt = generator( Mxx_ru::Cpp::Qt_gen.new( self ) )
    ...
  }
)
```

Конструктор класса `Qt_gen` получает два аргумента: ссылку на объект-цель, в которой будет использован генератор, и набор символов препроцессора, которые будут установлены при компиляции проекта. По-умолчанию второй аргумент равен `['QT_DLL', 'QT_THREAD_SUPPORT']`. Если используется вариант Qt в виде dll, с поддержкой многопоточности, то второй аргумент в конструктор `Qt_gen` можно не указывать. В остальных случаях следует передать во втором аргументе необходимые для используемой версии Qt символы препроцессора. Например, если необходим вариант Qt в виде статической библиотеки, но с поддержкой многопоточности, то следует указать:

```
require 'mxx_ru/cpp/qt'

Mxx_ru::setup_target(
  Mxx_ru::Cpp::Exe_target.new( "some/project/prj.rb" ) {
    ...
    qt = generator( Mxx_ru::Cpp::Qt_gen.new( self,
      [ 'QT_THREAD_SUPPORT' ] ) )
    ...
  }
)
```

8.2.3 Указание заголовочных файлов для генерации исходных файлов

Заголовочные файлы, из которых нужно сгенерировать исходные файлы посредством утилиты moc указываются генератору с помощью функции `h2moc`:

```
require 'mxx_ru/cpp/qt'

Mxx_ru::setup_target(
  Mxx_ru::Cpp::Exe_target.new( "some/project/prj.rb" ) {
    ...
    qt = generator( Mxx_ru::Cpp::Qt_gen.new( self ) )
    ...
    qt.h2moc( "mywidget.hpp" )
  }
)
```

Указанное имя ищется относительно текущего значения sources_root (см. 5.10.1 на стр. 39). В приведенном пример файл `mywidget.hpp` должен быть расположен в каталоге `some/project`. В этот же каталог будет помещен результат работы утилиты moc[1].

Из перечисленных с помощью метода `h2moc` файлов строятся файлы, в имя которого добавляется префикс `moc_`, а расширение заменяется на `.cpp` (если не изменено с помощью атрибута `cpp_ext`, см. 8.2.7 на стр. 61). Так из `some/project/mywidget.hpp` будет получен файл `some/project/moc_mywidget.cpp`.

8.2.4 Указание исходных файлов для генерации исходных файлов

Исходные (`.cpp`) файлы, из которых нужно сгенерировать исходные (`.moc`) файлы посредством утилиты moc указываются генератору с помощью функции `cpp2moc`:

```
require 'mxx_ru/cpp/qt'

Mxx_ru::setup_target(
  Mxx_ru::Cpp::Exe_target.new( "some/project/prj.rb" ) {
    ...
    qt = generator( Mxx_ru::Cpp::Qt_gen.new( self ) )
    ...
    qt.cpp2moc( "mywidget.cpp" )
  }
)
```

Указанное имя ищется относительно текущего значения sources_root (см. 5.10.1 на стр. 39). В приведенном пример файл `mywidget.cpp` должен быть расположен в каталоге `some/project`. В этот же каталог будет помещен результат работы утилиты moc[2].

Из перечисленных с помощью метода `cpp2moc` файлов строятся файлы, у которых расширение заменяется на `.moc` (если не изменено с помощью атрибута `moc_ext`, см. 8.2.9 на стр. 62). Так из `some/project/mywidget.cpp` будет получен файл `some/project/mywidget.moc`.

[1] С учетом значения moc_result_subdir (см. 8.2.6 на стр. 61)
[2] С учетом значения moc_result_subdir (см. 8.2.6 на стр. 61)

8.2.5 Указание .ui-файлов

Результаты работы Qt Designer, сохраненные в виде `.ui-файлов`, указываются генератору с помощью метода `ui`:

```
require 'mxx_ru/cpp/qt'

Mxx_ru::setup_target(
  Mxx_ru::Cpp::Exe_target.new( "some/project/prj.rb" ) {
    ...
    qt = generator( Mxx_ru::Cpp::Qt_gen.new( self ) )
    ...
    qt.ui( "myform.ui" )
  }
)
```

Указанное имя ищется относительно текущего значения sources_root (см. 5.10.1 на стр. 39). В приведенном пример файл `myform.ui` должен быть расположен в каталоге `some/project`. В этот же каталог будет помещен результат работы утилиты ui.

Из перечисленных с помощью метода `ui` файлов строятся файлы заголовочные файлы, у которых расширение заменяется на `.hpp` (если не изменено с помощью атрибута `hpp_ext`, см. 8.2.8 на стр. 62), и исходные файлы, у которых расширение заменяется на `.cpp` (если не изменено с помощью атрибута `cpp_ext`, см. 8.2.7 на стр. 61). Имена сгенерированных заголовочных файлов затем указываются генератору через метод `h2moc` для генерации вспомогательного исходного файла. В результате из `some/project/myform.ui` будут получены файлы `some/project/myform.hpp`, `some/project/myform.cpp`, `some/project/moc_myform.cpp`.

8.2.6 Расположение результатов работы утилиты moc

По-умолчанию, результаты работы утилиты moc помещаются в тот же каталог, в котором находились исходные файлы для утилиты moc. В некоторых случаях это не очень удобно. Например, если требуется перейти на новую версию Qt удалив при этом старые результаты работы moc. Проще это сделать, если размещать сгенерированные файлы в отдельный подкаталог.

В классе `Mxx_ru::Cpp::Qt_gen` есть атрибут `moc_result_subdir`, который определяет размещение разультатов работы moc. Если этот атрибут содержит `nil` (значение по-умолчанию), то сгенерированные файлы помещаются рядом с исходными файлами. Если назначить атрибуту `moc_result_subdir` значение, то в качестве результирующего будет использоваться указанный подкаталог исходного каталога. Например:

```
qt = generator( Mxx_ru::Cpp::Qt_gen.new( self ) )
qt.moc_result_subdir = "moc"
qt.h2moc( "h/mywidget.hpp" )
```

Из файла `h/mywidget.hpp` будет сгенерирован файл `h/moc/moc_mywidget.cpp`.

Значение атрибута `moc_result_subdir` оказывает влияние только на файлы, которые перечисляются с помощью методов `h2moc` и `cpp2moc`.

8.2.7 Изменение расширения для сгенерированных исходных файлов

По-умолчанию, при генерации исходных файлов с помощью moc и ui для результирующих файлов используется расширение `.cpp`. Изменить его можно назначив новое значение атрибуту `cpp_ext` объекта-генератора:

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

Из файла `h/mywidget.H` будет сгенерирован файл `h/mywidget.CC`.

8.2.8 Изменение расширения для сгенерированных заголовочных файлов

По-умолчанию, при генерации заголовочных файлов с помощью ui для результирующих файлов используется расширение `.hpp`. Изменить его можно назначив новое значение атрибуту `hpp_ext` объекта-генератора:

```
1 qt = generator( Mxx_ru::Cpp::Qt_gen.new( self ) )
2 qt.hpp_ext = ".h";
3 qt.ui( "myform.ui" )
```

Из файла `myform.ui` будет сгенерирован файл `myform.h`.

8.2.9 Изменение расширения для сгенерированных moc-файлов

По-умолчанию, при генерации исходных файлов с помощью moc для результирующих файлов используется расширение `.mic`. Изменить его можно назначив новое значение атрибуту `moc_ext` объекта-генератора:

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

Из файла `mywidget.cpp` будет сгенерирован файл `mywidget.cpp.moc`.

8.2.10 Изменение имени утилиты moc

Имя утилиты moc находится в атрибуте `moc_name` объекта-генератора. По-умолчанию это `moc`. Изменение данного атрибута позволяет указать другое имя, которое будет использоваться для запуска утилиты moc:

```
1 qt = generator( Mxx_ru::Cpp::Qt_gen.new( self ) )
2 qt.moc_name = "/usr/src/qt.4.0.b2/bin/moc"
```

8.2.11 Изменение имени утилиты uic

Имя утилиты uic находится в атрибуте `uic_name` объекта-генератора. По-умолчанию это `uic`. Изменение данного атрибута позволяет указать другое имя, которое будет использоваться для запуска утилиты uic:

```
1 qt = generator( Mxx_ru::Cpp::Qt_gen.new( self ) )
2 qt.uic_name = "/usr/src/qt.4.0.b2/bin/uic"
```