ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Д. В. ДУБРОВ

СИСТЕМА ПОСТРОЕНИЯ ПРОЕКТОВ СМАКЕ

Учебник

Ростов-на-Дону
Издательство Южного федерального университета
2015

УДК 681.3
ББК 32.85
Д79

Печатается по решению Редакционно-издательского совета Южного федерального университета (протокол № 3 от 23 ноября 2015 г.)

Рецензенты:

кандидат физико-математических наук, доцент кафедры алгебры и дискретной математики мехмата ЮФУ М. Э. Абрамян;

кандидат физико-математических наук, доцент кафедры программного обеспечения вычислительной техники и автоматизированных систем факультета информатики и вычислительной техники ДГТУ В. А. Стукопин

Дубров, Д. В.

Д79 Система построения проектов CMake : учебник / Д. В. Дубров ; Южный федеральный университет. — Ростов-на-Дону : Издательство Южного федерального университета, 2015. — 419 с.

ISBN 978-5-9275-1852-4

Работа посвящена инструменту CMake, который является современной системой для описания программных проектов и обладает богатыми возможностями. В учебнике изложен материал, достаточный для создания при помощи CMake проектов со сложной структурой, использующих внешние библиотеки или вспомогательные инструменты разработчика.

Учебник предназначен для магистрантов Института математики, механики и компьютерных наук им. И. И. Воровича Южного федерального университета по направлению подготовки «Фундаментальная информатика и информационные технологии», изучающих курс «Разработка кроссплатформенных приложений». Также учебник может быть полезен всем студентам, аспирантам и специалистам, которые участвуют в разработке сложных программных проектов.

ISBN 978-5-9275-1852-4 УДК 681.3

ББК 32.85

Оглавление

Введение

Настоящую работу можно рассматривать как вводное руководство по системе построения проектов CMake[1]. Учебник предназначен для студентов мехмата ЮФУ, изучающих курс «Разработка кроссплатформенных приложений», а также может быть полезным разработчикам программного обеспечения, планирующим внедрение данного инструмента в своей работе и приступающим к его освоению. Использование CMake может упростить выполнение курсовых или выпускных студенческих работ, связанных с использованием свободно распространяемых библиотек. Несмотря на высокую популярность системы CMake, ставшей в последние годы де-факто стандартом в индустрии программного обеспечения, к сожалению, имеется очень мало учебных материалов, посвящённых этой системе. Единственный печатный учебник на английском языке — Mastering CMake [11], авторами которого являются разработчики самой системы, на русском языке подобных изданий нет вообще.

В данном учебнике рассматриваются следующие вопросы:

— принципы работы системы CMake, сравнение её с другими аналогичными средствами;

[1] http://www.cmake.org/ (дата обращения: 24.02.2015).

— описание и построение простых проектов с помощью системы CMake;

— основные конструкции языка CMake;

— организация построения при помощи системы CMake программных проектов, использующих внешние библиотеки и инструменты. В частности, будут рассмотрены вопросы построения проектов, использующих набор библиотек Qt [1; 8].

Выбор рассматриваемых в учебнике библиотек неслучаен: помимо их высокой популярности, каждая из них требует отдельного подхода при подключении к проекту, управляемому системой CMake. Примеры рассматриваются в порядке возрастания сложности: от набора библиотек OpenCV, в котором встроена поддержка CMake, до библиотеки Crypto++, для которой необходимо вручную реализовывать все операции по поиску и подключению. Продемонстрированные в соответствующих разделах приёмы могут оказаться полезными при работе со многими другими библиотеками. Поэтому они по возможности оформлены в виде, облегчающем повторное использование.

Большинство приводимых в учебнике примеров ориентированы на сферу практического применения системы CMake. При необходимости даются минимальные сведения об используемых библиотеках и инструментах. Каждый из примеров сопровождается подробными инструкциями по его сборке. Рассматриваются наиболее типичные проблемы, которые могут возникнуть у читателя, желающего повторить описанные экс-

перименты. Из-за всего этого в тексте учебника содержится большое количество информации, не имеющей прямого отношения к CMake, но необходимой для понимания типичных сценариев использования тех или иных его средств.

На момент написания данного учебника последними доступными стабильными версиями ПО, которые использовались для проверки примеров, являлись:

— CMake 3.3.2;

— OpenCV 2.4.11 и 3.0.0;

— Boost 1.59.0;

— Qt 5.5.0;

— Crypto++ 5.6.2;

— Doxygen 1.8.10;

— GraphViz 2.38.

Текущий документ имеет версию 1.0.

1. Принципы работы систем автоматического построения

В этой главе мы коротко повторим основные принципы модульного программирования, которые понадобятся для дальнейшего изложения материала. Также будет приведён обзор основных систем построения проектов, будут рассмотрены их основные достоинства и недостатки в сравнении друг с другом.

1.1. Модульное программирование

При работе над программным проектом разработчик сталкивается со следующими основными проблемами:

— Время компиляции. При увеличении объёма исходного кода растёт и время, требуемое для его компилирования. В соответствии с каскадной моделью процесса разработки ПО (а также другими моделями) [6], этап кодирования и тестирования заключается во внесении изменений в исходный код, его компиляции, запуске тестовой программы, выявлении ошибок, внесении изменений в исходный код, повторной компиляции и т. д. Таким образом, вполне возможно,

что бóльшую часть рабочего времени разработчики будут просто ожидать завершения очередной компиляции.

— Организация коллективной работы над проектом. Если проект требует больших трудозатрат, желательно выделить для его выполнения группу из нескольких человек, работающих одновременно. Эффективное распределение задач между ними представляет собой одну из проблем руководства процессом разработки.

— Грамотная организация исходного кода. При увеличении объёма исходных кодов сверх критической массы вполне может оказаться так, что разобраться в них будет уже невозможно даже его авторам. Поэтому данная проблема также практически всегда рано или поздно встаёт перед разработчиками.

— Повторное использование собственного и чужого кода. При разработке алгоритмов, способных быть применёнными не только в текущем программном проекте, но в перспективе и в других, возникает задача их оформления в виде, удобном для повторного использования. Для ранее разработанных алгоритмов, как и для стороннего кода, возникает задача встраивания в текущий проект.

Для решения всех перечисленных задач предназначен подход, называемый *модульным программированием* (modular programming). В соответствии с ним исходный код проекта разделяется на части, называемые *исходными, или транслируемыми, модулями* (рис. 1.1). Между модулями устанавливаются зависи-

мости по вызываемым подпрограммам, доступу к глобальным переменным, определённым в других модулях, и т. д. Например, в языках C/C++ средствами межмодульного взаимодействия являются функции и переменные с внешней связью (имеющие в объявлении ключевое слово `extern`, которое в случае функций подразумевается по умолчанию).

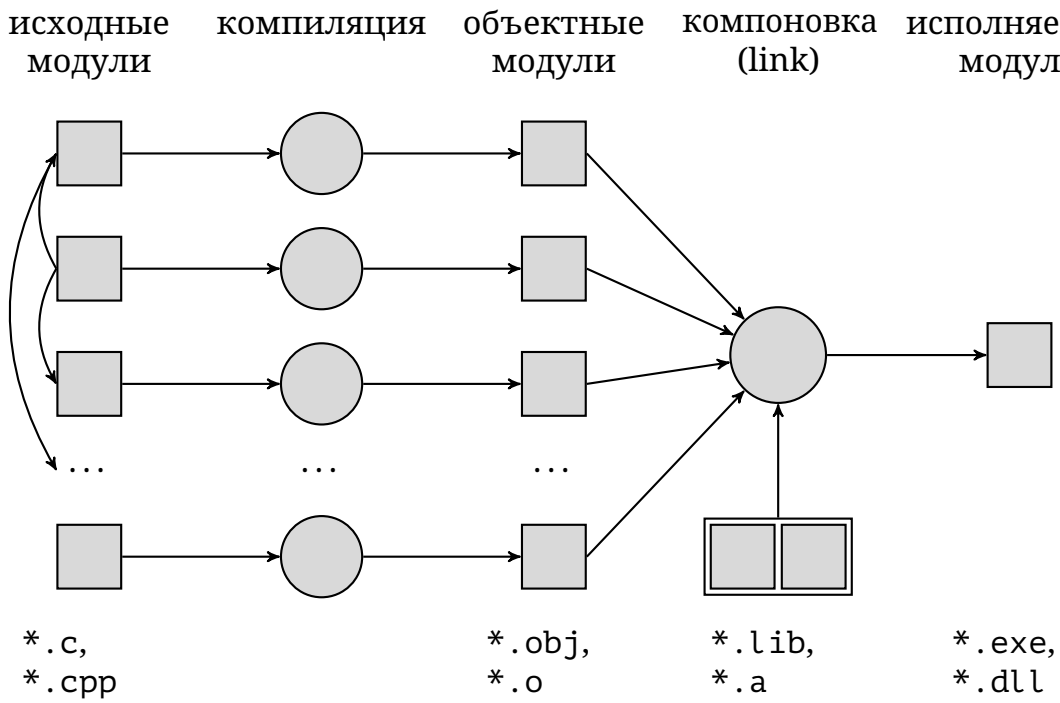

Рис. 1.1. Схема двухэтапного построения проекта

Процедура создания конечного *исполняемого модуля* при модульном подходе разделяется на два этапа. На первом из них для каждого транслируемого модуля независимо от остальных запускается компилятор, который создаёт соответствующий ему *объектный модуль*. Как правило, объектные модули представляют собой почти полностью скомпилированные версии исходных модулей, за исключением одной особенности. При вызове функций и при обращении к переменным в машин-

ном коде используются адреса этих объектов. При обращении к функциям и переменным, определённым вне текущего модуля, компилятор не может знать их адресов и поэтому вставляет в объектный модуль ссылки на их символические имена — имена, построенные некоторым образом на основе имён этих объектов в исходном коде. Каждый объектный модуль содержит *таблицу импорта* — набор символических имён объектов из других модулей, используемых в текущем. Также объектный модуль содержит *таблицу экспорта* — набор имён объектов, предоставляемых текущим модулем для использования извне.

На втором этапе все объектные модули передаются следующему инструменту, называемому *компоновщиком* (linker), или редактором связей. Он собирает из них исполняемый модуль, осуществляя *разрешение зависимостей* — просмотр всех таблиц импорта/экспорта и замену ссылок на символические имена реальными адресами (которые ему уже должны быть известны, поскольку он выполняет размещение в памяти всех функций и переменных).

Кроме объектных модулей компоновщику также могут передаваться *библиотеки*, как правило, представляющие собой несколько объектных модулей, объединённых для удобства в один файл. Именно так организованы стандартные библиотеки, которые неявно используются компоновщиком.

При помощи разбиения исходного кода на модули решаются задачи организации коллективной работы (разным разработчикам поручается реализация разных модулей), грамот-

ной организации исходного кода и его повторного использования. Но как решается задача ускорения компиляции? Описанная процедура двухэтапного построения (build) не быстрее (а часто и медленнее) обычного. Иногда даже используется приём, называемый *монолитным построением* (monolithic build), когда все транслируемые модули подключаются из одного при помощи директив `#include` и компилятору передаётся этот единственный «модуль».

Однако описанный здесь процесс двухэтапного построения может сочетаться с методом, который иногда называется *инкрементным построением* (incremental build[1]). Согласно ему при внесении изменений в какие-то транслируемые модули для повторного построения необходимо заново перекомпилировать только эти изменённые модули, а также, возможно, модули, от них зависящие. При этом если в модуле был изменён только внутренний код каких-то функций, но не их заголовки, зависимые модули в перекомпиляции не нуждаются. Изменение заголовка функции означает изменение машинного кода для её вызова, и если оно не произошло, то и перекомпилировать использующий её код не нужно. Работа компоновщика, как правило, осуществляется гораздо быстрее сложного процесса компиляции, поэтому здесь может быть достигнута значительная экономия времени.

[1] В инженерии программного обеспечения этот термин может употребляться в ином смысле.

1.2. Автоматизация построения проектов

Описанное двухэтапное построение может быть реализовано и вручную, однако на практике это очень неудобно. Автоматически этот процесс может быть реализован на основе сравнения даты изменения исходного и зависимого файлов.

Замечание: для реальных программных проектов схема построения[2] может быть более общей, чем изображённая на рис. 1.1. Одни файлы могут создаваться из других при помощи некоторых инструментов (не обязательно компиляторов). Например, приложения, использующие набор библиотек Qt, могут вызывать его инструменты, генерирующие промежуточный код на C++ [1; 8]. Вместо одного исполняемого модуля может создаваться несколько конечных целей. Например, при помощи различных инструментов может генерироваться файл справки или документация к коду. Концепция автоматизированного двухэтпаного построения легко распространяется и на такие случаи. ▲

Таким образом, описание построения программного проекта должно включать в себя информацию о следующих его компонентах:

Входные файлы: исходные файлы проекта (транслируемые модули, графические изображения для элементов пользова-

[2] В различных русскоязычных источниках термин «сборка» иногда употребляется как синоним используемого в настоящем учебнике слова «построение», а иногда — вместо слова «компоновка».

тельского интерфейса и т. д.), которые создаются и редактируются разработчиком.

Промежуточные файлы: вспомогательные файлы, создаваемые в процессе построения проекта (объектные модули и т. д.).

Выходные файлы: результирующие файлы, представляющие собой конечный результат построения (исполняемые модули, библиотеки и т. д.).

Правила вызова инструментов (или просто правила): описание того, какой внешний инструмент (компилятор, компоновщик и т. д.) с какими аргументами командной строки должен вызываться для получения одних файлов из других.

Цель: файл, создаваемый в результате исполнения правила (промежуточный или выходной).

Зависимости проекта (или просто зависимости): описание того, какие промежуточные и выходные файлы проекта зависят от входных, промежуточных и выходных файлов и какие правила должны быть применены для создания этих файлов. Таким образом, зависимости определяют ациклический граф с вершинами, соответствующими файлам проекта, и дугами, соответствующими правилам. Инструмент построения должен уметь сравнивать время изменения сгенерированного файла с временем изменения файлов, от которых он зависит. Если время изменения какого-либо из них окажется позже времени изменения зависимого

файла (или он ещё вообще не существует), он должен быть создан заново применением соответствующего правила.

Описание проекта: описание зависимостей проекта для инструмента построения в одном или нескольких файлах.

Инструмент автоматического построения должен уметь выполнять две операции:

Инкрементное построение: процесс создания всех выходных файлов проекта с применением правил только при необходимости (с учётом времени изменения файлов и их существования).

Полное построение (перестроение): применение всех описанных правил для создания выходных файлов без учёта времени изменения файлов.

Последняя операция может быть полезна в ряде случаев, например:

— Изменение системного времени на компьютере, на котором выполняется построение, из-за чего механизм проверки времени изменения файлов перестаёт быть надёжным средством определения необходимости применения правил.

— Проверка повторяемости построения проекта с выявлением некоторых типичных ошибок. Например, из проекта могут быть ошибочно удалены некоторые исходные файлы, в то время как созданные ранее по ним объектные модули до сих пор существуют на диске и используются при

инкрементном построении выходных файлов. Попытка построить проект на другом компьютере в этом случае, конечно, будет неудачной.

Ещё одной возможностью, которую предоставляют некоторые инструменты построения, является *построение вне каталога проекта* (out-of source build). Эти инструменты дают возможность выполнять построение таким образом, чтобы все промежуточные и выходные файлы создавались в так называемом *каталоге построения*, отдельном от каталога исходных файлов проекта (*каталога проекта*). При этом каталог исходных файлов не «захламляется» большим количеством генерируемых файлов, обычно занимающих очень много места.

В завершение обсуждения автоматизации построения осталось только рассмотреть вопрос, каким образом средства построения могут определить транслируемые модули, зависимые от изменённого, которые также нужно перекомпилировать. Для этого рассмотрим следующий пример.

Пример

Пусть в проекте имеется исходный модуль `a.cpp` на языке C++, в котором определена экспортируемая функция `f()`:

```
void f()
{
  // ...
}
```

Пусть в проекте также имеется исходный модуль `b.cpp`, который вызывает функцию `f()`, импортируемую из модуля `a.cpp`:

```
// ...

void g()
{
  // ...
  f();
  // ...
}
```

По этим исходным модулям при построении создаются объектные модули `a.o` и `b.o`.

Пусть теперь разработчик изменил заголовок функции `f()`, например добавив один параметр:

```
void f(int n = 0)
{
  // ...
}
```

Теперь для построения проекта нужно перекомпилировать модули `a.cpp` и `b.cpp`. Возможны два варианта:

1) Разработчик использовал повторное объявление функции `f()` непосредственно в модуле `b.cpp`:

```
void f();
```

```
void g()
{
  // ...
  f();
  // ...
}
```

В этом случае разработчик обязан исправить это объявление функции `f()`. Таким образом, он также изменяет модуль `b.cpp`. Инструмент построения обнаружит, что время изменения `b.cpp` окажется позже времени изменения `b.o`, и также запустит команды перекомпиляции `b.cpp`.

2) Разработчик для удобства вынес объявление функции `f()` в заголовочный файл `a.h` (что на практике чаще всего и используется):

```
void f();
```

При этом в файле `b.cpp` подключается `a.h` при помощи директивы `#include`:

```
#include "a.h"

void g()
{
  // ...
  f();
  // ...
```

```
}
```

При изменении заголовка функции `f()` разработчик теперь должен также изменить файл `a.h`, однако файл `b.cpp` остаётся без изменений. Разработчик мог бы вручную установить время изменения файла `b.cpp`, однако в крупных программных проектах поиск всех зависимых файлов является сложной задачей. Здесь возможны два выхода:

— Указание в описании проекта того, что файл `b.o` зависит не только от `b.cpp`, но и от `a.h`. Этот вариант не накладывает особых требований на инструмент построения, однако накладывает на разработчика обязанность исправлять описание проекта каждый раз одновременно с добавлением/удалением очередной директивы `#include`.

— Требование от инструмента построения способности синтаксического разбора исходных модулей с целью автоматического определения подключаемых файлов. *

1.3. Обзор инструментов построения проектов

Несмотря на то что в настоящее время уже существует большое количество инструментов автоматизации построения, принципы их работы близки всего к нескольким наиболее популярным системам. В данном разделе мы кратко рассмотрим эти инструменты, отметив их основные особенности.

1.3.1. make

Один из первых инструментов построения, появившийся в 1977 г. в операционной системе Unix. В настоящее время инструмент является частью стандарта POSIX [13]. Существует множество реализаций, в большей или меньшей степени совместимых со стандартом и друг с другом. Наиболее распространённой является реализация GNU make [10], входящая в состав систем на основе GNU/Linux.

Инструмент представляет собой утилиту командной строки, которая при запуске ищет текстовый файл с описанием проекта (по умолчанию имеющий имя `Makefile` в текущем каталоге) и осуществляет процесс построения в соответствии с записанными в нём правилами. В командной строке при вызове инструмента make можно указать основную цель. Некоторые команды, запускающие стандартные цели:

```
make              # построить проект
make install      # выполнить установку
make clean        # удалить временные файлы
```

Правила построения инструмента make определяют, какой из выходных/промежуточных файлов зависит от каких других файлов и какую последовательность команд («рецепт») необходимо запустить для создания этого файла.

Формат правила на языке make:

$$\langle \textit{цель} \rangle : \left[\langle \textit{зависимость}_1 \rangle \ \ldots \ \langle \textit{зависимость}_m \rangle \right]$$
$$\quad \langle \textit{команда}_1 \rangle$$

. . .

⟨*команда*$_n$⟩

Здесь ⟨*зависимость*$_1$⟩ и т. д. — файлы, от которых зависит данная цель. Команды записываются с начала строки после символа табуляции. Если список зависимостей не пуст, команды могут отсутствовать — такое правило может добавлять новые зависимости к уже объявленной цели.

Возможно использование параметризованных правил, при помощи которых можно определить зависимости сразу для целого класса файлов. Например, определить, каким образом можно получить любой объектный файл из cpp-файла с тем же базовым именем. Кроме файлов в качестве целей в правилах могут быть указаны абстрактные цели, не связанные с конкретным файлом («фальшивые» цели — phony targets). Примером фальшивой цели является цель `install` из примера выше, выполняющая установку программного компонента в систему.

Основными достоинствами утилиты make являются:

— Нетребовательность к вычислительным ресурсам: программа может выполняться в консольном режиме.

— Универсальность: при помощи языка make можно описать проект со сложными зависимостями и любыми правилами, требующими запуска инструментов командной строки.

— Настраиваемость: при правильном написании файлов проекта процесс построения можно легко настроить из ко-

мандной строки, определив нужную версию компилятора, параметры компиляции и т. д.

К основным недостаткам утилиты make относятся:

— Сложность и невыразительность языка make, являющиеся обратной стороной его универсальности: даже описание простых проектов может выглядеть слишком громоздким[3]. Задание правил автоматического поиска подключения заголовочных файлов возможно, но опирается на вызов компилятора gcc и также требует применения сложных конструкций.

— Ограниченность в переносимости: так как в правилах непосредственно указываются команды запуска инструментов, исполнение построения ограничено средой, в которой доступны все используемые инструменты и они совместимы по параметрам командной строки[4]. Например, для построения проектов, использующих инструменты POSIX, в системе Microsoft Windows необходима установка пакетов MinGW[5] и MSYS (или подобных), предоставляющих реализации этих инструментов в данной системе. Выбор компилятора ограничивается теми, которые понимают параметры командной строки, используемые в make-файле.

— Проблемы с разбиением сложных проектов на части и описанием правил построения подпроектов при помощи от-

[3] http://www.conifersystems.com/whitepapers/gnu-make/ (дата обращения: 26.12.2014).

[4] http://freecode.com/articles/what-is-wrong-with-make (дата обращения: 26.12.2014).

[5] http://mingw.org/ (дата обращения: 31.12.2014).

дельных make-файлов (так называемые «рекурсивные» make-файлы — утилита make вызывается в качестве команды при обработке make-файла верхнего уровня) [12].

1.3.2. Autotools

Autotools представляет собой набор инструментов для создания кроссплатформенных пакетов установки программного обеспечения, поставляемого в виде исходных кодов (так называемая система построения GNU — GNU build system[6]). Разработка инструментов Autotools началась в 1991 г. в качестве системы построения для утилит GNU. Инструменты предназначены для решения проблемы различий в компиляторах, наборах стандартных заголовочных файлов и т. д. на различных системах. Сами по себе эти программы не являются инструментами автоматизации построения, однако они генерируют файл описания проекта для утилиты make. На вход им подаётся описание проекта на более высокоуровневом языке, чем язык make. В результате их работы создаются, в частности, следующие файлы:

— `configure` — сценарий на языке командной оболочки Bourne Shell, создаваемый утилитой `autoconf`[7], который собирает сведения о системе, необходимые для корректного построения программного пакета;

— `Makefile.in` — шаблон файла описания проекта, создаваемый утилитой `automake`[8], при помощи которого сценарий `configure`[9] генерирует файл `Makefile`;

[6] http://airs.com/ian/configure/ (дата обращения: 26.12.2014).
[7] https://www.gnu.org/software/autoconf/ (дата обращения: 31.12.2014).
[8] https://www.gnu.org/software/automake/ (дата обращения: 26.12.2014).
[9] Точнее, генерируемый им сценарий `config.status`.

— `config.h.in` — (необязательный) шаблон заголовочного файла, создаваемый утилитой `autoheader`, при помощи которого сценарий `configure` генерирует заголовочный файл `config.h` с определениями макросов, описывающих целевую систему (например, поддерживает ли компилятор ту или иную возможность, найден ли заданный заголовочный файл в составе компилятора, есть ли заданная функция в стандартной библиотеке и т. д.). Без этого файла макросы пришлось бы определять для использования в исходном коде при помощи параметров командной строки компилятора, что при большом их количестве очень неудобно.

Таким образом, наличия Autotools в системе, на которой выполняется построение программного пакета, не требуется. Пользователю достаточно выполнить следующие команды в оболочке:

```
cd ⟨каталог, в котором находятся исходные коды⟩
./configure ⟨дополнительные настройки⟩
make
make install
```

Во время работы сценарий `configure` запускает серию тестов, для некоторых из них он может пытаться скомпилировать короткие программы. Результаты тестов могут сохраняться в файле кэша для ускорения выполнения сценария при повторных запусках.

Применение инструментов Autotools предоставляет следующие преимущества:

— Генерируемый файл описания проекта соответствует требованиям переносимости GNU (использование стандартных имён целей, переменных и т. д.). Разработчику не нужно специально беспокоиться об этих вопросах, как при непосредственной работе с утилитой make.

— Также генерируемый файл описания проекта автоматически решает многие типичные проблемы, возникающие перед разработчиком при ручном создании make-файла. Например, в него добавляются правила для автоматического отслеживания зависимостей от заголовочных файлов на момент запуска построения проекта.

Однако использование системы Autotools имеет и свои недостатки:

— Переносимость пакета установки ограничена системами, так или иначе совместимыми со стандартом POSIX. К требованиям относительно программного окружения при использовании утилиты make добавляется требование наличия интерпретатора сценариев Bourne Shell. Их реализации для системы Windows (cygwin[10], MSYS) очень неэффективны, из-за чего построение проектов может выполняться крайне медленно.

— Используемый системой Autotools язык описания построения является очень сложным.

[10] https://www.cygwin.com/ (дата обращения: 27.12.2014).

1.3.3. Интегрированные среды разработки

К интегрированным средам разработки (integrated development environments, или сокращённо IDE) относятся такие системы, как Microsoft Visual Studio[11], Code::Blocks[12], Eclipse[13], NetBeans[14] и т. д. Они представляют собой удобный для разработчика инструмент, реализованный по принципу «всё в одном». Помимо задачи построения проекта они также решают множество других взаимосвязанных задач: редактирование исходных кодов и элементов пользовательского интерфейса, быстрый поиск участков кода в соответствии с их семантикой, отладка, доступ к справочным материалам и т. д. Самые первые интегрированные среды появились ещё в 1960-е гг.[15]

Описание проекта представляется в современных интегрированных средах визуально в виде дерева исходных файлов. Различные цели представляются в них в виде «проектов». Несколько проектов объединяются в «решения» (solutions в терминологии Visual Studio), «рабочие пространства» (workspaces в терминологии Code::Blocks) и т. п. Между проектами одного решения можно устанавливать зависимости. Например, решение может состоять из проекта библиотеки и проекта исполняемого модуля, использующего эту библиотеку. В этом случае имеет смысл установить зависимость второго проекта от первого, тогда построение всегда будет выполняться в нужном порядке. Кроме того, при изменении исходного кода библиотеки для ис-

[11] http://www.visualstudio.com/ (дата обращения: 27.12.2014).
[12] http://www.codeblocks.org/ (дата обращения: 27.12.2014).
[13] https://eclipse.org/ (дата обращения: 27.12.2014).
[14] https://netbeans.org/ (дата обращения: 27.12.2014).
[15] http://en.wikipedia.org/wiki/Dartmouth_BASIC (дата обращения: 02.01.2015).

полняемого модуля будет заново выполняться компоновка при инкрементном построении.

Для каждого проекта есть возможность настроить способ построения при помощи диалогового окна его свойств (рис. 1.2). Можно указать общие свойства проекта: (тип: приложение, библиотека и т. д.); имя выходного файла и т. д., параметры компилятора (используемые оптимизации, включение отладочных символов и т. д.), препроцессора, компоновщика и другие. Некоторые из этих свойств можно устанавливать для каждого исходного файла проекта отдельно.

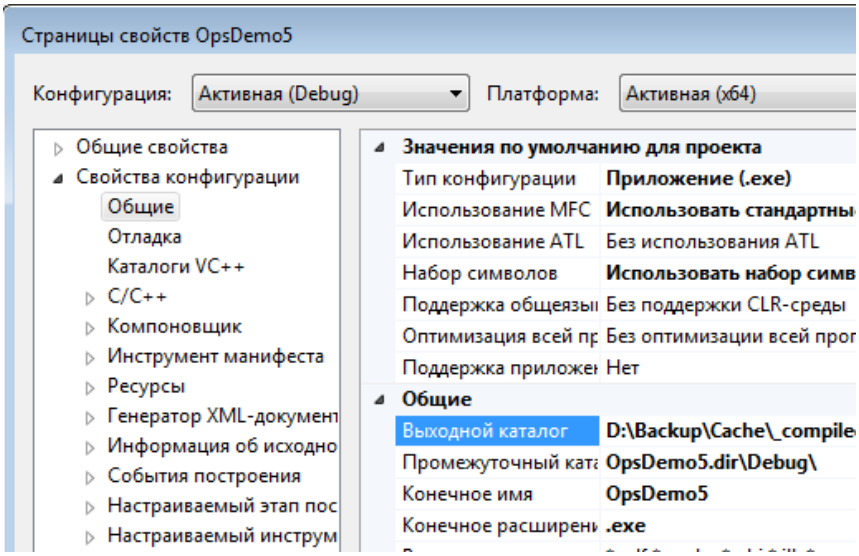

Рис. 1.2. Часть диалогового окна свойств проекта в среде Microsoft Visual Studio

Важными свойствами проекта являются пути для поиска препроцессором заголовочных файлов, подключаемых в исходном коде при помощи директив `#include`, пути для поиска компоновщиком библиотек, а также имена используемых при построении библиотек. При помощи этих свойств устанавливают-

ся связи с библиотеками, собираемыми в других проектах, а также сторонними, которые поставляются в готовом виде.

Если для обработки исходного файла требуется нестандартный инструмент, в свойствах проекта для него можно указать путь к инструменту вместе с аргументами командной строки (раздел «Настраиваемый инструмент построения» в среде Visual Studio). Таким образом можно, например, вызывать компилятор метаинформации и компилятор ресурсов библиотеки Qt.

Интегрированными средами поддерживаются «конфигурации» — наборы свойств проекта, между которыми можно переключаться для выполнения разных вариантов построения. По умолчанию создаются конфигурации «Debug» и «Release», предназначенные соответственно для отладки и поставки конечному пользователю. Например, в конфигурации «Debug» отключены оптимизации в целях экономии времени построения и упрощения отладки, но включено генерирование отладочных символов, увеличивающих размер выходных файлов, также в коде включаются дополнительные проверки корректности (макрос `assert()` и т. д.).

Для некоторых сред разработки свойства проектов можно параметризовать, аналогично make-файлам — с использованием переменных. Например, среда Visual Studio поддерживает использование в настройках стандартных переменных, называемых «макросами» (путь к выходному файлу, имя текущей конфигурации, значение переменной окружения и т. д.). Среда Code::Blocks поддерживает использование только переменных, задаваемых пользователем.

Если рассматривать только функцию управления построением проекта, интегрированные среды имеют следующие преимущества:

— Удобство и скорость вызова функции построения проекта из среды разработки: для этого достаточно нажать сочетание клавиш на клавиатуре. Не нужно помнить никаких команд, набираемых в командной строке.

— Интуитивно понятная процедура добавления файлов в проект и настройка свойств проекта. Для начала работы разработчику не нужно изучать никаких дополнительных языков описания проектов, достаточно освоиться с пользовательским интерфейсом интегрированной среды. При создании проекта среда задаёт ряд общих вопросов при помощи мастера, после чего автоматически заполняет свойства проекта, создавая конфигурации «Debug» и «Release», подходящие для большинства случаев.

— Автоматическое отслеживание зависимостей исходных модулей от подключаемых заголовочных файлов.

Кроме всех перечисленных выше достоинств, использование интегрированных сред разработки, к сожалению, также имеет свои недостатки:

— Визуальные редакторы очень удобны для решения узкого круга стандартных задач, на которые они рассчитаны, и крайне неудобны во всех остальных случаях. Так, добавление в проект входного файла, обрабатываемого нестан-

дартным инструментом (например, одним из инструментов Qt), требует ручного прописывания сложных команд в свойствах проекта. Автоматизировать такую операцию для массового добавления подобных файлов очень сложно.

— Настройки проектов сложно устанавливать и менять в массовом порядке. Скажем, если существует решение из десятка-двух проектов, каждый из которых использует некоторую библиотеку, то при изменении пути к ней (например, при желании разработчика попробовать построить решение с другой версией библиотеки) нужно открыть свойства всех проектов и поменять в каждом из них пути поиска библиотек и заголовочных файлов. Причём это нужно не забыть проделать для каждой из конфигураций.

— Не очень удобно реализовывать построение вне каталога проекта с использованием интегрированных сред. Как правило, они создают выходные и промежуточные файлы в нескольких подкаталогах, разбросанных по всему решению. Чтобы «заставить» интегрированные среды создавать временные файлы в отдельном каталоге, необходимо изменить соответствующие пути для каждого проекта, каждой конфигурации. При использовании относительных путей расположение выходного каталога фиксируется относительно каталога проекта, изменение его требует очередного массового изменения настроек проектов.

— Также не очень понятно, как следует лучше поступать при подключении внешних библиотек. Их расположение

может быть разным для разных систем, в то время как в настройках проектов всегда указываются конкретные пути. Нужно либо требовать расположения библиотек в фиксированных каталогах (что неудобно), либо менять настройки проектов при переходе на другую систему (например, при помощи переменных). Некоторые интегрированные среды поддерживают указание путей поиска в глобальных настройках. Однако эти настройки влияют на построение всех проектов, поэтому также нежелательны. В среде Visual Studio, начиная с версии 2010, от глобальных настроек путей поиска вообще отказались, перенеся их в настройки проекта. Однако использовать их для хранения путей к библиотекам всё равно нежелательно, так как файлы проектов обычно хранятся в общем репозитарии кода вместе с остальными исходными файлами. Таким образом, изменение их на одном компьютере приведёт к транслированию изменений всем остальным разработчикам.

— Для комфортной работы с интегрированными средами требуются повышенные вычислительные мощности. В отличие от утилиты make, их, как правило, невозможно запустить в консольном режиме.

В целом интегрированные среды больше подходят для регулярной разработки некоторого программного проекта. С другой стороны, инструменты make, Autotools и им подобные более удобны для организации системы одноразового построения и установки стороннего программного обеспечения, а также для

построения без участия пользователя системами ночного тестирования и т. п.

1.3.4. qmake

Инструмент qmake распространяется вместе с набором библиотек Qt начиная с версии 3.0 (2001 г.). Изначально он являлся переписанной на C++ версией инструмента tmake — сценария на языке Perl, разработка которого велась с 1996 г. Основной причиной, вынудившей разработчиков Qt создать собственный инструмент автоматизации построения, была недостаточная гибкость набора инструментов Autotools.

Так же как и Autotools, qmake не является системой построения в строгом понимании этого слова, он всего лишь генерирует файлы описания проектов для других систем (рис. 1.3).

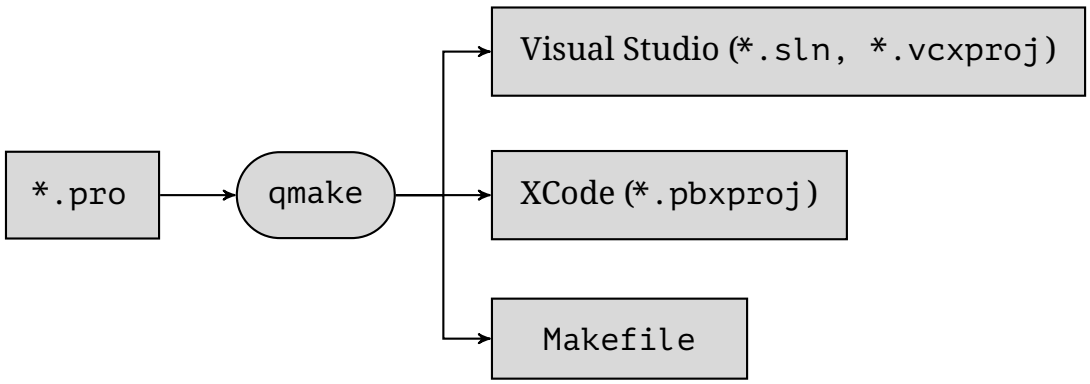

Рис. 1.3. Генерация проектов при помощи утилиты qmake

На вход программе qmake подаётся описание проекта на высокоуровневом языке (один или несколько файлов с расширением «`.pro`»), в результате инструмент создаёт файл для системы make (`Makefile`). Также возможно генерирование фай-

лов проектов и решений для интегрированных сред разработки Visual Studio и XCode[16].

Интегрированная среда Qt Creator использует файлы «`*.pro`» в качестве файлов описания проекта. Однако в отличие от других сред разработки, описанных в п. 1.3.3, эта среда поддерживает только визуальное редактирование списка файлов проекта, но не его настроек. Для изменения настроек необходимо отредактировать файл проекта в текстовом редакторе (например, в самой среде). В диалоговых окнах среды Qt Creator можно только устанавливать настройки, не указываемые в файлах проекта: каталог построения, параметры командной строки при вызове qmake и т. д. Для осуществления построения среда Qt Creator сначала запускает инструмент qmake, затем make.

Утилита qmake записывает генерируемые файлы в заданный каталог (по умолчанию в текущий). При этом входные файлы добавляются в генерируемые проекты по относительным путям, а выходные и промежуточные файлы позже, на этапе построения, записываются в подкаталоги относительно каталога с генерируемым проектом. Таким образом, можно легко организовать построение вне каталога проекта, причём расположение каталога с промежуточными и выходными файлами определяется запуском утилиты qmake. То есть расположение каталога построения можно легко менять, не меняя файла описания (`*.pro`).

Язык qmake поддерживает специальные переменные, определяющие тип проекта (приложение, библиотека и т. д.),

[16] https://developer.apple.com/xcode/ (дата обращения: 27.12.2014).

списки файлов исходных кодов, заголовочных файлов, файлов ресурсов и т. д. При необходимости автоматически генерируются правила для подключения библиотек Qt, вызова инструментов обработки исходных файлов из состава Qt, хотя qmake можно использовать и для проектов, не использующих Qt. Также языком поддерживаются условные конструкции и функции (встроенные и определяемые пользователем). В условных конструкциях можно проверять параметры конфигурации (например, целевую платформу), вызывать встроенные и пользовательские логические функции, при помощи которых можно организовывать циклы, выводить диагностические сообщения и т. д. Эти и другие средства языка позволяют расширять возможности инструмента qmake, добавляя поддержку новых компиляторов, языков программирования, инструментов, платформ, библиотек и т. д.

Таким образом, можно выделить следующие достоинства инструмента qmake:

— Простой высокоуровневый язык описания проектов.

— Переносимость: поддержка большого количества целевых платформ и компиляторов (GCC, clang, Intel C Compiler, Microsoft Visual C++ и т. д.). Требования к системе разработчика включают наличие инструмента qmake, т. е. подойдёт любая система, на которую портирован набор библиотек Qt.

— Возможность работы над проектами в средах Visual Studio, XCode, Qt Creator. Если желательно автоматизировать построение, можно сгенерировать обычные make-файлы.

— Поддержка в генерируемых проектах библиотек Qt, добавление которых вручную является слишком сложным.

— Простая в использовании поддержка построения вне каталога проекта.

— Расширяемость.

У инструмента qmake нет существенных недостатков, но всё же можно выделить несколько проблем, препятствующих его широкому распространению:

— Ориентированность в первую очередь на набор библиотек Qt. Хотя при помощи расширений возможно научить qmake работать с другими инструментами, в составе утилиты эти расширения отсутствуют. Что касается библиотек, инструменту qmake известны расположения только заголовочных файлов, библиотечных модулей и т. д. из состава Qt. Пути к файлам других библиотек нужно указывать явно.

— В отличие от инструментов Autotools, система qmake не предусматривает возможности запуска серии тестов для определения особенностей среды построения. Также не предусмотрены средства для генерирования заголовочных и прочих файлов.

1.3.5. CMake

CMake является свободным инструментом с открытым исходным кодом, основным разработчиком которого выступает компания Kitware[17]. Название системы расшифровывается как «cross-platform make». Разработка инструмента ведётся с 1999 г., в качестве прототипа была использована утилита pcmaker, написанная в 1997 г. одним из авторов CMake. В настоящее время инструмент внедряется в процесс разработки многих программных продуктов, в качестве примеров широко известных проектов с открытым кодом можно привести KDE[18], MySQL[19], Blender[20], LLVM + clang[21] и многие другие.

Принцип работы инструмента CMake аналогичен принципу работы qmake: из каталога исходных кодов считывается файл `CMakeLists.txt` с описанием проекта, на выходе инструмент генерирует файлы проекта для одной из множества конечных систем построения (рис. 1.4).

Требования для сборки самого CMake включают наличие утилиты make и интерпретатора сценариев на языке bash либо скомпилированного инструмента CMake одной из предыдущих версий, а также компилятора C++. При этом в исходных кодах CMake преднамеренно используются только возможности языка и стандартной библиотеки, поддерживаемые достаточно старыми версиями компиляторов. Таким образом, CMake переноси́м на большое количество платформ.

[17] http://www.kitware.com/ (дата обращения: 02.01.2015).
[18] https://www.kde.org/ (дата обращения: 03.01.2014).
[19] http://www.mysql.com/ (дата обращения: 03.01.2014).
[20] http://www.blender.org/ (дата обращения: 03.01.2014).
[21] http://llvm.org/ (дата обращения: 03.01.2014).

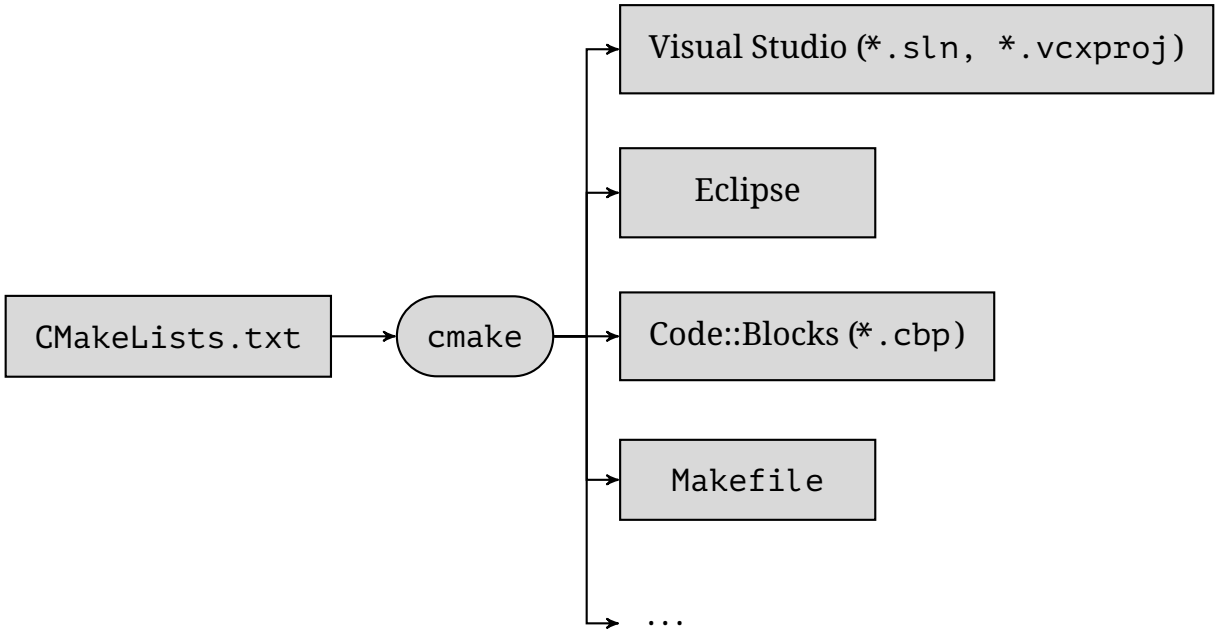

Рис. 1.4. Генерация проектов при помощи утилиты CMake

Следует отметить, что современные версии интегрированной среды Qt Creator в дополнение к описаниям проектов на языке qmake также поддерживают язык CMake.

Система CMake управляется при помощи универсального процедурного языка. В то время как инструмент qmake позволяет в более простой и компактной форме выполнять описание проектов, использующих набор библиотек и инструментов Qt, при помощи CMake легче описывать проекты, которые используют другие библиотеки и инструменты, а также решать нестандартные задачи, которые возникают при организации процесса построения [14]. Основные возможности CMake, отсутствующие в qmake, включают в себя следующее:

— Простой интерфейс для подключения библиотек, являющихся результатами построения одних целей, к другим. Например, проект может включать цель библиотеки и ис-

пользующего её приложения. В CMake можно легко установить зависимость между такими целями, при этом к правилам построения приложения будут автоматически добавлены все необходимые настройки компилятора и компоновщика для подключения библиотеки. Те же правила в qmake придётся определять на более низком уровне. Кроме этого, в описании цели библиотеки можно определять дополнительные настройки, которые будут использованы при построении всех её клиентов, что избавляет от их повтора для каждого из них.

— Команды и сценарии для поиска в системе наборов библиотек и/или инструментов (*пакетов* в терминологии CMake). В состав CMake входит большое количество сценариев (*модулей поиска*) для наиболее популярных и распространённых в среде разработчиков пакетов. Можно создавать собственные модули поиска на языке CMake, также можно встраивать поддержку CMake в собственные наборы библиотек.

— Средства генерирования исходных файлов и сценариев в процессе построения (аналогично системе Autotools, которая способна создавать правила для генерирования файлов `config.h` и `Makefile`).

В целом можно утверждать, что qmake больше ориентирован на использование инструментария Qt, в состав которого он входит, тогда как CMake, скорее, является более универсальным решением.

1.4. Упражнения

1.4.1. Тест рубежного контроля

1. *(Вопрос со множественным выбором.)* Укажите, какие из перечисленных ниже систем построения проектов выполняют свои функции самостоятельно вместо генерирования файлов для других систем:

   (a) make; (b) Autotools;

   (c) Microsoft Visual Studio; (d) qmake;

   (e) CMake.

2. *(Вопрос со множественным выбором.)* В примере на с. 19 было продемонстрировано, как в программе на языке C++ можно исправить всего лишь объявление функции так, чтобы скомпилированный код для её вызова потребовал изменений. Действительно, при появлении у функции параметра необходимо в месте её вызова помещать в стек аргументов его значение по умолчанию (0). Теперь рассмотрим следующий код модуля на языке C:

```
void f(double);

void g()
{
  f(1.);
}
```

Укажите все варианты изменения объявления функции `f()` из перечисленных ниже, которые являются допустимыми в языке C и которые приведут к изменению кода вызова этой функции:

(a) Удаление объявления функции `f()` вообще.

(b)
```
int f(double);
```

(c)
```
void f(int);
```

(d)
```
void f(double, int = 1);
```

(e)
```
void f(double, ...);
```

(f)
```
void f(double);
void f(int);
```

(g)
```
namespace X { void f(double); }
using X::f;
```

1.4.2. Проектное задание

1. Выполните сформулированное ниже задание, используя любую из интегрированных сред разработки. Результатом реализации части a задания должна быть статическая библиотека, а части b — программа, использующая библиотеку. Структура каталога проекта должна быть такой, как изображена на рис. 1.5. Настройте проект приложения таким образом, чтобы в модуле `rational_poly.cpp` можно было подключать заголовочный файл библиотеки директивой:

```
#include "lib_rational.h"
```

вместо:

```
#include "../lib_rational/lib_rational.h"
```

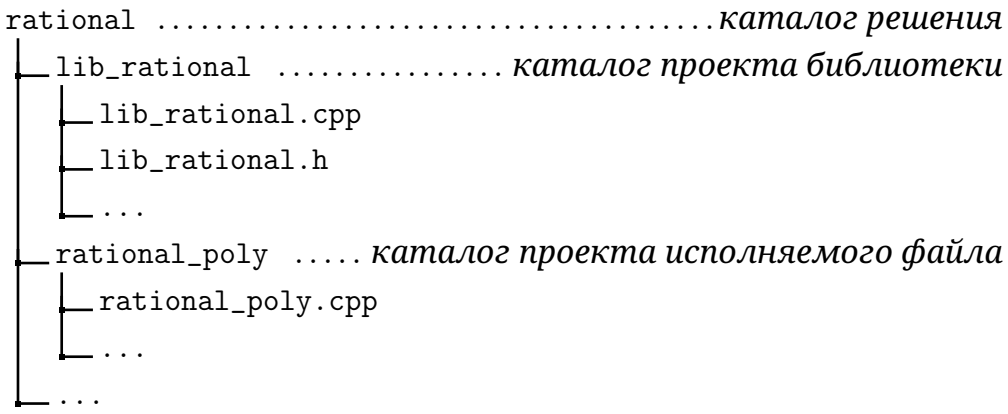

Рис. 1.5. Структура проекта задания

a) Операции с рациональными числами: сокращение, сложение, вычитание, умножение, деление, сравнение.

b) Для заданного многочлена с рациональными коэффициентами:

$$P_n(q) = a_0 q^n + a_1 q^{n-1} + \ldots + a_{n-1} q + a_n, \qquad a_i, q \in \mathbb{Q}$$

и двух чисел $q_1, q_2 \in \mathbb{Q}$ определить, какое из значений больше: $P_n(q_1)$ или $P_n(q_2)$. Вычисление значения многочлена производить по схеме Горнера.

2. Попробуйте в предыдущем задании реализовать построение вне каталога проекта, устанавливая соответствующие настройки в интегрированной среде.

2. Основы языка CMake

Эта глава посвящена изучению инструмента CMake. Будут рассмотрены концепции этой системы, синтаксис языка и основные команды вместе с простыми примерами их использования.

2.1. Основные концепции

2.1.1. Генераторы

Генераторами называются компоненты инструмента CMake, отвечающие за создание проектов для конечных систем построения. Для каждой поддерживаемой системы (make, Microsoft Visual Studio определённой версии и т. д.) существует свой генератор. Выбор конкретного генератора задаётся в командной строке CMake при помощи ключа «`-G`», например:

```
cmake -G "Visual Studio 14 2015" ..\project_src
```

Здесь последний аргумент (`..\project_src`) указывает утилите CMake с интерфейсом командной строки (`cmake`) путь к каталогу проекта. Текущий каталог, откуда запускается про-

грамма CMake, будет каталогом построения, в котором будут сгенерированы файлы проектов.

Реализации CMake для различных платформ могут поддерживать разные наборы генераторов. Полный список имён поддерживаемых генераторов для данной версии можно узнать при помощи команды:

```
cmake --help
```

Кроме этого, некоторые генераторы поддерживают различные варианты в виде наборов инструментов (ключ «-T») и платформ (ключ «-A»). Для совместимости с предыдущими версиями инструмента CMake возможно указание платформы в названии генератора. Например, следующие пары команд эквивалентны:

```
cmake -G "Visual Studio 11 2012" -A ARM ..\project_src
cmake -G "Visual Studio 11 2012 ARM" ..\project_src
```

```
cmake -G "Visual Studio 11 2012" -A x64 ..\project_src
cmake -G "Visual Studio 11 2012 Win64" ..\project_src
```

При отсутствии ключа «-G» в командной строке CMake выбирается некоторый генератор по умолчанию для данной платформы.

Замечание: в именах генераторов для среды Microsoft Visual C++ можно опускать номер года. Например, последнюю приведённую выше команду можно сократить до следующего вида:

```
cmake -G "Visual Studio 11 Win64" ..\project_src
```

▲

2.1.2. Входные файлы

Входными файлами на языке CMake являются файлы `CMakeLists.txt`, лежащие в корне каталога проекта (с файлами исходных кодов). Путь к каталогу проекта передаётся инструменту CMake в качестве последнего аргумента командной строки. Крупные проекты могут также содержать подкаталоги с проектами нижнего уровня, каждый из которых будет иметь в корне свой файл `CMakeLists.txt`. Каталоги с подпроектами подключаются из файла на языке CMake при помощи команды `add_subdirectory()` (п. 2.5.3). Для каждого обрабатываемого файла `CMakeLists.txt` система CMake создаёт в выходном каталоге подкаталог, служащий по умолчанию каталогом для промежуточных и выходных файлов этого (под)проекта, генерируя в нём все необходимые файлы для используемой конечной системы построения (make, интегрированные среды разработки, и т. д.).

Кроме файлов `CMakeLists.txt` также могут использоваться файлы с расширением «`.cmake`». Эти файлы могут подключаться из программы на языке CMake при помощи директивы `include()` (*модули*, п. 2.4.3), вызываться при исполнении команды `find_package()` (*модули поиска* или *конфигурационные файлы*, п. 2.11.1) или передаваться инструменту CMake для исполнения при помощи ключа командной строки «`-P`» (*сценарии*). При исполнении этих файлов система CMake не создаёт

отдельных выходных каталогов и не генерирует файлов для построения.

В процессе генерирования файлов описания проектов CMake определяет в них зависимости от входных файлов (`CMakeLists.txt` и подключаемые модули). Изменения в них приведут при запуске построения конечной системой к повторному вызову инструмента CMake, который перегенерирует описания проектов. Таким образом, автоматически отслеживаются ситуации изменения входных файлов CMake.

Начиная с версии CMake 3.2 официально поддерживаемой кодировкой для входных файлов является UTF-8. Благодаря этому в программе для CMake можно, например, указывать пути, содержащие символы Unicode. Работоспособность других кодировок (например, восьмибитной Windows-1251) не гарантируется. Кодировка ASCII, которая содержит латинские буквы, совместима с UTF-8 в прямом направлении. Поэтому входные файлы CMake, в которых используются только символы ASCII, не должны вызывать проблем.

2.1.3. Пути

Следует обратить особое внимание на тот факт, что система CMake использует абсолютные пути везде, где это возможно. В генерируемых файлах проектов для конечных систем построения содержатся абсолютные пути к исходным, промежуточным и конечным файлам. Определяемые пользователем пути поиска заголовочных файлов при генерировании преобразуются в абсолютные и т. д. Специальные переменные (п. 2.2.3), храня-

щие пути к файлам и каталогам (такие как `CMAKE_SOURCE_DIR`), содержат абсолютные пути. На платформах, файловые системы которых нечувствительны к регистру имён файлов, символы путей преобразуются к тому регистру, в котором они хранятся в системе.

Такой подход ускоряет и упрощает процесс генерирования файлов для конечных систем построения. Например, для проверки того, указывают ли два пути на один и тот же файл или каталог, достаточно проверить эти строки на равенство (см. пример на с. 139). Кроме этого, генерируемые файлы реализуют требуемые от них функции построения более надёжным образом. Например, в make-файлах указываются абсолютные пути к вызываемым инструментам, что делает процесс построения независимым от значения переменной окружения `PATH` и других общесистемных настроек.

Однако использование абсолютных путей также накладывает на разработчика определённые обязательства. Так, при перемещении каталога проекта или каталога построения в файловой системе приходится заново генерировать файлы повторным запуском инструмента CMake. Понимание принципа использования абсолютных путей позволяет избежать многих типичных ошибок при использовании CMake в качестве системы организации построения проекта.

2.2. Синтаксис

2.2.1. Команды

Программа на языке CMake состоит из последовательности команд, исполняемых интерпретатором. Синтаксис команды:

⟨*имя_команды*⟩([⟨*аргумент*$_1$⟩ . . . ⟨*аргумент*$_n$⟩])

Замечание: аргументы команд разделяются символами-разделителями, а не запятыми. ▲

Как и в большинстве современных языков программирования, в качестве *символов-разделителей* могут использоваться один или несколько пробелов, символов табуляции и перевода строки. Таким образом, в отличие от языков make, qmake, Bourne Shell и т. д., при записи длинной команды в несколько строк не нужно завершать каждую из них символом «\».

Языком поддерживаются однострочные комментарии, начинающиеся с символа «#». Также, начиная с версии CMake 3.0, поддерживаются блочные комментарии между символами «#[[» и «]]».

Команды являются единственным средством в языке, при помощи них, например, записываются все управляющие конструкции.

Пример

```
if(WIN32)
```

```
  message(STATUS "Building for Windows")
endif()
```

*

Замечание: все конструкции, включая `endif()` в примере выше, являются командами, поэтому после них необходимо ставить скобки. ▲

Как правило, команды поддерживают переменное количество аргументов.

Имена команд нечувствительны к регистру, в настоящем учебнике мы будем использовать для их записи строчные буквы и символ подчёркивания.

2.2.2. Строки

Аргументами команд являются строки. Как и в языках Bourne Shell, make и т. д., строки представляют собой произвольные последовательности символов (кроме специальных), поддерживаются активные символы (п. 2.2.3) и escape-последовательности. Чтобы включить в строку символы-разделители, можно заключить её в двойные кавычки. В этом случае интерпретатор будет считать данные символы частью строки, а не разделителями аргументов. Строки в кавычках могут занимать несколько строк в тексте программы, при этом символ «\» в конце строки текста приводит к тому, что следующий за ним символ перевода строки не будет считаться частью строки.

Пример

Команда вывода сообщения (п. 2.4.4):

```
message(
  "AB\nCD
  \tEF\
  GH \${DATA}\"\\#"     # строка
  )
```

выведет на консоль следующую строку:

```
AB
CD
        EF  GH ${DATA}"\#
```

— Команда выполняет первый перевод строки после символов «AB», так как в строковой константе после них указана escape-последовательность «\n», означающая символ перевода строки, как и в языках C++ и т. д.

— Следующий перевод строки после символов «CD» указан в строковой константе явно.

— Следующая строка начинается с двух пробелов и символа табуляции («\t»), символ перевода строки после «EF» подавляется стоящим перед ним escape-символом «\». Таким образом, после символов «EF» на печать выводятся два пробела и символы «GH».

— За символами «GH» выводится пробел, так как он считается частью строки, ограниченной двойными кавычками. Если

бы кавычек не было, пробел считался бы разделителем аргументов команды `message()` и на печать не был бы выведен.

— Дальше выводятся символы «`${DATA}`». Если бы в строковой константе перед символом «`$`» не было escape-символа «`\`», конструкция «`${DATA}`» внутри строковой константы считалась бы ссылкой на переменную с именем `DATA` и была бы заменена её значением (п. 2.2.3).

— Далее выводится символ «`"`». Поскольку в строковой константе перед ним стоит escape-символ «`\`», он считается частью строки, а не завершающей её двойной кавычкой.

— Затем на печать выводится символ «`\`». Так как в языке CMake он является escape-символом, для его включения в строковую константу его нужно дополнить им же.

— В конце выводится символ «`#`». Так как он указан внутри строковой константы, ограниченной двойными кавычками, он считается её частью, а не началом комментария.

— Так как следующий символ «`#`» встречается в программе уже после завершения строковой константы и вне двойных кавычек, он считается началом комментария.

*

Хотя все данные в CMake имеют строковый тип, в некоторых случаях строковые значения служат в качестве данных других типов. Так, строки используются для представления чисел

в десятичной записи — как целых, так и с плавающей точкой. Формат записи аналогичен другим языкам программирования.

При использовании управляющих конструкций необходим способ представления логических значений. Для этого могут быть использованы различные строковые значения (табл. 2.1).

Таблица 2.1

Строковые значения, соответствующие логическим константам

«Истина»	«Ложь»
`TRUE`	`FALSE`
`YES`	`NO`
`Y`	`N`
`ON`	`OFF`
Десятичная запись ненулевого числа	`0`
	`IGNORE`
	`NOTFOUND`
	Любая строка, заканчивающаяся на «`-NOTFOUND`»
	`""` (пустая строка)

Запись всех именованных логических констант нечувствительна к регистру. Значения, заканчивающиеся на «`-NOTFOUND`», можно использовать для проверки результатов успешности поиска файлов, библиотек и т. д. при помощи команд `find_file()`, `find_library()` (п. 2.9.2) и т. д., которые записывают путь к най-

денному объекту в заданную переменную. Если поиск закончился неудачей, эти команды записывают в переменную её имя, дополненное суффиксом «`-NOTFOUND`».

Другим типом данных, поддерживаемым языком CMake, является список. Списки представляются при помощи обычных строк, внутри которых символы «`;`» («точка с запятой») разделяют соседние элементы. Перебор элементов списка можно выполнять при помощи команд `foreach()` и `endforeach()` (п. 2.8). При передаче списков в команды в качестве аргументов их элементы передаются внутрь команд несколькими аргументами вместо одного. Чтобы заставить команду воспринимать список как один аргумент, его можно поместить в двойные кавычки.

Пример

Следующий фрагмент кода:

```
message(a;b;c)
message("a;b;c")
```

выведет на печать следующие строки:

```
abc
a;b;c
```

Здесь первая команда `message()` (п. 2.4.4) получит три аргумента: «`a`», «`b`» и «`c`». Команда выводит свои аргументы друг за другом без пробелов. Вторая команда `message()` получит один аргумент: строку «`a;b;c`». *

2.2.3. Переменные

Так же как и в языках Bourne Shell и др., в языке CMake есть поддержка переменных строкового типа. Значения переменных могут использоваться внутри аргументов команд при помощи конструкции «ссылка на переменную»: `${VAR_NAME}`.

Пример

```
set(GREETING Hello)
message("${GREETING} world!")
```

Здесь переменной `GREETING` присваивается значение «`Hello`», после чего выводится сообщение «`Hello world!`». *

Кроме пользовательских переменных в CMake также существует множество встроенных, имеющих специальное назначение (*специальные переменные*). Некоторые из этих переменных будут рассмотрены далее по мере необходимости.

Имена переменных чувствительны к регистру и могут состоять почти из любых символов. В настоящем учебнике мы будем использовать для них прописные буквы, цифры и символ подчёркивания.

Замечание: при обработке проектов система CMake создаёт переменные с именами вида ⟨*имя_проекта*⟩`_SOURCE_DIR` (с. 75). Более свободные по сравнению с другими языками программирования правила именования переменных позволяют накладывать минимальные ограничения на названия проектов. Например, имя `01-hello_SOURCE_DIR` является допустимым, хотя лек-

сический анализатор большинства других языков мог бы воспринять его как арифметическое выражение. ▲

Как и в других языках программирования, переменные имеют собственную *область действия*. Области действия определяются следующими сущностями:

Функция. Функции в языке CMake определяют пользовательские команды, которые могут использовать локальные переменные. Функции определяются при помощи команд `function()` и `endfunction()` (п. 2.8.4).

Каталог (под)проекта. Каждый обрабатываемый каталог исходных файлов со своим описанием `CMakeLists.txt` задаёт собственную область действия переменных. Перед началом обработки подпроекта с собственным файлом `CMakeLists.txt` система CMake копирует в новую область все переменные из родительской области. Сценарии, запускаемые командой «`cmake -P ...`» (п. 2.1.2), также задают область действия.

Кэш. Некоторые переменные могут быть определены в программе как предназначенные для постоянного хранения в файле `CMakeCache.txt`, записываемом в выходной каталог. Эти переменные будут сохранять свои значения между повторными запусками инструмента CMake (до момента удаления файла `CMakeCache.txt`). Значения переменных можно редактировать, например, при помощи графической утилиты `ccmake` из состава CMake. Переменные кэша предназначены для хранения изменяемых настро-

ек построения проекта. Первый запуск CMake заполняет их некоторыми значениями по умолчанию, последующие запуски используют уже значения, отредактированные пользователем.

Замечания:

— В отличие от подпроектов, подключаемых командой `add_subdirectory()` (п. 2.5.3), модули CMake, которые подключаются командой `include()` (п. 2.4.3), отдельной области действия переменных не образуют.

— Область действия также относится и к специальным переменным. То есть их значения и действие ограничиваются текущей областью. Таким образом, статус «специальности» переменной определяется её именем. При этом в разных областях действия могут существовать разные специальные переменные с одним и тем же именем.

— Переменные, которые создаются в командной строке при вызове инструмента CMake с помощью ключа «`-D`» (см. пример на с. 68), попадают в кэш. ▲

При исполнении команд каждая найденная ссылка на переменную (`${VAR_NAME}`) заменяется значением этой переменной. Поиск значения выполняется от вершины в глубину стека областей действия переменных, образуемого динамически вложенными вызовами функций. Последней просматривается область действия текущего интерпретируемого файла `CMakeLists.txt` (если переменная не была найдена ранее), а по-

сле неё — область кэша. Если переменная нигде не найдена, ссылка на неё заменяется пустой строкой.

Для установки значения переменных в языке CMake существует ряд команд (п. 2.7).

Специальная конструкция вида `ENV{`⟨*имя_переменной*⟩`}` может быть использована для доступа к переменной окружения работающего процесса CMake. Она может использоваться везде, где можно применять обычные переменные (не кэша). Доступ к значению переменной окружения осуществляется при помощи конструкции `$ENV{`⟨*имя_переменной*⟩`}`.

Пример

```
set(ENV{PATH} "$ENV{PATH};${CMAKE_BINARY_DIR}")
```

Здесь при помощи команды `set()` (п. 2.7.1) к переменной окружения `PATH` добавляется через точку с запятой путь к каталогу построения проекта верхнего уровня, который хранится в специальной переменной `CMAKE_BINARY_DIR`. Переменная окружения `PATH` в различных системах содержит список каталогов, в которых осуществляется поиск, в частности, исполняемых файлов при указании их имени в командной строке (в формате «`cmd_name`», т. е. без пути в начале). *

Замечания:

— В разных системах пути в переменной окружения `PATH` могут отделяться друг от друга разными символами. В POSIX-совместимых системах для этого используется двоеточие.

— Выполненное в примере выше изменение переменной окружения, унаследованной процессом CMake от родителя, будет действительно только для этого процесса на время его работы. Оно не повлияет на значение этой переменной для родительского процесса или процессов, которые будут в дальнейшем выполнять построение проекта. Чтобы установить значения переменных окружения для инструментов, которые запускаются во время исполнения правил, определяемых командами `add_custom_target()` (п. 2.10.4) и `add_custom_command()` (п. 2.10.5), можно воспользоваться командой «`cmake -E env ...`» (см. также пример и замечание к нему на с. 223).

— CMake выдвигает те же требования к именам переменных окружения, что и для своих переменных. В частности, в именах переменных недопустимы скобки. Это затрудняет использование переменной окружения `ProgramFiles(x86)`, определённой в 64-битных версиях системы Windows. Пример на с. 192 демонстрирует, как можно обойти это ограничение. ▲

2.2.4. Свойства

Свойства (properties) аналогичны переменным, но относятся к различным объектам:

— каталогам проектов (как каталогам верхнего уровня, передаваемым CMake в качестве аргумента командной строки, так и подключаемым при помощи команды `add_subdirectory()` (п. 2.5.3));

— целям;

— тестам (п. 2.10.2);

— исходным файлам;

— переменным, хранящимся в кэше (п. 2.2.3);

— файлам для установки (п. 2.10.3).

Также существуют глобальные свойства. Как и переменные, свойства могут быть стандартными (имеющими предопределённое назначение). Изменение таких свойств приводит к определённым изменениям в генерируемой системе построения. В дальнейшем стандартные свойства будут рассматриваться по мере необходимости. Некоторые стандартные свойства доступны только на чтение. Таким образом, можно провести аналогию между свойствами CMake и свойствами классов в объектно-ориентированных языках программирования.

Для считывания и установки значений свойств используются команды `get_property()` и `set_property()` (п. 2.11.2).

Замечание: в отличие от переменных, которые имеют области действия, свойства всегда связаны с объектами, для которых они определены. Например, глобальное свойство с заданным именем всегда существует в единственном экземпляре, в то время как одновременно может существовать сразу несколько переменных с одним и тем же именем в разных областях действия. ▲

2.2.5. Регулярные выражения

Некоторые из команд CMake (`string()`, п. 2.7.4, `if()`, п. 2.8.1, `install()`, п. 2.10.3) имеют возможность работы с регулярными выражениями. В CMake поддерживается язык регулярных выражений, аналогичный многим утилитам POSIX (табл. 2.2).

Таблица 2.2

Элементы регулярных выражений

Строка	Описание
`^`	Соответствует началу строки
`$`	Соответствует концу строки
`.` (точка)	Соответствует одному любому символу
`[`⟨*символы*⟩`]`	Соответствует одному любому символу из перечисленных в скобках
`[^`⟨*символы*⟩`]`	Соответствует одному любому символу, которого нет в скобках
⟨*символ*⟩`-`⟨*символ*⟩	Внутри скобок определяет диапазон символов между находящимися слева и справа от «-»: «`[a-d]`» эквивалентно «`[abcd]`». Если символ «-» граничит со скобками, он рассматривается как литерал. Например: «`[-+]`» соответствует «-» или «+»

Окончание табл. 2.2

Строка	Описание
⟨*образец*⟩*	Соответствует образцу 0 или больше раз
⟨*образец*⟩+	Соответствует образцу 1 или больше раз
⟨*образец*⟩?	Соответствует образцу 0 или 1 раз
⟨*образец*⟩|⟨*образец*⟩	Соответствует образцу слева или справа от «|»
(⟨*образец*⟩)	Используется для явного указания порядка операций. Кроме этого, сохраняет соответствующую строку для дальнейшего использования в *выражении замены* (п. 2.7.4) и в специальной переменной `CMAKE_MATCH_1` и т. д.

— Операции повтора образца («*», «+» и «?») имеют более высокий приоритет по сравнению с конкатенацией. Конкатенация (в теории формальных грамматик) — это следование обычных символов (не представленных в табл. 2.2) и образцов, она обозначается пустой строкой (как операция умножения в арифметике). Таким образом, регулярное выражение «`aX*`» следует понимать как «`a(X*)`», а не «`(aX)*`». То есть ему соответствует строка «`aXXX`», но не «`aXaX`».

— Операция дизъюнкции образцов («|»), наоборот, имеет более низкий приоритет по сравнению с конкатенацией. Так, выражение «ab|cd» следует понимать как «(ab)|(cd)», а не «a(b|c)d». Ему соответствуют строки «ab» и «cd», но не «abd» и «acd».

Специальная переменная CMAKE_MATCH_0 содержит последнюю подстроку, распознанную как соответствующую регулярному выражению любой командой CMake. Аналогично, переменные CMAKE_MATCH_1, CMAKE_MATCH_2 и т. д. содержат подстроки, соответствующие группирующим подвыражениям (частям регулярных выражений в круглых скобках). Начиная с версии 3.2 CMake также поддерживает переменную CMAKE_MATCH_COUNT, которая принимает значение максимального номера n заполненной последним сопоставлением переменной CMAKE_MATCH_$\langle n \rangle$.

Замечания:

— Регулярные выражения имеют свойство «жадности». Это значит, что при выделении командой string() подстрок, соответствующих заданному регулярному выражению, выделяется наиболее длинная подстрока, насколько это возможно. Например, в строке «---ab-cd-cd---» в соответствии с выражением «ab.*cd» будет выделена подстрока «ab-cd-cd», а не «ab-cd».

— CMake поддерживает меньший набор возможностей по сравнению с регулярными выражениями POSIX[1]. Например, конструкции, ссылающиеся на распознанные

[1] http://en.wikibooks.org/wiki/Regular_Expressions (дата обращения: 16.02.2015).

подстроки («\1», «\2» и т. д.), допустимы только в выражениях замены команды `string(REGEX REPLACE ...)` (п. 2.7.4), но не в том же самом регулярном выражении. Образцы повтора после скобок («(aX)*») в CMake не имеют смысла. ▲

Примеры использования регулярных выражений приведены в описании команды `string()` (п. 2.7.4).

2.3. Примеры простых проектов

Прежде чем приступать к изучению основных команд языка CMake, рассмотрим несколько простых примеров его использования.

Пример

Пусть требуется построить проект из одного исходного файла, в результате чего должен быть создан исполняемый модуль. Рассмотрим возможную структуру каталогов проекта (рис. 2.1). Здесь в некоторой рабочей папке находятся каталог исходных файлов проекта (`test_cmake`) и каталог построения (`_build`). В этом примере оба каталога расположены рядом друг с другом для упрощения команды вызова инструмента CMake, в реальности эти каталоги могут располагаться совершенно в разных местах.

Для решения задачи в каталоге проекта должен содержаться файл `CMakeLists.txt` со следующим возможным содержанием:

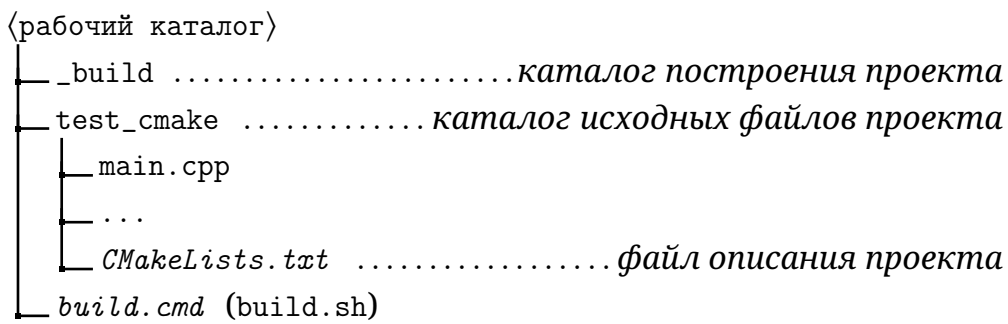

Рис. 2.1. Структура каталога простого проекта

```
# CMakeLists.txt - описание простого проекта

project(test_cmake)
add_executable(test_cmake main.cpp)
```

Первая строка файла является комментарием. Дальше следует команда `project()`, задающая имя проекта. Это имя будет использовано при создании системы построения в качестве имени решения или основного проекта.

Последняя строка содержит команду `add_executable()`, которая определяет новую цель построения с именем `test_cmake` и исходным файлом `main.cpp`. По умолчанию построение цели должно привести к созданию исполняемого модуля с тем же именем, что и имя цели (`test_cmake`) в выходном каталоге проекта. На платформе Windows имя файла будет дополнено расширением «`.exe`».

Если в дальнейшем будет необходимо добавить в проект другие исходные файлы, их имена можно добавить в список аргументов команды `add_executable()`.

Для удобства можно создать файл сценария `build.cmd` в ОС Windows (`build.sh` в системах на основе Linux, OS X и т. д.) рядом с каталогом `_build`, при помощи которого можно автоматизировать процесс построения вызовом инструмента CMake с нужными аргументами. Содержимое этого файла является специфичным для компьютера, на котором выполняется построение, поэтому нет смысла помещать его в каталог исходных файлов.

```
mkdir _build
cd _build
set PATH=C:\Program Files\CodeBlocks\MinGW\bin;%PATH%
cmake -G "CodeBlocks - MinGW Makefiles" ../test_cmake
```

В первой строке исполняется команда `mkdir`, создающая каталог построения, если его не существует. Следующая команда (`cd`) выбирает каталог построения в качестве текущего. Следующая за ней команда добавляет в начало системной переменной окружения `PATH` путь к исполняемым файлам компилятора MinGW, входящего в состав дистрибутива интегрированной среды Code::Blocks. Это необходимо для того, чтобы инструмент CMake мог найти нужные файлы инструментов компилятора MinGW и сгенерировать рассчитанные на них файлы построения.

В последней строке вызывается сам инструмент CMake. Ему передаётся два аргумента: имя генератора при помощи ключа «`-G`» (здесь генерируется файл проекта для среды Code::Blocks,

использующий утилиту построения `mingw32-make`) и путь к каталогу проекта, в котором расположен файл `CMakeLists.txt`.

Таким образом, каталог `_build` можно удалить в любой момент. Для повторного построения достаточно запустить командный файл `build.cmd` и обработать созданный проект при помощи среды Code::Blocks (рис. 2.2).

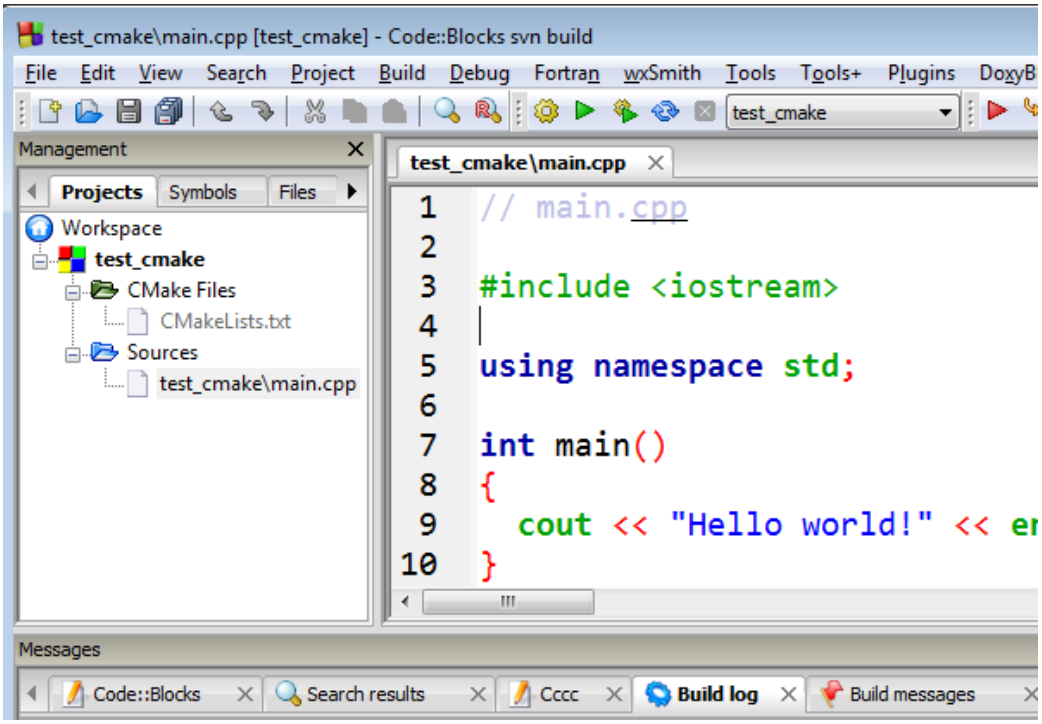

Рис. 2.2. Сгенерированный проект, открытый в среде Code::Blocks

Похожим образом можно написать аналогичный сценарий `build.sh` для операционных систем на основе Linux, FreeBSD и т. п.

Пример

Пусть необходимо построить проект, содержащий две цели: статическую библиотеку и приложение, использующее эту библио-

теку. Пусть файлы в каталоге проекта организованы так, как изображено на рис. 2.3.

```
⟨рабочий каталог⟩
├─ _build_lib ..... каталог построения проекта
├─ lib ..... каталог исходных файлов проекта
│  ├─ sample_lib ..... подкаталог проекта библиотеки
│  │  ├─ sample_module.cpp
│  │  ├─ sample_module.h
│  │  └─ CMakeLists.txt ..... описание проекта библиотеки
│  ├─ sample_program .. подкаталог проекта исполняемого файла
│  │  ├─ main.cpp
│  │  └─ CMakeLists.txt ..... описание проекта приложения
│  └─ CMakeLists.txt ..... описание проекта верхнего уровня
└─ build_lib.cmd (build_lib.sh)
```

Рис. 2.3. Структура каталога проекта с библиотекой

Здесь исходные файлы библиотеки и приложения расположены в двух подкаталогах проектов нижнего уровня, каждый из которых имеет собственный файл описания `CMakeLists.txt`. Оба проекта объединяются третьим файлом `CMakeLists.txt`, расположенным на каталог выше, в корне составного проекта. Эти файлы могут иметь следующее содержание:

Файл `CMakeLists.txt` верхнего уровня:

```
cmake_minimum_required(VERSION 2.8)

project(lib)

add_subdirectory(sample_lib)
add_subdirectory(sample_program)
```

Файл CMakeLists.txt проекта библиотеки:

```
add_library(
  sample_lib sample_module.cpp sample_module.h)
```

Файл CMakeLists.txt проекта исполняемого файла:

```
add_executable(sample_program main.cpp)

include_directories(../sample_lib)
target_link_libraries(sample_program sample_lib)
```

Описание проекта верхнего уровня начинается с команды cmake_minimum_required() (п. 2.4.1), устанавливающей наименьшую версию инструмента CMake, которым может быть обработан проект. Далее устанавливается название проекта при помощи команды project() (п. 2.4.2). Затем к проекту присоединяются два проекта нижнего уровня при помощи команд add_subdirectory() (п. 2.5.3), задающих пути к подкаталогам.

Описание проекта библиотеки содержит единственную команду add_library() (п. 2.5.2), в которой задаются имя цели и список исходных файлов проекта. Сборка проекта должна привести к созданию статической библиотеки в выходном каталоге проекта sample_lib (подкаталога sample_lib выходного каталога проекта верхнего уровня). Имя файла библиотеки будет по умолчанию совпадать с именем цели sample_lib с префиксом и расширением, зависящими от компилятора. Например, для компилятора gcc имя библиотеки будет libsample_lib.a.

Описание проекта приложения отличается от предыдущего примера добавлением двух новых команд. Команда

include_directories() (п. 2.6.1) определяет дополнительные пути поиска заголовочных файлов компилятором. В подкаталоге sample_lib находится заголовочный файл sample_module.h с описаниями, необходимыми для использования библиотеки. Исполнение команды include_directories() приводит к тому, что в генерируемом проекте команда для компиляции исходных файлов проекта sample_program будет содержать аргумент командной строки, передающий компилятору путь к каталогу sample_lib. Например, для компилятора gcc это будет ключ «-I» с путём:

```
gcc ... -I ⟨полный_путь_к_sample_lib⟩ ...
```

Таким образом, в исходных файлах проекта приложения можно использовать следующую директиву:

```
#include "sample_module.h"
```

вместо:

```
#include "../sample_lib/sample_module.h"
```

Наконец, команда target_link_libraries() (п. 2.6.7) определяет зависимость цели sample_program от sample_lib. Кроме того, команда устанавливает, что при построении цели sample_program будет использоваться библиотека, создаваемая в результате построения цели sample_lib.

В завершение осталось рассмотреть вопрос о том, каким образом можно указать системе CMake, что при построении цели sample_lib необходимо создавать статическую библиотеку,

как требуется по условию. Дело в том, что по умолчанию команда `add_library()` может приводить к генерированию правил для создания динамической (разделяемой) библиотеки (тип библиотеки по умолчанию зависит от операционной системы и версии CMake). Сборки цели в виде статической библиотеки можно добиться тремя способами:

1) Можно при вызове инструмента CMake в командной строке определить значение специальной переменной `BUILD_SHARED_LIBS` как `FALSE`:

```
cmake -G "..." -D BUILD_SHARED_LIBS=0 ../lib
```

   Здесь в качестве значения переменной можно указать любую строку, обозначающую «ложь» (см. табл. 2.1). Таким образом, тип библиотеки можно определять в командной строке.

2) Если необходимо построение статической библиотеки в любом случае, можно упростить команду вызова инструмента CMake, выполнив нужное присваивание прямо в тексте описания проекта:

```
set(BUILD_SHARED_LIBS FALSE)
```

   Эту команду можно вставить в файл `CMakeLists.txt` проекта верхнего уровня или проекта библиотеки. Она повлияет на поведение всех команд `add_library()`, исполняемых после неё в текущей области действия переменной `BUILD_SHARED_LIBS` (п. 2.2.3).

3) Наконец, тип собираемой библиотеки можно указать прямо в команде `add_library()`, передав ей аргумент `STATIC`:

```
add_library(
  sample_lib STATIC
  sample_module.cpp
  sample_module.h)
```

Эта настройка будет влиять только на данную цель. *

2.4. Команды общего назначения

2.4.1. `cmake_minimum_required()`

`cmake_minimum_required`(`VERSION` ⟨*версия*⟩ [`FATAL_ERROR`])

Команда определяет минимальную версию инструмента CMake, при помощи которого можно обрабатывать текущий входной файл проекта. Аргумент ⟨*версия*⟩ может содержать до 4 чисел, разделённых точками.

Если версия инструмента CMake окажется ниже заданной, обработка файла остановится с сообщением об ошибке.

— Необязательный аргумент `FATAL_ERROR` имеет значение для версии CMake 2.4 или ниже — с ним команда приведёт к останову с ошибкой вместо обычного предупреждения.

Команда в соответствии с номером версии также устанавливает *политики совместимости* — набор аспектов поведения

инструмента CMake, делающий его совместимым с предыдущей версией. Множество аспектов поведения инструмента CMake может меняться от версии к версии. Явное указание политик совместимости желательно, поскольку обеспечивает корректную обработку описаний проектов при переходе на новую версию CMake.

Команду `cmake_minimum_required()` рекомендуется указывать самой первой в файле `CMakeLists.txt` верхнего уровня. Если команда ещё не исполнялась и инструменту CMake требуется выполнить действие, зависящее от политики совместимости, он выведет предупреждение.

2.4.2. `project()`

---

project(
⟨*имя_проекта*⟩
[VERSION ⟨*версия*⟩] [LANGUAGES ⟨*язык*$_1$⟩ ... ⟨*язык*$_n$⟩])

Команда устанавливает имя для иерархии проектов. Например, для генераторов Microsoft Visual Studio это имя определяет имя создаваемого решения.

— После необязательного аргумента `VERSION` можно указать версию проекта — строку, содержащую до 4 чисел, разделённых точками (по умолчанию устанавливается пустая строка). Установка версии проекта доступна в CMake начиная с версии 3.0.

— После необязательного аргумента `LANGUAGES` указываются языки программирования, для которых обеспечивается

поддержка в проекте. По умолчанию устанавливаются языки C и CPP, также можно указать Fortran. Если после аргумента LANGUAGES добавить NONE, поддержка всех языков будет отключена. Если в команде project() не определена версия проекта, аргумент LANGUAGES можно пропустить. Передача аргумента LANGUAGES доступна в CMake начиная с версии 3.0.

Команда устанавливает значения нескольким специальным переменным CMake (табл. 2.3).

Таблица 2.3

Некоторые переменные, устанавливаемые командой project()

Имя переменной	Значение
PROJECT_NAME	Имя проекта, определяемое командой
PROJECT_SOURCE_DIR	Полный путь к каталогу проекта верхнего уровня (каталогу исходных файлов)
⟨*имя_проекта*⟩_SOURCE_DIR	То же самое
PROJECT_BINARY_DIR	Полный путь к каталогу построения проекта верхнего уровня (каталогу выходных и промежуточных файлов)
⟨*имя_проекта*⟩_BINARY_DIR	То же самое

Окончание табл. 2.3

Имя переменной	Значение
PROJECT_VERSION	Версия проекта, указанная после аргумента VERSION
⟨*имя_проекта*⟩_VERSION	То же самое

Команду project() рекомендуется указывать в файле CMakeLists.txt верхнего уровня. Она должна быть указана в явном виде (не внутри модулей, подключаемых при помощи команды include(), п. 2.4.3). Если её там нет, система создаёт проект с некоторым именем по умолчанию. Внутри файлов CMakeLists.txt для подпроектов эту команду имеет смысл указывать, если эти проекты вместе с проектами, от которых для них установлены зависимости, могут собираться отдельно от остального набора проектов. В этом случае для подпроектов будут сгенерированы отдельные решения, которые будут обрабатываться интегрированной средой гораздо быстрее полного набора (обычно при открытии решений интегрированные среды собирают информацию обо всех исходных файлах с целью построения информации для быстрого перехода и т. д.).

2.4.3. include()

---

include(
⟨*файл*⟩ | ⟨*модуль*⟩
[OPTIONAL] [RESULT_VARIABLE ⟨*имя_переменной*⟩])

Загружает из внешнего файла и исполняет заданный модуль CMake. Модуль может задаваться при помощи пути к файлу (включая расширение «`.cmake`», относительный путь определяет местонахождение относительно каталога проекта) либо только именем модуля (без пути и расширения). В последнем случае файл с именем ⟨*модуль*⟩`.cmake` ищется в каталогах, список которых задан в переменной `CMAKE_MODULE_PATH` (по умолчанию пустая), и затем в каталоге стандартных модулей, поставляемых вместе с CMake.

Замечание: разница между командой `include()` и командой `add_subdirectory()` (п. 2.5.3) объясняется в п. 2.1.2. ▲

Пример

Пусть требуется загрузить модуль с именем `module.cmake`, находящийся в подкаталоге `build` каталога проекта. Это можно сделать, указав относительный путь к файлу:

```
include(build/module.cmake)
```

Или можно указать только имя модуля, но тогда также необходимо добавить путь к файлу к списку каталогов в переменной `CMAKE_MODULE_PATH` при помощи команды `set()` (п. 2.7.1):

```
set(
  CMAKE_MODULE_PATH
  ${CMAKE_MODULE_PATH}
  "${CMAKE_CURRENT_SOURCE_DIR}/build")
```

include(module)

Здесь в специальной переменной CMake с именем CMAKE_CURRENT_SOURCE_DIR хранится полный путь к текущему обрабатываемому CMake каталогу исходных файлов проекта. *

Замечание: при передаче в команды CMake путей, составленных из подстановок переменных (как в приведённом примере), весь аргумент желательно заключить в кавычки. Это нужно для того, чтобы команда восприняла путь как один аргумент. Если в пути будут пробелы, они будут восприняты командой как разделители соседних аргументов.

Например, пусть при исполнении приведённого выше кода каталог проекта имеет путь, содержащий пробелы (d:/my projects/pr1). Тогда команда set() воспримет последнюю строку как несколько разных аргументов, если она не будет заключена в кавычки. В этом случае команда добавит в конец переменной-списка CMAKE_MODULE_PATH не один путь с пробелом, а несколько несуществующих («d:/my» и «projects/pr1/build»). ▲

— Если указан необязательный аргумент OPTIONAL, исполнение команды include() не приведёт к ошибке, если указанный модуль не будет найден. Это может быть полезным в тех случаях, когда какие-то необязательные для построения компоненты проекта не восстанавливаются из репозитария исходных кодов и не собираются в целях экономии времени.

— После необязательного аргумента `RESULT_VARIABLE` можно указать имя переменной, в которую будет записан полный путь к найденному модулю или значение `NOTFOUND`, если модуль не будет найден. Таким образом, можно программно проконтролировать успешность загрузки модуля.

2.4.4. `message()`

`message``(` [⟨*режим*⟩] ⟨*строка*$_1$⟩ . . . ⟨*строка*$_n$⟩ `)`

⟨*режим*⟩ ::=
`STATUS` | `WARNING` | `AUTHOR_WARNING` | `SEND_ERROR` |
`FATAL_ERROR` | `DEPRECATION`

Команда предназначена для вывода сообщения различными утилитами CMake, интерпретирующими входной файл. Консольная программа `cmake` выводит сообщение в стандартный поток вывода при использовании режима `STATUS` и в стандартный поток ошибок в остальных случаях. Другие утилиты выводят сообщения в окно журнала, в виде диалоговых окон и т. д.

Выводимое сообщение получается в результате конкатенации строк, передаваемых команде.

— При помощи первого аргумента можно указать режим, который влияет на способ отображения сообщения, способ его форматирования, а также дальнейшее поведение инструмента CMake. Он может продолжить или прервать обработку входного файла, а также выполнить или пропу-

стить генерацию файлов для конечной системы построения (табл. 2.4).

Таблица 2.4

Режимы выполнения команды `message()`

Режим	Описание	Обработка	Генерация
(не указан)	Важная информация	✔	✔
`STATUS`	Обычная информация	✔	✔
`WARNING`	Предупреждение	✔	✔
`AUTHOR_WARNING`	Предупреждение для разработчиков	✔	✔
`SEND_ERROR`	Локальная ошибка	✔	✗
`FATAL_ERROR`	Серьёзная ошибка	✗	✗

Окончание табл. 2.4

Режим	Описание	Обработка	Генерация
DEPRECATION	Использование устаревшей возможности	зависит от настроек *	

* При исполнении команды `message(DEPRECATION ...)` поведение системы CMake зависит от значений специальных переменных `CMAKE_ERROR_DEPRECATED` и `CMAKE_WARN_DEPRECATED`. Если первая из них содержит значение «истина» (см. табл. 2.1), поведение аналогично режиму `FATAL_ERROR`, иначе, если вторая содержит значение «истина», — режиму `WARNING`. Иначе обработка файла и генерация продолжаются, и сообщение не выводится. По умолчанию обе переменные содержат значение `FALSE`.

2.5. Команды описания целей

2.5.1. add_executable()

---

add_executable(
⟨*логическое_имя_цели*⟩
[WIN32] [MACOSX_BUNDLE] [EXCLUDE_FROM_ALL]
⟨*исходный_модуль*$_1$⟩ ... ⟨*исходный_модуль*$_n$⟩)

Команда добавляет к проекту цель с заданным логическим именем, построение которой из указанных исходных модулей должно привести к созданию исполняемого файла.

Имя исполняемого файла формируется из имени цели и расширения «.exe» (на платформе Windows). Изменить имя можно также при помощи установки свойства OUTPUT_NAME цели (см. пример на с. 214).

По умолчанию файл должен создаваться в подкаталоге построения, соответствующем текущему обрабатываемому подкаталогу проекта. Изменить этот каталог можно при помощи установки соответствующего свойства цели, которое изначально инициализируется значением специальной переменной CMAKE_RUNTIME_OUTPUT_DIRECTORY. Конечные системы построения, которые поддерживают множественные конфигурации (Micsrosoft Visual Studio, XCode и т. д.), могут добавлять к этому пути ещё один вложенный каталог, соответствующий имени используемой конфигурации (Debug, Release и т. д.).

— Передача команде необязательного аргумента WIN32 приводит к тому, что при построении для платформы Windows приложение не будет иметь создаваемой по умолчанию консоли, даже если главной функцией программы является функция main(), а не WinMain(). Это достигается передачей компоновщику аргумента командной строки, зависящего от компилятора. Например, для системы gcc-MinGW компоновщику передаётся ключ -Wl,--subsystem,windows. Это бывает удобно, в частности, для разработки с использованием библиотек Qt.

При использовании аргумента WIN32 можно обойтись без функции WinMain(), таким образом, упростить переносимость кода. Для остальных платформ этот аргумент игнорируется.

— Передача необязательного аргумента MACOSX_BUNDLE сообщит системе CMake, что создаваемый исполняемый файл должен быть пакетом приложения системы OS X.

— Передача необязательного аргумента EXCLUDE_FROM_ALL приведёт к тому, что генерируемая цель будет исключена из цели «all». Таким образом, например, при работе с системой make команда make или make all приведёт к построению данной цели, только если от неё зависимы другие цели, включённые в цель «all».

Замечание: при использовании компилятора Microsoft Visual C++ передачи аргумента WIN32 команде add_executable() недостаточно, если требуется использовать функцию main() в качестве точки входа. В этом случае также требуется передача компоновщику ключа «/ENTRY:mainCRTStartup» (см. пример на с. 258). ▲

2.5.2. add_library()

---

add_library(
⟨*логическое_имя_цели*⟩
[STATIC | SHARED | MODULE]
[EXCLUDE_FROM_ALL]
⟨*исходный_модуль*$_1$⟩ ... ⟨*исходный_модуль*$_n$⟩)

```
add_library(
  ⟨логическое_имя_цели⟩
  ⟨тип_библиотеки⟩
  IMPORTED)
```

⟨*тип_библиотеки*⟩ ::= SHARED | STATIC | MODULE | UNKNOWN

Первая форма команды `add_library()` аналогична команде `add_executable()` (п. 2.5.1), но создаёт цель для построения библиотеки.

Имя библиотеки будет сформировано из базового имени (по умолчанию соответствующего имени цели): например, «⟨*имя*⟩`.lib`» для компилятора Microsoft Visual C++, «`lib`⟨*имя*⟩`.a`» для gcc и т. д.

По умолчанию библиотека будет создана в подкаталоге построения, соответствующем текущему обрабатываемому подкаталогу проекта. Изменить расположение библиотеки можно при помощи установки соответствующих свойств цели, которые инициализируются значениями специальных переменных (табл. 2.5). Как и в случае с командой `add_executable()`, конечные системы построения могут добавлять к этим путям каталог с именем используемой конфигурации.

Таблица 2.5

Специальные переменные, определяющие выходные каталоги для библиотек

Переменная	Виды библиотек
CMAKE_ARCHIVE_OUTPUT_DIRECTORY	статические (+импорта)
CMAKE_RUNTIME_OUTPUT_DIRECTORY	DLL
CMAKE_LIBRARY_OUTPUT_DIRECTORY	модули, разделяемые

— Тип библиотеки можно задать при помощи необязательного аргумента:

STATIC: статическая;

SHARED: динамическая (разделяемая);

MODULE: разделяемая, предназначенная исключительно для загрузки при помощи функций API (POSIX `dlopen()` и т. п.). Такой тип библиотеки используется для реализации загружаемых модулей (plugins). См. пример определения и использования такой библиотеки на с. 230.

По умолчанию создаются правила для построения разделяемой библиотеки, если переменная BUILD_SHARED_LIBS содержит истинное значение, и статической, если иначе.

Таблица 2.5 нуждается в пояснении. При построении все статические библиотеки помещаются в каталог, определяемый переменной CMAKE_ARCHIVE_OUTPUT_DIRECTORY. Аналогично, все загружаемые модули попадают в каталог, путь к кото-

рому задаётся переменной CMAKE_LIBRARY_OUTPUT_DIRECTORY. Сложнее обстоит дело с разделяемыми библиотеками. Дело в том, что в POSIX-совместимых системах разделяемые библиотеки принято помещать в специальные каталоги — туда же, где находятся и статические библиотеки. При создании процесса динамический загрузчик исполняемого файла будет искать все требуемые для него библиотеки в этих каталогах. Некоторые из этих путей могут быть заданы системой (переменная окружения LD_LIBRARY_PATH и т. д.). Другие из этих путей могут храниться в относительном виде в самом исполняемом файле («rpath»). Если библиотеки будут расположены в другом месте, загрузчик не сможет их найти. Как правило, исполняемые файлы помещаются в каталог с именем bin, а разделяемые библиотеки — в находящийся рядом каталог lib. Таким образом, разделяемые библиотеки помещаются при построении в каталог, путь к которому задаётся переменной CMAKE_LIBRARY_OUTPUT_DIRECTORY.

В системах, совместимых с Windows, динамические библиотеки ищутся загрузчиком прежде всего в тех же каталогах, что и исполняемые файлы. Например, динамическая библиотека может находиться в том же каталоге, что и использующая её программа. По этой причине динамические библиотеки создаются при построении проекта в таких системах в том же каталоге, что и исполняемые модули, т. е. путь к которому находится в переменной CMAKE_RUNTIME_OUTPUT_DIRECTORY. При этом для облегчения процесса подключения динамической библиотеки к использующему её приложению при её постро-

ении также создаётся небольшая статическая библиотека, содержащая информацию об экспортируемых символах динамической (так называемая *библиотека импорта* — import library). Как и другие статические библиотеки, библиотеки импорта помещаются в каталог, путь к которому определяется переменной CMAKE_ARCHIVE_OUTPUT_DIRECTORY.

Пример

Пусть проект включает исполняемый файл и две библиотеки, которые он использует. Пусть структура каталога проекта соответствует рис. 2.4.

```
my_project ..................................... каталог проекта
├── my_library_1 ..................... подкаталог библиотеки 1
│   ├── ...
│   └── CMakeLists.txt ......... описание проекта библиотеки 1
├── my_library_2 ..................... подкаталог библиотеки 2
│   ├── ...
│   └── CMakeLists.txt ......... описание проекта библиотеки 2
├── my_program ............... подкаталог проекта программы
│   ├── ...
│   └── CMakeLists.txt .......... описание проекта приложения
└── CMakeLists.txt ..........описание проекта верхнего уровня
```

Рис. 2.4. Структура каталога проекта с исполняемым файлом и двумя библиотеками

В этом случае для формирования при построении системы выходных каталогов, совместимой с рекомендациями GNU, можно использовать следующее описание для системы CMake:

```
project(my_project)
```

```
set(BINARY_DIR "${CMAKE_BINARY_DIR}")
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY "${BINARY_DIR}/bin")
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY "${BINARY_DIR}/lib")
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY "${BINARY_DIR}/lib")

add_subdirectory(my_library_1)
add_subdirectory(my_library_2)
add_subdirectory(my_program)
```

Здесь специальная переменная `CMAKE_BINARY_DIR` хранит полный путь к каталогу построения проекта верхнего уровня. Вместо неё в этом примере можно использовать переменную `CMAKE_CURRENT_BINARY_DIR`, в которой хранится путь к каталогу построения текущего (под)проекта.

В результате обработки этого примера инструментом CMake и сборки проекта в системе Windows каталог построения будет иметь структуру, изображённую на рис. 2.5, а в POSIX-совместимой системе — как на рис. 2.6. Как можно видеть, в системе Windows все выходные файлы будут помещены в один каталог, так что при отладке приложения загрузчик сможет найти все требуемые ему библиотеки. То же самое будет справедливым и для систем, совместимых с POSIX. *

Вторая форма команды `add_library()` предназначена для добавления к проекту внешней заранее собранной библиотеки (как правило, сторонней). Как и для предыдущей формы команды, создаётся цель с заданным логическим именем, которая по умолчанию имеет область видимости текущего ката-

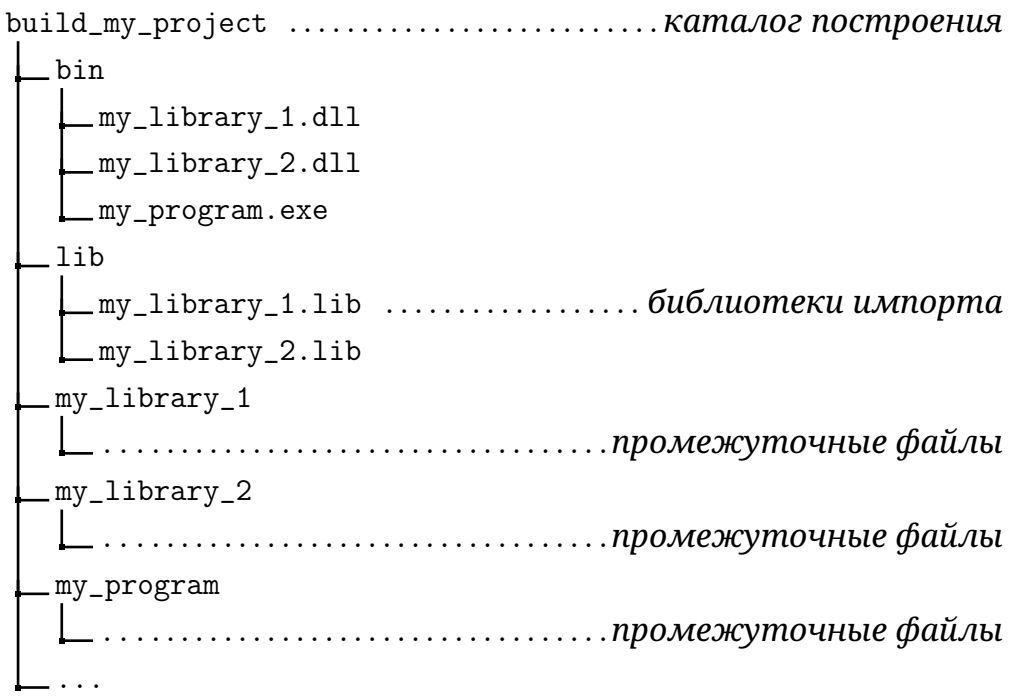

Рис. 2.5. Структура каталога построения в системе Windows

лога построения и ниже и которую можно использовать, как и остальные цели библиотек, для связывания с другими целями проекта при помощи команды `target_link_libraries()` (п. 2.6.7). Однако в этом случае не создаётся никаких правил построения библиотеки. Чтобы указать местоположение файла библиотеки для создаваемой цели, необходимо записать его в свойство цели `IMPORTED_LOCATION`, а также в свойства `IMPORTED_LOCATION_DEBUG` и т. д. для каждой используемой конфигурации (подробнее о конфигурациях см. п. 2.12.1) при помощи команды `set_property()` (п. 2.11.2). Для получения пути к исполняемому файлу библиотеки можно использовать команду `find_library()` (п. 2.9.2).

Вообще говоря, использовать дополнительную команду `add_library(... IMPORTED)` для того, чтобы подключить

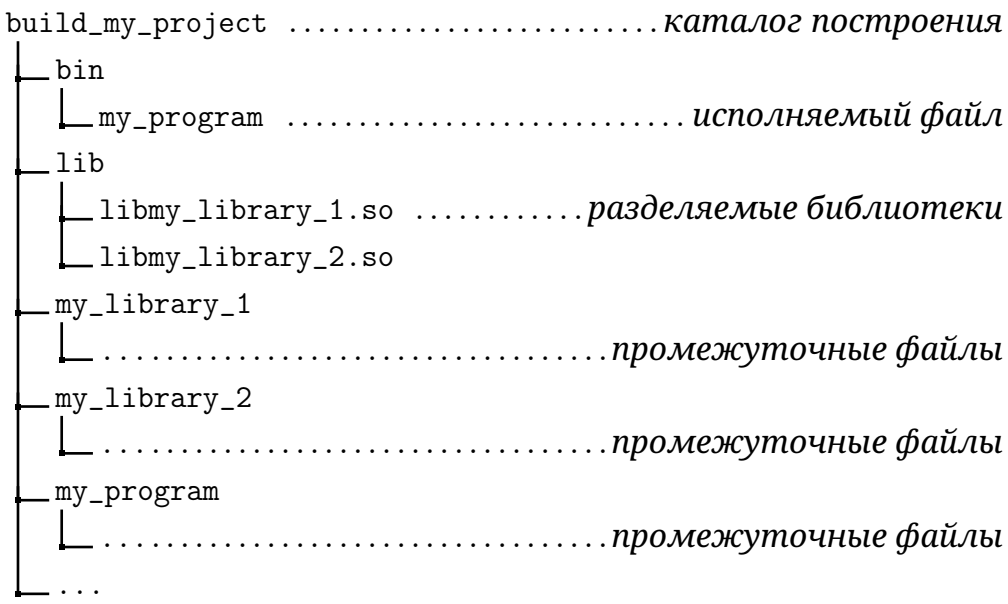

Рис. 2.6. Структура каталога построения в POSIX-совместимой системе

внешнюю библиотеку, необязательно, так как команда `target_link_libraries()` может получать на вход непосредственно пути к файлам библиотек вместо логических имён их целей. Однако команда позволяет существенно упростить повторное использование библиотеки, поскольку для определяемой ею цели можно настроить свойства, используемые при построении зависимых целей (например, каталоги поиска заголовочных файлов). Таким образом, эти свойства не нужно устанавливать заново для каждой цели, к которой подключается библиотека. К сожалению, цель библиотеки, которая определяется этой командой, нельзя передавать первым параметром командам `target_include_directories()` (п. 2.6.2) и т. п. Однако можно устанавливать соответствующие свойства цели командой `set_property()` (п. 2.11.2), что менее удобно, но всё равно не влияет на удобство описания зависимых целей.

Пример подключения библиотеки Crypto++ на с. 350 демонстрирует использование для этой цели вызовов команды `add_library(... IMPORTED)`.

2.5.3. add_subdirectory()

---

add_subdirectory(
⟨*подкаталог_проекта*⟩ [⟨*подкаталог_построения*⟩]
[EXCLUDE_FROM_ALL])

Команда добавляет к построению подпроект, расположенный в заданном подкаталоге. В нём должен находиться файл `CMakeLists.txt` с описанием подпроекта. Имеет смысл указывать относительный путь к каталогу, он будет определён относительно текущего каталога исходных файлов проекта.

— При помощи необязательного аргумента ⟨*подкаталог_построения*⟩ можно указать подкаталог для выходных и промежуточных файлов подпроекта. Так же как и подкаталог проекта, он определяется относительно каталога построения текущего проекта. Если этот аргумент пропущен, в качестве него выбирается такой же относительный путь, что и ⟨*подкаталог_проекта*⟩.

— При помощи передачи команде необязательного аргумента `EXCLUDE_FROM_ALL` можно исключить подпроект вместе со всеми зависимыми проектами из общего построения. При использовании системы make цель будет исключена из цели `all`. Таким образом, команда «`make`»

или «`make all`» будет выполнять построение этого подпроекта только в том случае, если от него зависят другие цели, строящиеся вместе с целью `all`. То же самое относится и к интегрированным средам в случае их использования: подпроект будет включён в общее решение, только если от него зависят другие проекты этого решения. В противном случае имеет смысл использовать команду `project()` (п. 2.4.2) в файле `CMakeLists.txt` подпроекта. Тогда система CMake создаст отдельное решение для этого подпроекта и зависимых от него проектов. Обычно таким образом организуются подпроекты, не обязательные для общего построения (примеры и т. п.).

2.6. Команды настроек целей

2.6.1. include_directories()

---

include_directories(
  [AFTER | BEFORE] [SYSTEM]
  ⟨*каталог*$_1$⟩ ... ⟨*каталог*$_n$⟩)

Команда `include_directories()` добавляет указанные каталоги к списку каталогов, в которых компилятор должен искать заголовочные файлы, подключаемые директивами `#include`. Эти каталоги добавляются к соответствующему свойству всех целей, определённых в текущем файле `CMakeLists.txt`. Относительные пути интерпретируются по отношению к текущему каталогу исходных файлов проекта. При

построении целей эти пути передаются компилятору при помощи аргументов командной строки, зависящих от конкретного компилятора. Например, для gcc каждый путь передаётся при помощи ключа «`-I`».

При построении вне каталога проекта бывает необходимо добавить текущие каталоги исходных и выходных файлов к списку каталогов для поиска подключаемых файлов (значения, содержащиеся в переменных `CMAKE_CURRENT_SOURCE_DIR` и `CMAKE_CURRENT_BINARY_DIR`). Это бывает необходимо, если в выходном каталоге генерируются промежуточные подключаемые файлы. Добавить эти каталоги можно, установив значение специальной переменной `CMAKE_INCLUDE_CURRENT_DIR` в истину (см. также примеры использования библиотеки Qt):

```
set(CMAKE_INCLUDE_CURRENT_DIR ON)
```

— При помощи необязательного аргумента `BEFORE` или `AFTER` можно сообщить команде, что заданные в ней каталоги должны добавляться соответственно в начало или конец списка каталогов для поиска. Если аргумент не указан, каталоги добавляются в начало, если специальная переменная `CMAKE_INCLUDE_DIRECTORIES_BEFORE` установлена в значение истины.

— При помощи необязательного аргумента `SYSTEM` можно сообщить, что указанные каталоги являются системными (относящимися к компилятору). Некоторые компиляторы в таком случае обрабатывают такие каталоги с некоторыми отличиями (не генерируют предупреждений, не созда-

ют информацию о зависимостях для находящихся в них файлов и т. п.).

2.6.2. target_include_directories()

---

target_include_directories(
  ⟨*имя_цели*⟩
  [SYSTEM] [BEFORE]
  INTERFACE | PUBLIC | PRIVATE
    [⟨*каталог*$_{1,1}$⟩ ... ⟨*каталог*$_{1,m}$⟩]
  [
    INTERFACE | PUBLIC | PRIVATE
      [⟨*каталог*$_{2,1}$⟩ ... ⟨*каталог*$_{2,n}$⟩]
    ...
  ])

Команда `target_include_directories()` появилась в версии CMake 2.8.11. Результат её исполнения аналогичен вызову команды `include_directories()` (п. 2.6.1), основные же отличия состоят в следующем:

— В аргументе команды указывается имя цели, к которой она относится. Команда влияет на соответствующие свойства этой цели (п. 2.12.2), в то время как команда `include_directories()` влияет на свойство каталога проекта (т. е. действует на все цели, определяемые в текущем подпроекте).

— Есть возможность указать, будут ли использованы указанные в аргументах команды каталоги для компиляции самой цели или для зависимых от неё целей.

Перед путями к каталогам указывается один из следующих аргументов:

PRIVATE: каталоги будут использованы для поиска заголовочных файлов при компиляции файлов текущей цели (аналогично команде `include_directories()`).

INTERFACE: каталоги будут использованы при компиляции файлов зависимых целей от текущей.

PUBLIC: каталоги будут использованы при компиляции файлов как текущей цели, так и зависимых от неё целей (как будто они описаны одновременно как `PRIVATE` и `INTERFACE`).

Таким образом, команда позволяет упростить подключение библиотек к исполняемым файлам.

Пример

Рассмотрим проект, состоящий из библиотеки и использующего её приложения, аналогичный приведённому в примере на с. 68 (рис. 2.7).

С использованием команды `target_include_directories()` можно реализовать файлы описания проектов следующим образом:

Файл `CMakeLists.txt` проекта верхнего уровня:

```
# CMakeLists.txt для проекта верхнего уровня
```

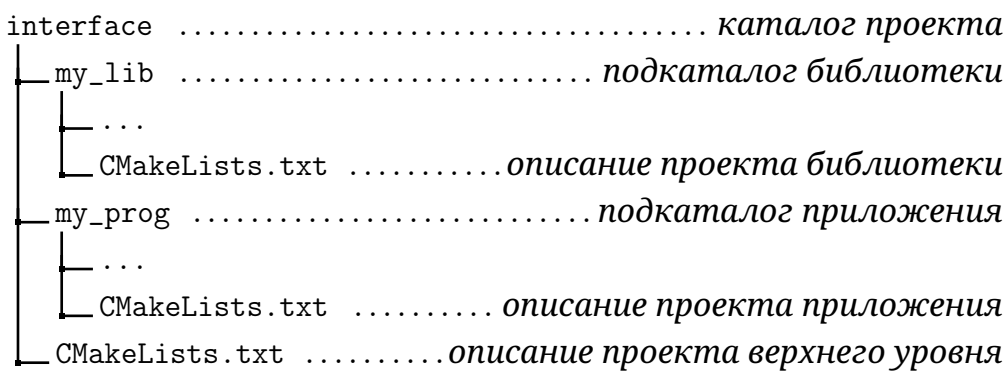

Рис. 2.7. Структура каталога проекта с исполняемым файлом и библиотекой

```
#   "interface"

cmake_minimum_required(VERSION 2.8.11)

project(interface)

add_subdirectory(my_lib)
add_subdirectory(my_prog)
```

Здесь при помощи первой команды устанавливается минимальная версия CMake в 2.8.11, поскольку начиная с неё доступна команда `target_include_directories()`.

Файл `CMakeLists.txt` проекта библиотеки:

```
# CMakeLists.txt для подпроекта "my_lib"

add_library(my_lib my_lib.cpp my_lib.h)

target_include_directories(
```

```
  my_lib
  INTERFACE .)
```

Здесь в качестве каталога поиска заголовочных файлов указывается путь «.», который относительно каталога проекта библиотеки означает сам этот каталог. Этот путь будет использован зависимым от проекта библиотеки проектом приложения.

Файл CMakeLists.txt проекта приложения:

```
# CMakeLists.txt для подпроекта my_prog

add_executable(my_prog my_prog.cpp)

target_link_libraries(my_prog my_lib)
```

Описание этого проекта упрощается по сравнению с предыдущими похожими примерами, так как в нём больше не нужно использовать команду include_directories(). Достаточно лишь установить зависимость между проектами при помощи команды target_link_libraries() (п. 2.6.7). *

Замечание: для удобства в CMake есть специальная переменная CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE, установка которой в истинное значение приводит к автоматическому добавлению каталога подпроекта и каталога его построения в список подключаемых путей для зависимых целей. То есть эта переменная аналогична переменной CMAKE_INCLUDE_CURRENT_DIR (с. 93), которая влияет на построение текущей цели. Таким обра-

зом, вызов команды `target_include_directories()` в примере выше можно заменить на команду:

```
set(CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE ON)
```

▲

2.6.3. add_definitions(), add_compile_options()

```
add_definitions(
  ⟨определение_1⟩ ... ⟨определение_n⟩)

add_compile_options(
  ⟨аргумент_1⟩ ... ⟨аргумент_n⟩)
```

Команда `add_definitions()` добавляет заданные определения символов препроцессора к свойствам текущего каталога и подчинённых (подключаемых при помощи команды `add_subdirectory()`, п. 2.5.3). Определения символов должны быть в формате «`-D`⟨*имя_символа*⟩» или «`/D`⟨*имя_символа*⟩». Система CMake автоматически преобразует такие определения к аргументам командной строки, поддерживаемым используемым компилятором.

Пример

```
add_definitions(-DDEBUG -DEXTRA_TESTS)
```

При использовании в построении компилятора gcc ему будут переданы аргументы «`-DDEBUG -DEXTRA_TESTS`», а при

использовании Microsoft Visual C++ — «`/DDEBUG /DEXTRA_TESTS`». После этого в компилируемых файлах препроцессором будут положительно обрабатываться директивы вида:

```
#ifdef EXTRA_TESTS
  do_extra_tests();
#endif
```

*

Определения в любом другом формате будут переданы без изменения компилятору в качестве аргументов командной строки. Однако в этом случае предпочтительнее использовать команду `add_compile_options()` (появилась в CMake 2.8.12), которая всегда приводит к передаче компилятору заданных аргументов в неизменном виде. Так как допустимые аргументы командной строки зависят от используемого компилятора, при их передаче необходимо использовать проверки на тип компилятора при помощи команды `if()` (п. 2.8.1) и специальных переменных, описывающих систему.

Пример

```
if(MSVC AND MSVC_VERSION GREATER 1400)
  add_compile_options(/MP)
endif()
```

Здесь в случае использования компилятора Microsoft Visual C++ версии 2008 или выше ему при компиляции исходных файлов передаётся аргумент «`/MP`», который приводит к распарал-

леливанию компиляции на доступное системе число процессорных ядер. *

Замечание: при использовании инструмента make добиться параллельного выполнения нескольких целей можно при помощи аргумента командной строки «`-j` ⟨*количество_процессов*⟩». ▲

Пример

```
make -j 8
```

*

Замечание: при передаче компилятору аргументов командной строки, отвечающих за некоторые часто используемые функции, вместо команды `add_compile_options()` предпочтительнее использовать другие команды, которые способны передавать компилятору аргументы в поддерживаемом им формате:

— Для определения символов препроцессора с пустыми значениями предназначена команда `add_definitions()`, описанная в данном параграфе.

— Для передачи компилятору дополнительных путей поиска заголовочных файлов предназначена команда `include_directories()` (п. 2.6.1), которая оказывает влияние на свойства текущего каталога построения, а также команда `target_include_directories()` (п. 2.6.2), которая определяет свойства отдельной цели.

— Для передачи компоновщику дополнительных библиотек используется команда `target_link_libraries()` (п. 2.6.7). При помощи этой же команды можно передать компоновщику любые аргументы.

— Для передачи компилятору аргументов, делающих доступными возможности новых стандартов языков C и C++, используется команда `target_compile_features()` (п. 2.6.6).

— Для передачи компоновщику дополнительных путей поиска библиотек используется команда `link_directories()`. Впрочем, в реальных ситуациях эта команда почти никогда не нужна, так как при добавлении библиотек от других целей или найденных при помощи команд `find_library()` (п. 2.9.2) и т. д. компоновщику будет передан полный путь к библиотеке. ▲

2.6.4. `target_compile_definitions()`

`target_compile_definitions`(
  ⟨*имя_цели*⟩
  `INTERFACE` | `PUBLIC` | `PRIVATE`
    [⟨*определение*$_{1,1}$⟩ ... ⟨*определение*$_{1,m}$⟩]
  [
    `INTERFACE` | `PUBLIC` | `PRIVATE`
      [⟨*определение*$_{2,1}$⟩ ... ⟨*определение*$_{2,n}$⟩]
    ...
  ])

Команда `target_compile_definitions()` доступна в системе CMake начиная с версии 2.8.11. Эта команда аналогична команде `add_definitions()` (п. 2.6.3), однако позволяет добавлять определения символов препроцессора через командную строку вызовов компилятора не только для указанной цели, но и автоматически для всех целей, зависимых от данной, аналогично команде `target_include_directories()` (п. 2.6.2).

2.6.5. `target_compile_options()`

---

target_compile_options(
  ⟨*имя_цели*⟩ [BEFORE]
  INTERFACE | PUBLIC | PRIVATE
    [⟨*аргумент*$_{1,1}$⟩ ... ⟨*аргумент*$_{1,m}$⟩]
  [
    INTERFACE | PUBLIC | PRIVATE
      [⟨*аргумент*$_{2,1}$⟩ ... ⟨*аргумент*$_{2,n}$⟩]
    ...
  ])

Команда `target_compile_options()` доступна в CMake начиная с версии 2.8.12. Эта команда аналогична команде `add_compile_options()` (п. 2.6.3), однако позволяет передавать аргументы компилятору не только для указанной цели, но и автоматически для всех целей, зависимых от данной, аналогично команде `target_include_directories()` (п. 2.6.2).

— При помощи необязательного аргумента BEFORE можно добавлять аргументы компилятора в начало их списка вместо конца (по умолчанию).

2.6.6. target_compile_features()

---

target_compile_features(
⟨*имя_цели*⟩
INTERFACE | PUBLIC | PRIVATE
⟨*возможность*$_1$⟩ ... ⟨*возможность*$_n$⟩)

Команда target_compile_features() доступна в CMake начиная с версии 3.1.0. Эта команда предназначена для передачи специфических для используемого компилятора аргументов, включающих заданные возможности новых стандартов языков C и C++. Передаваемые команде возможности должны быть перечислены в специальных переменных CMAKE_C_COMPILE_FEATURES и CMAKE_CXX_COMPILE_FEATURES. Эти переменные содержат список возможностей, поддерживаемых выбранным компилятором, и являются подмножествами глобальных свойств CMAKE_C_KNOWN_FEATURES и CMAKE_CXX_KNOWN_FEATURES. См. пример на с. 214, который демонстрирует вывод на печать значений этих свойств.

— Аргументы INTERFACE, PUBLIC и PRIVATE позволяют передавать ключи компилятора не только для указанной цели, но и автоматически для всех целей, зависимых от данной. Их смысл аналогичен таким же аргументам для команды target_include_directories() (п. 2.6.2).

Если выбранный для построения компилятор не поддерживает указанную возможность, обработка описания проекта системой CMake прерывается и генерация файлов для конечной системы построения не выполняется (аналогично команде `message(FATAL_ERROR ...)`, п. 2.4.4).

Замечание: хотя эта команда появилась в CMake версии 3.1.0, имеет смысл использовать её как минимум с CMake версии 3.3.0, так как в ней стала доступной информация о возможностях компиляторов gcc, Microsoft Visual C++ и т. д. Для более новых компиляторов необходимо использовать более позднюю версию CMake. ▲

Пример

Пусть необходимо скомпилировать следующую программу:

```
#include <iostream>

int main()
{
  auto n = 0b0'0100'1011;
  std::cout << n << std::endl;
}
```

Для этого необходимо включить соответствующие возможности компилятора C++14:

```
cmake_minimum_required(VERSION 3.2.0)

project(ex-cpp14)
```

```
add_executable(ex-cpp14 ex-cpp14.cpp)

target_compile_features(
  ex-cpp14
  PRIVATE
    cxx_auto_type
    cxx_binary_literals
    cxx_digit_separators)
```

*

Замечание: на момент написания учебника приведённый выше пример работал с компиляторами Microsoft Visual C++ 2015 и gcc-MinGW 4.9.2. ▲

2.6.7. target_link_libraries()

```
target_link_libraries(
  ⟨имя_цели⟩ [⟨строка_1⟩ ... ⟨строка_n⟩]
  [
    [debug | optimized | general] ⟨строка⟩
    ...
  ])

target_link_libraries(
  ⟨имя_цели⟩
  INTERFACE | PUBLIC | PRIVATE ⟨строка_1⟩ ... ⟨строка_m⟩
```

```
[
  INTERFACE | PUBLIC | PRIVATE ⟨строка_1⟩ ... ⟨строка_n⟩
  ...
])
```

Команда предназначена для определения подключаемых библиотек к заданной цели или произвольных аргументов командной строки, передаваемых компоновщику.

Замечание: имя цели должно быть определено в том же самом каталоге проекта, внутри описания которого вызывается эта команда. ▲

Команда определяет тип аргументов, начиная со второго, по следующим правилам:

1) Если передаваемая строка начинается с символов «`-l`» или «`-framework`», то оставшаяся часть строки интерпретируется как имя библиотеки (без префикса `lib` и расширения) или каркаса OS X соответственно.

2) Иначе, если передаваемая строка начинается с символа «`-`», она интерпретируется как аргумент командной строки для компоновщика и во время построения передаётся ему без изменения.

3) Иначе, если строка совпадает с именем другой цели, ранее определённой при помощи команды `add_library()` (п. 2.5.2), текущая цель будет строиться с использованием библиотеки, которая является результатом построения указанной цели. Кроме этого, устанавливается зависимость

при построении текущей цели от указанной, аналогично команде `add_dependencies()` (п. 2.6.8).

4) Иначе строка должна представлять путь к файлу библиотеки, которая будет использована при построении цели.

Несколько команд `target_link_libraries()`, последовательно исполняемых для одной и той же цели, добавляют новые настройки к процессу её компоновки.

Использование команды также транзитивно переносит на зависимую цель требования, определённые для целей связываемых библиотек при помощи интерфейсных аргументов команд `target_include_directories()` (п. 2.6.2) и других, имеющих префикс «`target_`».

Замечание: по умолчанию (см. ниже) устанавливаемые отношения зависимости между целями при помощи команды `target_link_libraries()` являются транзитивными, т. е. распространяются на зависимые цели. Это может оказаться нужным при построении приложения, использующего функции из библиотеки, которая, в свою очередь, использует другие библиотеки. Даже если библиотеки являются статическими, следовательно, одни библиотеки физически никак не участвуют в компоновке других библиотек, они все вместе будут использоваться при компоновке приложения. ▲

Пример

```
# ...
```

```
add_library(lib_aa STATIC ...)
add_library(lib_ab STATIC ...)
add_library(lib_a  STATIC ...)

target_link_libraries(lib_a lib_aa lib_ab)

add_executable(prog ...)

target_link_libraries(prog lib_a)
```

Здесь при компоновке исполняемого файла `prog` будут использоваться статические библиотеки, получаемые в результате построения целей `lib_aa` и т. д. Граф зависимостей между проектами изображён на рис. 2.8. *

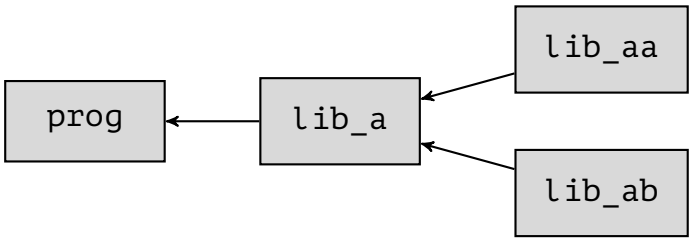

Рис. 2.8. Граф зависимостей между целями исполняемого файла и библиотек

— При помощи необязательных аргументов `debug`, `optimized` или `general` можно ограничить действие следующего аргумента на различные конфигурации:

debug: следующий аргумент будет применяться в конфигурации `Debug` и всех конфигурациях, перечисленных в глобальном свойстве `DEBUG_CONFIGURATIONS`.

optimized: следующий аргумент будет применяться во всех остальных конфигурациях.

general: следующий аргумент будет применяться во всех конфигурациях, как и в случае, когда перед настройкой не указан никакой из этих трёх аргументов.

— Во второй форме команды, появившейся в CMake версии 2.8.12, перед аргументами, определяющими подключаемые библиотеки, можно указывать аргументы `INTERFACE`, `PUBLIC` и `PRIVATE`. Они позволяют переопределять стандартное поведение, когда настройки компоновщика транзитивно передаются зависимым целям. Их смысл аналогичен таким же аргументам для команды `target_include_directories()` (п. 2.6.2).

Пример

Пусть требуется связать приложение с библиотекой Qt Core. Эта библиотека, как и остальные из набора Qt, поставляется в двух вариантах построения: отладочная (с окончанием «d») и предназначенная для конечного пользователя. Правильный вариант в соответствии с конфигурацией построения самого приложения можно выбрать при помощи следующей команды:

```
target_link_libraries(
  myprog debug -lQt5Cored optimized -lQt5Core)
```

*

Замечания:

— Приведённый выше пример предназначен исключительно для демонстрации использования команды `target_link_libraries()` совместно с аргументами `debug` и `optimized`. Более предпочтительный способ подключения библиотек Qt описан в п. 3.3.1.

— Не имеет смысла устанавливать зависимость при компоновке от целей библиотек, собираемых при помощи команд `add_library(`⟨*имя*⟩ `MODULE ...)` (п. 2.5.2), поскольку такой тип библиотек предназначен только для динамического связывания. ▲

Пример

Пусть требуется создать проект динамической (разделяемой) библиотеки. При использовании компилятора Microsoft Visual C++ в таблицу экспорта динамических библиотек попадают только те функции, которые в объявлении имеют атрибут описания `__declspec (dllexport)`. Можно создать заголовочный файл `my_dll.h` вида:

```
int __declspec (dllexport) my_dll_f();
// ...
```

Этот файл нужно будет подключить перед определением функции `my_dll_f()` и всех других, которые необходимо экспортировать. С другой стороны, в коде клиента библиотеки эти же функции требуется объявить без атрибута `__declspec (dllexport)`. Более того, чтобы компилятор смог

сгенерировать более эффективный код, желательно объявление этих функций с атрибутом `__declspec (dllimport)`. Чтобы обойтись одним заголовочным файлом, обычно в таких случаях используются директивы препроцессора. Таким образом, окончательная версия заголовочного файла может быть следующей:

```
#ifndef MY_DLL_H__
#define MY_DLL_H__

#ifdef _MSC_VER
  // Если используется Microsoft Visual C++
  #ifdef MY_DLL_BUILD
    // Если файл подключается из проекта библиотеки
    #define MY_DLL_INTERFACE __declspec (dllexport)
  #else    // MY_DLL_BUILD
    // Если файл подключается из проекта клиента
    #define MY_DLL_INTERFACE __declspec (dllimport)
  #endif    // MY_DLL_BUILD (else)
#else    //  _MSC_VER
  // Если используется любой другой компилятор
  #define MY_DLL_INTERFACE
#endif    //  _MSC_VER (else)

int MY_DLL_INTERFACE my_dll_f();
// ...

#endif    //  MY_DLL_H__
```

Если определён символ препроцессора MY_DLL_BUILD (предполагается, что он определён при построении библиотеки и не определён во всех остальных проектах), символ MY_DLL_INTERFACE, который используется в объявлениях экспортируемых функций, определяется как __declspec (dllexport), в противном случае — как __declspec (dllimport). Теперь описание цели библиотеки может выглядеть следующим образом:

```
add_library(my_dll SHARED my_dll.cpp my_dll.h)

target_compile_definitions(
  my_dll
  PRIVATE -DMY_DLL_BUILD)

set(CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE ON)
```

Для определения символа MY_DLL_BUILD в цели используется команда target_compile_definitions() (п. 2.6.4). Использование в ней аргумента PRIVATE приводит к тому, что символ не будет определён в других целях, подключающих библиотеку. В отличие от команды add_definitions() (п. 2.6.3), команда target_compile_definitions() не делает символ MY_DLL_BUILD определённым для других целей текущего подпроекта.

Описание цели исполняемого файла, использующего динамическую библиотеку (в текущем или отдельном подпроекте), может быть следующим:

```
add_executable(my_prog my_prog.cpp)

target_link_libraries(my_prog my_dll)
```

Команда `target_link_libraries()` транзитивно переносит интерфейсные требования проекта `my_dll` в проект `my_prog`. В данном случае эти требования представляют собой каталог библиотеки, который нужно использовать для поиска заголовочных файлов (включается установкой специальной переменной `CMAKE_INCLUDE_CURRENT_DIR_IN_INTERFACE`, п. 2.6.2). Определение символа `MY_DLL_BUILD` в эти требования не входит, что и требовалось. *

2.6.8. add_dependencies()

---

add_dependencies(
⟨*имя_цели*⟩ [⟨*имя_цели*$_1$⟩ ... ⟨*имя_цели*$_n$⟩])

Команда устанавливает отношение зависимости при построении заданной цели от других. Команда аналогична команде `target_link_libraries()` (п. 2.6.7) за исключением того, что её аргументами могут быть только имена целей и что она не устанавливает зависимостей по подключению библиотек во время компоновки. По сравнению с ней эта команда применяется не так часто. Как правило, она используется при установлении зависимостей для целей, созданных командой `add_custom_target()` (п. 2.10.4).

2.7. Команды обработки данных

2.7.1. set(), unset(), option()

---

set(
⟨*имя_переменной*⟩ [⟨*значение*$_1$⟩ ... ⟨*значение*$_n$⟩])

set(
⟨*имя_переменной*⟩ ⟨*значение*⟩
[
[CACHE ⟨*тип*⟩ ⟨*строка_описания*⟩ [FORCE]] |
PARENT_SCOPE
])

⟨*тип*⟩ ::=
FILEPATH | PATH | STRING | BOOL | INTERNAL

unset(⟨*имя_переменной*⟩ [CACHE | PARENT_SCOPE])

option(
⟨*имя_переменной*⟩ ⟨*строка_описания*⟩ [⟨*значение*⟩])

Первый вариант команды `set()` предназначен для присваивания заданной переменной списка из заданных значений (п. 2.2.2). Таким образом, переменной будет присвоено значение «⟨*значение*$_1$⟩ ; ... ;⟨*значение*$_n$⟩».

Второй вариант команды `set()` при указании аргумента CACHE или PARENT_SCOPE предназначен для установки значения

переменной в другой области действия (п. 2.2.3). В этом случае переменная не может быть переменной окружения.

— При помощи необязательного аргумента CACHE указывается, что команде следует осуществлять присваивание переменной кэша. После этого аргумента указывается тип переменной и строка описания, которая будет использована в качестве информации о переменной в утилите CMake с оконным пользовательским интерфейсом. Тип переменной используется исключительно для удобства и влияет на способ редактирования значения переменной, предоставляемый графической утилитой CMake. Возможны следующие значения типов:

FILEPATH: в переменной хранится путь к файлу. Для редактирования возможно использовать диалоговое окно выбора файла.

PATH: в переменной хранится путь к каталогу. Для редактирования возможно использовать диалоговое окно выбора каталога.

STRING: в переменной хранится строка общего назначения. При редактировании используется обычное поле ввода строки.

BOOL: в переменной хранится логическое значение (п. 2.2.2). Для редактирования будет использовано поле «выключателя».

INTERNAL: утилита CMake не будет отображать информации об этой переменной. Предполагается, что пере-

менная используется для внутренних целей и должна храниться в кэше.

После строки описания может быть указан необязательный аргумент FORCE. Его использование приводит к тому, что переменной будет присвоено значение в любом случае, даже если она уже находится в кэше. По умолчанию, если переменная уже там есть, присваивания не происходит. Такое поведение команды set() подходит в большинстве случаев, так как позволяет сохранять ранее отредактированные значения кэша. Также при присваивании будет удалена из текущей области действия переменная с тем же именем (см. пример ниже).

— При помощи необязательного аргумента PARENT_SCOPE указывается, что следует осуществлять присваивание переменной родительской области действия.

Команда unset() позволяет удалить информацию о заданной переменной (CMake или окружения), т. е. вернуться к состоянию до первого её присваивания. Как и для предыдущей команды, необязательные аргументы CACHE и PARENT_SCOPE позволяют указать, к какой области действия следует применить команду.

Команда option() аналогична команде set() с аргументами CACHE BOOL. По умолчанию переменной устанавливается значение OFF.

Замечание: так как кэш представляет собой отдельную область действия переменных, в нём могут храниться переменные с те-

ми же именами, что и у переменных других областей. При этом такие переменные с одним именем будут разными. ▲

Пример

```
set(VALUE "value 1" CACHE STRING "Test value" FORCE)
set(VALUE "value 2")
message(${VALUE})
unset(VALUE)
message(${VALUE})
```

При исполнении этого фрагмента на печать будут выведены следующие строки:

```
value 2
value 1
```

Этот вывод демонстрирует принцип поиска переменных в различных областях действия (п. 2.2.3). Первая команда `message()` обращается к переменной в текущей области действия. После удаления этой переменной вторая команда `message()` уже обращается к переменной кэша.

Однако если в этот фрагмент внести небольшое изменение:

```
set(VALUE "value 1")
set(VALUE "value 2" CACHE STRING "Test value" FORCE)
message(${VALUE})
unset(VALUE)
message(${VALUE})
```

то в этом случае вывод инструмента CMake будет уже следующим:

```
value 2
value 2
```

Такой вывод происходит потому, что, как было отмечено выше, вторая команда `set()` удаляет из текущей области переменную с тем же именем. При этом команда `unset()` попытается удалить информацию об уже не существующей переменной текущей области. *

2.7.2. `math()`

math(EXPR ⟨*имя_переменной*⟩ ⟨*выражение*⟩)

⟨*выражение*⟩ ::=
⟨*целочисленный_литерал*⟩ |
(⟨*выражение*⟩) |
⟨*выражение*⟩ ⟨*операция*⟩ ⟨*выражение*⟩

⟨*целочисленный_литерал*⟩ ::=
⟨*цифра*⟩...⟨*цифра*⟩

⟨*операция*⟩ ::=
+ | - | * | / | % | | | & | ^ | ~ | << | >>

Эта команда позволяет вычислить арифметическое выражение и присвоить строковое представление его результата за-

данной переменной. Поддерживаются целочисленные операнды и бинарные операции, аналогичные языку C.

Замечания:

— Так как унарные операции командой не поддерживаются, выражения вида `"-4 + 1"` являются неправильными с её точки зрения. Для представления отрицательных чисел следует вычитать положительные литералы из нуля.

— При использовании пробелов внутри выражения его необходимо заключать в кавычки. ▲

2.7.3. `list()`

Команда `list()` реализует основные алгоритмы работы со списками. Команда введена для удобства: её использование необязательно, все задачи, которые она выполняет, можно реализовать другими средствами. Однако для сокращения исходного текста описания проектов и для придания ему большей ясности рекомендуется использовать эту команду там, где это возможно.

Так же как и команда `set()` (п. 2.7.1), команда `list()` может создавать новые переменные. Однако она создаёт их всегда в текущей области действия (п. 2.2.3), в отличие от команды `set()`, для которой можно указать аргументы `CACHE` или `PARENT_SCOPE`. Таким образом, новый список создаётся в текущей области действия, даже если исходный список находится в другой области. Чтобы скопировать полученное значение в исходную область, можно использовать команду `set()`.

Пример

```
function(test)
  list(APPEND TEST_LIST d e f)
  set(TEST_LIST ${TEST_LIST} PARENT_SCOPE)
endfunction()

set(TEST_LIST a b c)
test()
message("${TEST_LIST}")
```

Этот фрагмент кода выведет на печать «a;b;c;d;e;f». Сначала в нём определяется команда test() при помощи команд function()/endfunction() (п. 2.8.4). Далее команда set() присваивает значение переменной TEST_LIST. После этого исполняется команда test(), которая добавляет в конец списка три значения. В результате создаётся переменная с тем же именем TEST_LIST, но уже в области действия функции. Чтобы скопировать значение этой переменной в исходную область действия, используется команда set() с аргументом PARENT_SCOPE. *

При обращении к элементам списков можно использовать неотрицательные индексы $0, 1, 2, \ldots$ для обращения к первому, второму и т. д. элементу с начала или $-1, -2, -3, \ldots$ для обращения к первому, второму и т. д. с конца.

Замечание: при передаче командам значений индексов вне диапазона элементов списка выводится сообщение об ошибке. Выполнение команд продолжается, но генерирование файлов ко-

нечной системы построения не происходит (как при исполнении команды `message(SEND_ERROR ...)`, п. 2.4.4). ▲

---

list(LENGTH ⟨*имя_списка*⟩ ⟨*имя_переменной*⟩)

Эта форма команды записывает в переменную длину списка.

---

list(
  GET ⟨*имя_списка*⟩ ⟨*индекс*$_1$⟩ ... ⟨*индекс*$_n$⟩
  ⟨*имя_переменной*⟩)

Эта форма команды записывает в заданную переменную список из значений исходного списка с заданными индексами.

---

list(APPEND ⟨*имя_списка*⟩ [⟨*значение*$_1$⟩ ... ⟨*значение*$_n$⟩])

Эта форма команды дописывает в конец заданного списка заданные значения. Команда эквивалентна команде:

---

set(
  ⟨*имя_списка*⟩ ${⟨*имя_списка*⟩}
  [⟨*значение*$_1$⟩ ... ⟨*значение*$_n$⟩])

---

list(FIND ⟨*имя_списка*⟩ ⟨*значение*⟩ ⟨*имя_переменной*⟩)

Эта форма команды находит в списке заданное значение и возвращает индекс его первого вхождения (–1 при отсутствии значения в списке).

---

list(

INSERT ⟨*имя_списка*⟩ ⟨*индекс*⟩
[⟨*значение*$_1$⟩ ... ⟨*значение*$_n$⟩])

Эта форма команды вставляет в заданную позицию списка заданные значения.

list(REMOVE_ITEM ⟨*имя_списка*⟩ ⟨*значение*$_1$⟩ ... ⟨*значение*$_n$⟩)

Эта форма команды удаляет из списка все вхождения заданных значений.

list(REMOVE_AT ⟨*имя_списка*⟩ ⟨*индекс*$_1$⟩ ... ⟨*индекс*$_n$⟩)

Эта форма команды удаляет из списка все элементы с заданными индексами (в исходном списке).

list(REMOVE_DUPLICATES ⟨*имя_списка*⟩)

Эта форма команды удаляет из списка все одинаковые элементы, кроме их первых вхождений.

list(REVERSE ⟨*имя_списка*⟩)

Эта форма команды меняет в списке порядок его элементов на противоположный.

list(SORT ⟨*имя_списка*⟩)

Эта форма команды сортирует элементы списка по возрастанию в лексикографическом порядке.

2.7.4. string()

Команда string() реализует основные алгоритмы обработки строк: выделение подстроки, поиск, замену и т. д. В отличие от других языков, в CMake нет средств для индексации отдельных элементов строк как массивов, поэтому команда string() является единственным доступным способом анализа и изменения их содержимого. В этом руководстве будут рассмотрены лишь основные варианты применения команды, наиболее часто используемые на практике.

---

string(⟨*унарная_операция*⟩ ⟨*имя_переменной*⟩ ⟨*строка*⟩)

⟨*унарная_операция*⟩ ::=
LENGTH | TOLOWER | TOUPPER | STRIP

— Команда, вызванная с аргументом LENGTH, записывает в заданную переменную длину заданной строки.

— Команда с аргументом TOLOWER/TOUPPER записывает в переменную заданную строку в нижнем/верхнем регистре.

— Команда с аргументом STRIP записывает в переменную подстроку исходной строки без ведущих и завершающих пробелов.

---

string(CONCAT ⟨*имя_переменной*⟩ [⟨*строка*$_1$⟩ ... ⟨*строка*$_n$⟩])

Команда string(CONCAT ...) выполняет конкатенацию нескольких строк, записывая результат в заданную перемен-

ную. Таким образом, эта команда позволяет легко преобразовать список в строку без разделителей («;»).

---

string(
  FIND ⟨*строка*⟩ ⟨*подстрока*⟩ ⟨*имя_переменной*⟩ [REVERSE])

Команда `string(FIND ...)` выполняет поиск в строке первого (или последнего, если используется аргумент `REVERSE`) вхождения подстроки, записывая индекс её начала в заданную переменную. Позиции в строке нумеруются с `0`, при отсутствии подстроки возвращается `-1`.

---

string(
  SUBSTRING ⟨*строка*⟩ ⟨*начало*⟩ ⟨*длина*⟩ ⟨*имя_переменной*⟩)

Команда `string(SUBSTRING ...)` выделяет из строки подстроку, начинающуюся с заданной позиции, заданной длины (или до конца, если в качестве длины указано `-1`), записывая результат в заданную переменную.

---

string(
  [REGEX] REPLACE
  ⟨*выражение_поиска*⟩ ⟨*выражение_замены*⟩ ⟨*имя_переменной*⟩
  ⟨*строка*$_1$⟩ ... ⟨*строка*$_n$⟩)

Команда `string(REPLACE ...)` выполняет замену всех вхождений заданной подстроки, которая передаётся ей через аргумент ⟨*выражение_поиска*⟩, на строку, которая передаётся через аргумент ⟨*выражение_замены*⟩, в строке, являющейся конкатенацией строк, которые передаются команде через последние аргу-

менты. Изменённая строка записывается в заданную переменную.

Команда `string(REGEX REPLACE ...)` работает аналогично, однако рассматривает свой аргумент ⟨*выражение_поиска*⟩ как регулярное выражение (п. 2.2.5), а ⟨*выражение_замены*⟩ — как выражение замены, в котором могут встречаться ссылки на части найденных подстрок («\⟨*номер*⟩»).

---

string(
  REGEX ⟨*операция*⟩
  ⟨*выражение_поиска*⟩ ⟨*имя_переменной*⟩
  ⟨*строка*$_1$⟩ ... ⟨*строка*$_n$⟩)

⟨*операция*⟩ ::= MATCH | MATCHALL

Команда `string(REGEX MATCH ...)` выполняет поиск первого вхождения подстроки, удовлетворяющей заданному регулярному выражению, в строке, являющейся конкатенацией заданных строк. Найденная подстрока записывается в переменную. Как и в предыдущей форме команды, выполняется выделение наиболее длинной подстроки, насколько это возможно. Если подстрока не найдена, возвращается пустая строка.

Команда `string(REGEX MATCHALL ...)` работает аналогично, но находит все возможные подстроки, соответствующие регулярному выражению, и возвращает их в виде списка.

Примеры

1)

```
string(FIND AsAsAs AsA POSITION1)
```

```
string(FIND AsAsAs AsA POSITION2 REVERSE)
string(FIND AsAsAs AsD POSITION3)
string(REPLACE aS 0ab REP1 a Starta Start)

dump_vars(POSITION1 POSITION2 POSITION3 REP1)
```

Здесь пользовательская команда `dump_vars()` выполняет вывод значений переменных вместе с их именами, переданных ей в качестве аргументов. Определение такой команды рассматривается далее в примере на с. 147. Исполнение команд в итоге приведёт к выводу следующей информации:

```
-- ==============
-- Dumping 4 variables:
--   POSITION1 == "0"
--   POSITION2 == "2"
--   POSITION3 == "-1"
--   REP1 == "0abtart0abtart"
-- ==============
```

Переменная `POSITION1` содержит значение `0`, так как подстрока «AsA» находится в строке «AsAsAs», начиная с позиции 0.

Аналогично, `POSITION2` содержит 2, так как последнее вхождение этой подстроки начинается с позиции 2.

`POSITION3` содержит `-1`, так как искомая подстрока в строке отсутствует.

Команда `string(REPLACE ...)` сначала выполняет конкатенацию своих последних аргументов «a», «Starta» и «Start», а затем в полученной строке выполняет замену.

2)
```
string(
  REGEX MATCHALL
  a[a-z]*z VARS
  az aaxcz--amz-- -az- --za azzx)

string(
  REGEX REPLACE
  "(begin)(.*)(end\.)" "\\3\\2\\1" REPL
  "begin WriteLn end. begin ReadLn end.")

dump_vars(
  VARS REPL CMAKE_MATCH_1 CMAKE_MATCH_2
  CMAKE_MATCH_3)
```

Выполнение команд приведёт к выводу следующего результата:

```
-- ==============
-- Dumping 5 variables:
--   VARS == "azaaxcz;amz;az;aazz"
--   REPL == "end. WriteLn end. begin ReadLn begin"
--   CMAKE_MATCH_1 == "begin"
--   CMAKE_MATCH_2 == " WriteLn end. begin ReadLn "
--   CMAKE_MATCH_3 == "end."
-- ==============
```

В этом примере проявляется «жадный» характер регулярных выражений. Первая команда ищет соответствие в конкатенации последних пяти аргументов регулярному выражению «a[a-z]*z». Первой соответствующей ему подстрокой будет «az», однако команда пытается найти соответствие как можно более длинной подстроки, поэтому первой выделяет подстроку «azaaxcz». Выделению более длинной подстроки препятствует то, что следующим за ней символом будет «-», а он не соответствует образцу «[a-z]».

По той же причине вторая команда в качестве соответствия третьему группирующему подвыражению распознает последнюю строку «end.», а не первую.

3)

```
string(
  REGEX MATCH ab.*cd VAR1
  ---ab-cd-cd---)
string(
  REGEX MATCHALL "aX*" VAR2
  "aXaX-aXXX--aXX--(aX))")
string(
  REGEX MATCHALL "(ab|cd)" VAR3
  "abd--acd--ab--cd--ad")
string(
  REGEX MATCHALL "<([^>]+)>([^<]*)</([^>]+)>" VAR4
  "<code>hello</code>")

dump_vars(
```

```
  CMAKE_MATCH_0 CMAKE_MATCH_1 CMAKE_MATCH_2
  CMAKE_MATCH_3 VAR1 VAR2 VAR3 VAR4)
```

Выполнение команд приведёт к выводу следующего результата:

```
-- ==============
-- Dumping 8 variables:
--   CMAKE_MATCH_0 == "<code>hello</code>"
--   CMAKE_MATCH_1 == "code"
--   CMAKE_MATCH_2 == "hello"
--   CMAKE_MATCH_3 == "code"
--   VAR1 == "ab-cd-cd"
--   VAR2 == "aX;aX;aXXX;aXX;aX"
--   VAR3 == "ab;cd;ab;cd"
--   VAR4 == "<code>hello</code>"
-- ==============
```

Здесь регулярное выражение в последней команде устроено таким образом, чтобы была подавлена его «жадность». Например, в подвыражении, определяющем открывающий тег языка HTML, должен отсутствовать символ «>».

*

Замечание: если требуется только установить факт того, что заданная строка соответствует регулярному выражению, для этого проще воспользоваться командой `if()` (п. 2.8.1) с аргументом `MATCHES`. ▲

2.8. Команды управляющих конструкций

2.8.1. `if()`, `elseif()`, `else()`, `endif()`

---

if(⟨*условие*$_1$⟩)
  ⟨*команды*⟩
[
  elseif(⟨*условие*$_2$⟩)
    ⟨*команды*⟩
  ...
]
[
  else()
    ⟨*команды*⟩
]
endif()

⟨*условие*⟩ ::=
  (⟨*условие*⟩) |
  NOT ⟨*условие*⟩ |
  ⟨*условие*⟩ AND ⟨*условие*⟩ |
  ⟨*условие*⟩ OR ⟨*условие*⟩ |
  ⟨*логическая_константа*⟩ |
  ⟨*имя_переменной*⟩ |
  ⟨*операция*⟩ ⟨*путь*⟩ |
  ⟨*путь*⟩ IS_NEWER_THAN ⟨*путь*⟩ |
  ⟨*значение*⟩ ⟨*бинарная_операция*⟩ ⟨*значение*⟩ |

⟨*значение*⟩ MATCHES ⟨*регулярное_выражение*⟩ |
COMMAND ⟨*имя*⟩ |
TARGET ⟨*имя*⟩ |
DEFINED ⟨*имя_переменной*⟩

⟨*значение*⟩ ::=
⟨*имя_переменной*⟩ | ⟨*строка*⟩

⟨*операция*⟩ ::=
EXISTS | IS_DIRECTORY | IS_SYMLINK | IS_ABSOLUTE

⟨*бинарная_операция*⟩ ::=
LESS | GREATER | EQUAL |
STRLESS | STRGREATER | STREQUAL |
VERSION_LESS | VERSION_GREATER | VERSION_EQUAL

— Команда `if()` вычисляет логическое выражение, которое составляют её аргументы. Если его значение истинно, исполняются команды от текущей команды `if()` до первой соответствующей ей `else()`, `elseif()` или `endif()`.

— Иначе следующая команда `elseif()` проверяет условие. Если оно истинно, исполняются команды до следующей `elseif()` и т. п.

— ...

— Если все условия в `if()` и `elseif()` оказываются ложными, исполняются команды между `else()` и `endif()`.

Наличие команд `elseif()` и `else()` необязательно. Одной команде `if()` может соответствовать несколько команд `elseif()`.

Вычисление выражений происходит следующим образом:

— Скобки и операции `AND`, `OR`, `NOT` имеют ту же семантику, что и в других языках программирования.

— Строка, соответствующая одному из значений логических констант табл. 2.1, интерпретируется в соответствии с этой таблицей. Названия логических констант нечувствительны к регистру.

— Иначе строка интерпретируется как имя переменной. Выражение считается истинным, если в этой переменной не содержится значения «ложь» в соответствии с табл. 2.1.

— Также выражением может состоять из нескольких значений со строковыми обозначениями операций (табл. 2.6). Имена переменных в выражениях (⟨*имя_переменной*⟩) эквивалентны их подстановкам: `${`⟨*имя_переменной*⟩`}`. Гарантируется, что операции с путями будут давать правильные результаты для абсолютных путей.

Таблица 2.6

Операции в логических выражениях

Операция	Значение
Унарные	

Продолжение табл. 2.6

Операция	Значение
`EXISTS`	Проверка того, является ли указанная строка путём к существующему файлу или каталогу в файловой системе
`IS_DIRECTORY`	Проверка того, является ли указанная строка путём к существующему каталогу в файловой системе
`IS_SYMLINK`	Проверка того, является ли указанная строка путём к существующей символической ссылке в файловой системе
`IS_ABSOLUTE`	Проверка того, является ли указанная строка полным путём в файловой системе
`COMMAND`	Проверка того, является ли указанная строка именем команды CMake, макроса или функции (определённой при помощи команды `function()`, п. 2.8.4)
`TARGET`	Проверка того, является ли указанная строка именем цели (определённой при помощи команд `add_executable()`, п. 2.5.1 и т. д.)
`DEFINED`	Проверка того, является ли указанная строка именем переменной с ранее установленным значением

Продолжение табл. 2.6

Операция	Значение
Бинарные	
`LESS`, `EQUAL`, `GREATER`	Аргументы операций интерпретируются как десятичная запись с плавающей точкой вещественных чисел. Выполняется проверка того, является ли первое число соответственно меньше, равным или больше второго
`STRLESS`, `STREQUAL`, `STRGREATER`	Проверка того, является ли первая строка соответственно лексикографически меньше, равной или больше второй
`VERSION_LESS`, `VERSION_EQUAL`, `VERSION_GREATER`	Аргументы операций интерпретируются как номера версий, содержащие до трёх точек. Выполняется проверка того, является ли первый номер соответственно меньше, равным или больше второго
`MATCHES`	Проверка того, соответствует ли строка заданному регулярному выражению (п. 2.2.5)

Окончание табл. 2.6

Операция	Значение
`IS_NEWER_THAN`	Проверка того, является ли время изменения первого файла или каталога больше или равным времени изменения второго. Выражение истинно также в том случае, когда какой-либо из двух файлов или каталогов не существует в файловой системе. Таким образом, эта операция подходит для проверки зависимости между файлами во время построения проекта

Для операций установлены следующие приоритеты в порядке убывания:

1) унарные; 2) бинарные; 3) `NOT`; 4) `AND`; 5) `OR`.

Чтобы избежать неоднозначности, строки в выражениях можно указывать в кавычках. В этом случае они не будут интерпретироваться как имена переменных или операций.

Основной областью применения команды `if()` является организация передачи различных настроек компилятору, компоновщику и т. д. в зависимости от целевой операционной системы и используемого компилятора. Для этого существует ряд специальных переменных CMake, доступных только на чтение, значения которых зависят от используемой среды (табл. 2.7, 2.8, 2.9)[2].

[2]Часть этих переменных описана на публичной странице Wiki проекта CMake, но отсутствует в официальной документации.

Таблица 2.7

Логические переменные для тестирования целевой операционной системы

APPLE *	UNIX **	WIN32
WINCE	WINDOWS_PHONE	WINDOWS_STORE

* OS X.

** Переменная содержит истину для всех UNIX-подобных систем, таких как системы, основанные на ядре Linux, OS X, Windows с набором инструментов cygwin и т. д.

Таблица 2.8

Логические переменные для тестирования используемого компилятора и окружения

Константа	Значение
BORLAND	Любой компилятор фирмы Borland/Embarcadero
CMAKE_CL_64	64-битный компилятор Microsoft
CMAKE_COMPILER_IS_GNUCC	Любая реализация компилятора gcc C
CMAKE_COMPILER_IS_GNUCXX	Любая реализация компилятора gcc C++
CYGWIN	Реализация CMake для cygwin

Продолжение табл. 2.8

Константа	Значение
MINGW	Реализация компилятора MinGW для Windows
MSVC	Microsoft Visual C++ любой версии
MSVC_IDE	Microsoft Visual C++ с проектами для среды Microsoft Visual Studio в отличие от использования компилятора из командной строки
MSVC60	Microsoft Visual C++ 6.0
MSVC70	Microsoft Visual C++.NET 2002 (7.0)
MSVC71	Microsoft Visual C++.NET 2003 (7.1)
MSVC80	Microsoft Visual C++ 2005 (8.0)
MSVC90	Microsoft Visual C++ 2008 (9.0)
MSVC10	Microsoft Visual C++ 2010 (10.0)
MSVC11	Microsoft Visual C++ 2012 (11.0)
MSVC12	Microsoft Visual C++ 2013 (12.0)
MSVC14	Microsoft Visual C++ 2015 (14.0)
MSYS	Реализация компилятора MinGW с make-файлами для оболочки MSYS

Окончание табл. 2.8

Константа	Значение
WATCOM	Компилятор Open Watcom для Windows

Таблица 2.9

Переменные, хранящие версии компиляторов

MSVC_VERSION	XCODE_VERSION

Значения переменной MSVC_VERSION соответствуют значениям макроса _MSC_VER компилятора Microsoft Visual C++. Эти значения соответствуют первым двум компонентам версии компилятора и не совпадают с номером версии Visual Studio (табл. 2.10).

Таблица 2.10

Значения переменной MSVC_VERSION

Константа	Значение
1200	Microsoft Visual C++ 6.0
1300	Microsoft Visual C++ .NET 2002 (7.0)
1310	Microsoft Visual C++ .NET 2003 (7.1)
1400	Microsoft Visual C++ 2005 (8.0)
1500	Microsoft Visual C++ 2008 (9.0)
1600	Microsoft Visual C++ 2010 (10.0)
1700	Microsoft Visual C++ 2012 (11.0)

Окончание табл. 2.10

Константа	Значение
`1800`	Microsoft Visual C++ 2013 (12.0)
`1900`	Microsoft Visual C++ 2015 (14.0)

Например, в Visual Studio 2015 версия компилятора имеет два старших компонента, равных 19.0.

Переменная `XCODE_VERSION` содержит номер версии среды XCode, например `"3.1.2"`.

Пример на с. 99 демонстрирует применение команды `if()` для проверки использования конечной системой построения компилятора Microsoft Visual C++ заданной версии. Это используется для передачи компилятору требуемых аргументов. Пример на с. 142 демонстрирует применение команды `if()` для проверки сложного условия: значение переменной кэша и существование файла в системе.

Пример

Следующий код, будучи помещённым в файл `CMakeLists.txt` корневого проекта перед командой `project()` (п. 2.4.2), выполняет аварийный останов инструмента CMake в случае, если пользователь пытается запустить построение проекта в том же каталоге, где находятся его исходные файлы:

```
if(CMAKE_SOURCE_DIR STREQUAL CMAKE_BINARY_DIR)
  message(
    FATAL_ERROR
```

```
    "Use build directory different from source "
    "directory!")
endif()
```

Для определения путей к каталогам проекта и построения используются специальные переменные `CMAKE_SOURCE_DIR` и `CMAKE_BINARY_DIR`. При сравнении их значений используется тот факт, что система CMake хранит все пути в абсолютном представлении (п. 2.1.3). Команда `message()` (п. 2.4.4) с аргументом `FATAL_ERROR` выполняет прерывание обработки проекта. ⋆

Замечание: продемонстрированная в примере проверка может быть особенно полезной, если в результате построения в выходном каталоге генерируются файлы с именами, совпадающими с именами исходных файлов. Такая проверка выполняется, например, в проекте LLVM. ▲

2.8.2. `while()`, `endwhile()`, `break()`, `continue()`

```
while(⟨условие⟩)
  ⟨команды⟩
endwhile()

break()

continue()
```

Команда `while()` вычисляет условие по таким же правилам, что и команда `if()` (п. 2.8.1). Пока это условие истин-

но, выполняются все команды до соответствующей команды `endwhile()`.

Команда `break()` досрочно прерывает выполнение ближайшего охватывающего цикла `while()` или `foreach()` (п. 2.8.3).

Команда `continue()`, которая появилась в CMake 3.2, досрочно переходит к следующей итерации ближайшего цикла.

2.8.3. `foreach()`, `endforeach()`

---

foreach(⟨*имя_переменной*⟩ ⟨*значение*$_1$⟩ ... ⟨*значение*$_n$⟩)
  ⟨*команды*⟩
endforeach()

foreach(
  ⟨*имя_переменной*⟩ IN
  [LISTS [⟨*список*$_1$⟩ ... ⟨*список*$_n$⟩]]
  [ITEMS [⟨*значение*$_1$⟩ ... ⟨*значение*$_n$⟩]])

foreach(⟨*имя_переменной*⟩ RANGE ⟨*максимум*⟩)

foreach(⟨*имя_переменной*⟩ RANGE ⟨*старт*⟩ ⟨*стоп*⟩ [⟨*шаг*⟩])

Первая форма команды `foreach()` выполняет все команды до соответствующей команды `endforeach()` в количестве раз, равном количеству передаваемых ей значений. При этом на каждой итерации цикла переменная с заданным именем бу-

дет последовательно принимать значения со второго по последний аргумент команды `foreach()`.

Вторая форма команды `foreach()` аналогична первой, но удобнее неё в определённых случаях. В ней можно указывать имена переменных-списков, значения которых будут обходиться в цикле (после необязательного аргумента `LISTS`), а также сами значения непосредственно (после необязательного аргумента `ITEMS`). Пустые списки рассматриваются как списки без элементов.

Третья форма команды `foreach()` выполняет тело цикла с целочисленными значениями переменной от 0 до заданного максимума. Таким образом, тело цикла выполняется «⟨*максимум*⟩ + 1» раз.

Наконец, последняя форма команды `foreach()` выполняет тело цикла с целочисленными значениями переменной в заданном отрезке с заданным шагом (1 по умолчанию). Тело цикла выполняется, пока значение переменной не превышает значения ⟨*стоп*⟩. Таким образом, тело цикла будет выполнено всего «$(\langle стоп \rangle - \langle старт \rangle + 1) \div \langle шаг \rangle$» раз (здесь используется деление нацело с недостатком).

Пример

Пусть проект верхнего уровня должен состоять из нескольких независимых подпроектов. Требуется организовать схему, предусматривающую возможность исключения каждого подпроекта из общего построения в целях экономии времени разработчиков, которым для работы не нужно собирать все компоненты системы. Компонент выбирается для построения, если де-

рево его исходных файлов загружается из отдельного репозитария кода в каталог корневого проекта. В каждом таком проекте есть его описание в файле CMakeLists.txt. Кроме этого, разработчик может исключить проект из построения при помощи флага настройки из кэша CMake.

Файл CMakeLists.txt корневого проекта может быть реализован следующим образом:

```
cmake_minimum_required(VERSION 2.8)

project(complex_project)

set(
  SUBPROJECTS
  program1 program2 super_program
  )

include(build.cmake)
```

Здесь для удобства разработчиков все детали реализации перенесены во вспомогательный файл build.cmake, который подключается командой include() (п. 2.4.3). Таким образом, корневой CMakeLists.txt, который, скорее всего, придётся редактировать чаще, имеет небольшой размер. Список компонент присваивается переменной SUBPROJECTS при помощи команды set() (п. 2.7.1).

Файл build.cmake может иметь следующее содержимое:

```
foreach(PROJ ${SUBPROJECTS})
```

```
  set(
    MY_BUILD_${PROJ} TRUE
    CACHE BOOL "Build the ${PROJ} subproject")
  if(MY_BUILD_${PROJ} AND
    EXISTS
      "${CMAKE_SOURCE_DIR}/${PROJ}/CMakeLists.txt")
    message(
      STATUS
      "The project ${PROJ} will be included")
    add_subdirectory(${PROJ})
  else()
    message(
      STATUS
      "The project ${PROJ} will NOT be included")
  endif()
endforeach()
```

Здесь команда `foreach()` организует перебор значений списка из переменной `SUBPROJECTS`, присваивая каждое из них последовательно переменной `PROJ`. В теле цикла при помощи команды `set()` создаётся логическая переменная в кэше (если на текущий момент её там ещё нет), значение которой определяет включение подпроекта в построение. Имя переменной получается конкатенацией строки «`MY_BUILD_`» и имени подпроекта. Далее команда `if()` (п. 2.8.1) проверяет условие, при котором подпроект будет включён в построение. Для этого необходимо, чтобы значение соответствующей переменной

было истинным и чтобы исходные файлы подпроекта были загружены в дерево корневого проекта. Последнее проверяется существованием файла `CMakeLists.txt` в каталоге с именем подпроекта, который должен находиться в каталоге корневого проекта. Путь к нему содержится в специальной переменной `CMAKE_SOURCE_DIR`. Подпроект подключается командой `add_subdirectory()` (п. 2.5.3).

Команда `foreach()` может быть также записана в следующем виде:

```
foreach(PROJ IN LISTS SUBPROJECTS)
```

После первой обработки проекта системой CMake будет создан файл кэша в каталоге построения с переменными `MY_BUILD_program1` и т. д. Дальше включение отдельных компонент в построение можно настроить, например, при помощи утилиты CMake с графическим интерфейсом (рис. 2.9). *

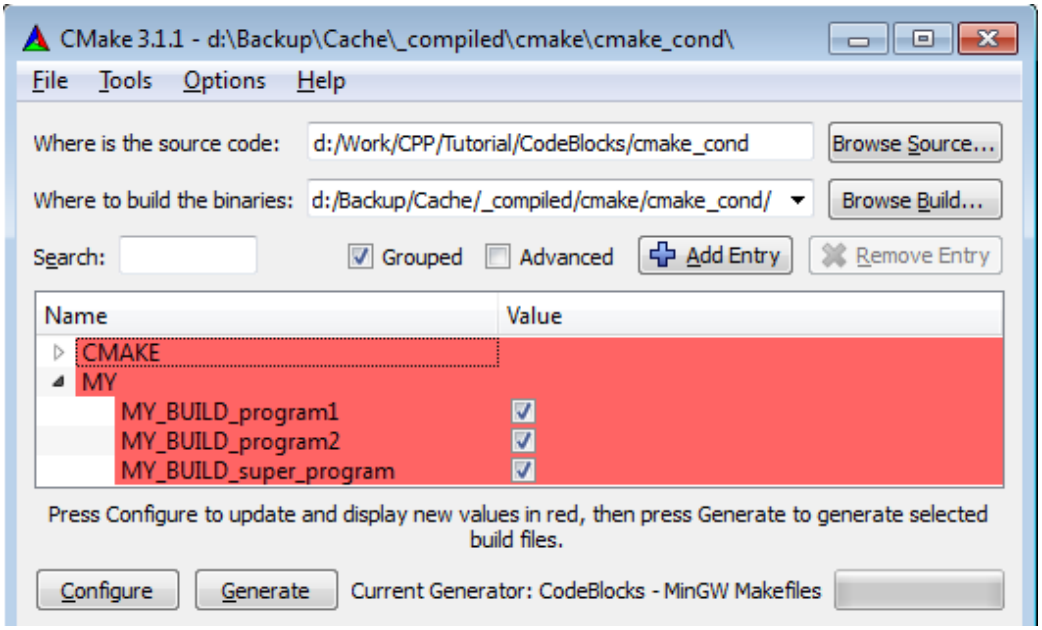

Рис. 2.9. Редактирование переменных кэша в утилите CMake с графическим интерфейсом пользователя

2.8.4. `function()`, `endfunction()`, `return()`

`function``(`⟨*имя_функции*⟩ [⟨*параметр*$_1$⟩ ... ⟨*параметр*$_n$⟩])
⟨*команды*⟩
`endfunction``()`

`return``()`

Команда `function()` запоминает все команды, следующие после неё до соответствующей команды `endfunction()`, без их исполнения. Кроме этого, создаётся новая команда с именем ⟨*имя_функции*⟩. При её вызове сохранённые ранее команды исполняются. При этом все ссылки на формальные параметры вида `${`⟨*параметр*$_1$⟩`}` заменяются значениями фактических аргументов, переданных команде.

Любую команду, определённую при помощи `function()`, можно вызывать с переменным количеством аргументов. Для доступа к ним можно использовать специальные переменные, приведённые в табл. 2.11.

Таблица 2.11

Специальные переменные CMake для доступа к аргументам функций

Перменная	Значение
`ARGC`	Общее количество аргументов
`ARGV0`, `ARGV1`, ...	Первый, второй и т. д. фактические аргументы
`ARGV`	Список всех аргументов
`ARGN`	Список дополнительных аргументов, переданных в дополнение к объявленным

Пример

В следующем фрагменте кода:

```
function(dump_vars)
  message(STATUS ==============)
  message(STATUS "Dumping ${ARGC} variables:")
  foreach(VAR_NAME ${ARGN})
    message(
      STATUS "  ${VAR_NAME} == \"${${VAR_NAME}}\"")
  endforeach()
  message(STATUS ==============)
```

```
endfunction()

dump_vars(DATA GREETING VALUES)
```

создаётся команда с именем `dump_vars`, которая выполняет отладочную печать значений всех переменных, имена которых передаются ей в качестве аргументов. Список аргументов `${ARGN}` обрабатывается при помощи команды `foreach()` (п. 2.8.3). Команда `message()` (п. 2.4.4) в теле цикла печатает имя очередной переменной, которое хранится в переменной `VAR_NAME` (подстановка `${VAR_NAME}`), и её значение (`${${VAR_NAME}}`).

Вызов команды `dump_vars()` в последней строке приводит к выводу на печать следующей информации:

```
-- ==============
-- Dumping 3 variables:
--   DATA == "message(Hello)"
--   GREETING == "Hello"
--   VALUES == "a;b;c\;d"
-- ==============
```

*

Замечания:

— При помощи языковых конструкций вида «`${`⟨*аргумент*⟩`}`» и «`${${`⟨*аргумент*⟩`}}`» можно организовать передачу в функции переменных по ссылке.

— Функциональностью, аналогичной команде `dump_vars()` из приведённого выше примера, обладает команда `cmake_print_variables()` из стандартного модуля CMake `CMakePrintHelpers`.

— В стандартном модуле `CMakeParseArguments` определена команда `cmake_parse_arguments()`, которая предназначена для облегчения синтаксического разбора списков аргументов функций. С её помощью можно определять требования для аргументов функций, аналогичные стандартным командам CMake. ▲

Пример

```
function(quote VAR_NAME)
  set(${VAR_NAME} "\"${${VAR_NAME}}\"" PARENT_SCOPE)
endfunction()

set(STR "some string")
message(${STR})
quote(STR)
message(${STR})
```

Здесь создаётся пользовательская команда `quote()`, которой передаётся имя переменной. Команда добавляет в начало и конец значения этой переменной символы двойных кавычек. Изменение переменной осуществляется командой `set()` (п. 2.7.1) с аргументом `PARENT_SCOPE`, так как переменная нахо-

дится во внешней по отношению к функции `quote()` области действия (п. 2.2.3).

В результате исполнения последних четырёх строк кода на печать будет выведена следующая информация:

```
some string
"some string"
```

*

Команда `return()`, как и в других языках программирования, позволяет досрочно прервать выполнение пользовательской команды и передать управление следующей за ней команде. Также команда `return()` способна прервать выполнение текущего модуля CMake и обработку подпроекта. Таким образом, этой командой можно досрочно прервать исполнение команд `include()` (п. 2.4.3), `add_subdirectory()` (п. 2.5.3) и `find_package()` (п. 2.11.1). Наконец, командой `return()` можно прервать обработку описания проекта верхнего уровня.

2.9. Команды работы с файлами

2.9.1. `get_filename_component()`

```
get_filename_component(
  ⟨имя_переменной⟩ ⟨путь_к_файлу⟩ ⟨компонент⟩
  [CACHE])

get_filename_component(
```

⟨*имя_переменной*⟩ ⟨*команда*⟩
PROGRAM [PROGRAM_ARGS ⟨*имя_переменной_аргументов*⟩]
[CACHE])

С помощью команды `get_filename_component()` в первой форме можно выделить из заданного пути файловой системы (передаётся команде вторым аргументом) нужную часть (имя, расширение и т. д.) или преобразовать путь к абсолютной форме. Относительные пути интерпретируются по отношению к каталогу текущего подпроекта. При этом файл или каталог, к которому указывается путь, не обязательно должен существовать. Возвращаемые пути содержат символы «/» в качестве разделителей имён каталогов и не содержат последнего символа «/».

Результат выполнения команды записывается в переменную, имя которой передаётся в качестве первого аргумента. Третий аргумент определяет, что именно должно быть получено в качестве результата (табл. 2.12).

Таблица 2.12

Значения третьего аргумента команды `get_filename_component()`, определяющие возвращаемый результат

Аргумент	Возвращаемый результат
DIRECTORY	Каталог, содержащий заданный файл или каталог
NAME	Имя с расширением без каталога
EXT	Расширение (часть имени от первой точки)

Окончание табл. 2.12

Аргумент	Возвращаемый результат
NAME_WE	Имя без каталога и расширения (до первой точки)
ABSOLUTE	Полный путь относительно каталога подпроекта
REALPATH	Как ABSOLUTE, но с разрешением символических ссылок

Замечание: значение REALPATH является единственным, при котором исполнение команды приводит к попытке обращения к файлу по указанному пути. ▲

— При помощи необязательного аргумента CACHE можно поместить результирующую переменную в кэш.

Вторая форма команды ищет исполняемый файл по заданной командной строке с учётом системных путей поиска исполняемых файлов (значение переменной окружения PATH и т. д.). Возвращается полный путь к файлу или пустая строка, если файл не найден.

— После необязательного аргумента PROGRAM_ARGS можно указать имя переменной, в которую будут записаны аргументы команды (начиная с первого пробела).

Пример

Пусть требуется указать в описании проекта список файлов, из которых должна собираться некоторая цель. При этом

для некоторых из этих CPP-файлов в каталоге проекта также существуют соответствующие им H-файлы, которые в списке не указаны. Их нужно найти и также включить в цель.

Эту задачу можно решить при помощи следующего файла `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 2.8)

project(ex-find-h)

set(
  FILES_SRC
  main.cpp source1.cpp subdir/source2.cpp)

set(FILES_H)
foreach(SRC IN LISTS FILES_SRC)
  get_filename_component(
    SRC_EXT "${SRC}" EXT)
  if(SRC_EXT STREQUAL .cpp)
    get_filename_component(
      SRC_FULL "${SRC}" ABSOLUTE)
    get_filename_component(
      SRC_DIR "${SRC_FULL}" DIRECTORY)
    get_filename_component(
      SRC_NAME "${SRC}" NAME_WE)
    set(SRC_H "${SRC_DIR}/${SRC_NAME}.h")
    if(
      EXISTS "${SRC_H}" AND
```

```
      NOT IS_DIRECTORY "${SRC_H}")
      message(STATUS "Found ${SRC_H}")
      list(APPEND FILES_H "${SRC_H}")
    endif()
  endif()
endforeach()

add_executable(ex-find-h ${FILES_SRC} ${FILES_H})
```

Здесь список файлов в переменной `FILES_SRC` обходится командой `foreach()` (п. 2.8.3). Для каждого пути выделяется расширение файла и проверяется, что оно равно строке «`.cpp`». Дальше путь к файлу преобразуется к абсолютной форме, затем из него выделяются каталог и имя без расширения. Из них строится имя H-файла и проверяется, что такой файл существует и является файлом, а не каталогом, при помощи команды `if()` (п. 2.8.1). Если это так, путь добавляется к списку в переменной `FILES_H` при помощи команды `list(APPEND ...)` (п. 2.7.3). *

Замечание: в make, CMake и других системах построения существуют средства для поиска файлов с заданными расширениями в каталоге проекта и автоматического их добавления к целям. Такой подход является плохой практикой, так как добавление новых исходных файлов не приводит к изменению описаний проектов. Из-за этого механизм автоматического перестроения в таких случаях не срабатывает и возникают непредвиденные ошибки времени компоновки. Кроме того, сильно замедляется построение крупных проектов. При этом поиск и автомати-

ческое включение заголовочных файлов, продемонстрированные в примере выше, вполне допустимы, так как заголовочные файлы не передаются системой построения на обработку компилятору, а только используются интегрированными средами разработки для отображения в списке исходных файлов. ▲

2.9.2. `find_file()`, `find_library()`, `find_path()`, `find_program()`

Эти команды предназначены для поиска файла общего назначения (`find_file()` — как правило, заголовочных файлов), каталога, содержащего заданный файл (`find_path()`), библиотеки (`find_library()`) или исполняемого файла (`find_program()`) по заданным именам в указываемых каталогах, а также в некоторых стандартных каталогах.

Замечание: если программный пакет поддерживает систему CMake, для его использования вместо описанных здесь команд предпочтительнее применять более высокоуровневую команду `find_package()` (п. 2.11.1). ▲

Все команды имеют почти одинаковый синтаксис аргументов и позволяют указать множество дополнительных аргументов для исключения тех или иных стандартных каталогов из поиска. Ниже приведён сокращённый вариант синтаксиса с основными настройками.

⟨*имя_команды*⟩(

⟨*имя_переменной*⟩

⟨*альтернативные_имена*⟩

```
  [HINTS ⟨путь1,1⟩ ... ⟨путь1,m⟩ [ENV ⟨имя_окружения1⟩]]
  [PATHS ⟨путь2,1⟩ ... ⟨путь2,p⟩ [ENV ⟨имя_окружения2⟩]]
  [PATH_SUFFIXES ⟨суффикс1⟩ ... ⟨суффиксk⟩]
  [DOC ⟨строка_документации⟩])

⟨имя_команды⟩ ::=
  find_file | find_library | find_path | find_program

⟨альтернативные_имена⟩ ::=
  ⟨имя⟩ | NAMES ⟨имя1⟩ ... ⟨имяn⟩
```

Команды сначала проверяют наличие переменной с заданным именем в кэше. Только если переменной там нет или её значение равно «`...-NOTFOUND`» (см. также табл. 2.1), команды выполняют поиск и записывают найденный путь первого встреченного файла, удовлетворяющего критериям, в переменную кэша. Если файл не был найден, в переменную запишется значение «⟨*имя_переменной*⟩`-NOTFOUND`». Таким образом, при последующих запусках инструмента CMake без очистки кэша ранее удавшийся поиск пропускается и генерирование проектов для конечной системы построения выполняется быстрее.

— После аргумента `NAMES` указывается одно или несколько альтернативных имён искомого файла. Команда `find_program()` может добавить к ним расширение исполняемого файла, специфичное для системы, на которой выполняется инструмент CMake. Аналогично, команда `find_library()` может добавлять префикс в начале име-

ни и расширение в конце, например «`lib`⟨*имя*⟩`.so`». Если указывается одно имя, аргумент `NAMES` можно не передавать.

— После необязательного аргумента `HINTS` можно указать дополнительные пути для поиска файла. Здесь рекомендуется указывать пути, которые были ранее найдены в результате исследования системы, например каталоги, где расположены другие ранее найденные файлы. После необязательного аргумента `ENV` можно указать имя переменной окружения, из которой будут прочитаны дополнительные пути поиска.

— Необязательный аргумент `PATHS` аналогичен аргументу `HINTS`, однако в отличие от него указываемые здесь пути будут проверены в последнюю очередь (о порядке просмотра путей см. ниже). Здесь рекомендуется указывать при необходимости жёстко заданные пути к каталогам, такие как `C:\Program Files\SomeProg` или `/usr/local`. Если другие необязательные аргументы не указаны (включая `NAMES`), аргумент `PATHS` также можно не передавать, указывая в команде пути к каталогам сразу после единственного имени файла.

— После необязательного аргумента `PATH_SUFFIXES` можно указать дополнительные подкаталоги, которые будут использованы для поиска в каждом из каталогов.

— После необязательного аргумента DOC можно указать строку документации, которая будет записана в кэш для создаваемой переменной (см. описание команды set(), п. 2.7.1).

Поиск файла выполняется внутри каталогов в следующем порядке:

1) Каталоги, пути к которым указаны в переменных кэша CMake:
   — Команды find_file() и find_path() ищут в каталогах:
     — ⟨*путь*⟩/include для всех путей из специальной переменной CMAKE_PREFIX_PATH. Также, если установлено значение специальной переменной CMAKE_LIBRARY_ARCHITECTURE (архитектура библиотек), используются каталоги ⟨*путь*⟩/include/⟨*архитектура*⟩.
     — Пути из переменной CMAKE_INCLUDE_PATH.
     — Пути из переменной CMAKE_FRAMEWORK_PATH.
   — Команда find_library() ищет в каталогах:
     — ⟨*путь*⟩/lib для всех путей из специальной переменной CMAKE_PREFIX_PATH. Также, если установлено значение специальной переменной CMAKE_LIBRARY_ARCHITECTURE, используются каталоги ⟨*путь*⟩/lib/⟨*архитектура*⟩. При необходимости для заданной архитектуры также выполняется поиск в каталогах lib64 (чтобы повлиять на это, можно вручную

установить значение глобального свойства `FIND_LIBRARY_USE_LIB64_PATHS`).

- Пути из переменной `CMAKE_LIBRARY_PATH`.
- Пути из переменной `CMAKE_FRAMEWORK_PATH`.

- Команда `find_program()` ищет в каталогах:
  - ⟨*путь*⟩`/bin` и ⟨*путь*⟩`/sbin` для путей из переменной `CMAKE_PREFIX_PATH`.
  - Пути из переменной `CMAKE_PROGRAM_PATH`.
  - Пути из переменной `CMAKE_APPBUNDLE_PATH`.

2) Аналогично пункту 1, но вместо указанных переменных CMake используются одноимённые переменные окружения.

3) Пути, переданные командам после аргумента `HINTS`.

4) Пути, перечисленные в переменной окружения `PATH`. Команды `find_file()` и `find_path()` также используют переменную `INCLUDE`, а команда `find_library()` — `LIB`.

5) Аналогично пункту 1, но вместо переменных кэша CMake с именами `CMAKE_..._PATH` используются специальные переменные CMake с именами в форме `CMAKE_SYSTEM_..._PATH` (например, `CMAKE_SYSTEM_PREFIX_PATH`).

6) Пути, переданные командам после аргумента `PATHS`.

В системах, которые не используют имени диска и т. п. в начале полного пути, возможно перенаправление поиска в ка-

талоги, пути к которым добавляются в виде префиксов в начало всех перечисленных выше путей. Это может быть удобно для выполнения кросс-компиляции, когда файлы сборки для целевой системы располагаются в отдельных каталогах. Поиск выполняется в следующем порядке:

1) Префиксы путей, перечисленные в специальной переменной CMake `CMAKE_FIND_ROOT_PATH`. При этом пути, попадающие в каталог, путь к которому хранится в переменной `CMAKE_STAGING_PREFIX` (путь установки при кросс-компиляции), не изменяются.

2) Префикс из специальной переменной `CMAKE_SYSROOT`.

3) Далее выполняется поиск в исходных каталогах без префиксов.

Пример создания цели вызова архиватора 7-zip при помощи команды `add_custom_target()` на с. 192 также демонстрирует использование команды `find_program()`. Пример подключения библиотеки Crypto++ на с. 350 демонстрирует использование команд `find_library()` и `find_path()`.

2.10. Команды добавления специальных целей

2.10.1. configure_file()

---

configure_file(
⟨*входной_файл*⟩ ⟨*выходной_файл*⟩
[COPYONLY] [ESCAPE_QUOTES] [@ONLY]

[NEWLINE_STYLE ⟨*стиль*⟩])

⟨*стиль*⟩ ::= UNIX | DOS | WIN32 | LF | CRLF

Команда configure_file() предназначена для генерирования текстового файла по заданному входному файлу-образцу. Функциональность команды аналогична средствам системы построения Autotools (п. 1.3.2), при помощи которых можно генерировать заголовочные и make-файлы по их шаблонам. Входной файл копируется в выходной, при этом специальные конструкции в нём заменяются в соответствии со значениями переменных CMake. Кроме этого, в генерируемый проект добавляется правило построения, которое заставляет заново запускать инструмент CMake для повторной генерации после изменения входного файла.

Относительный путь для входного файла интерпретируется по отношению к каталогу текущего подпроекта (переменная CMAKE_CURRENT_SOURCE_DIR), для выходного — к текущему каталогу построения (CMAKE_CURRENT_BINARY_DIR). Для выходного файла можно указать существующий каталог, в этом случае файл будет записан в него с тем же именем, что и входной файл.

Замене будут подвержены следующие конструкции входного файла:

— Конструкция @⟨*имя_переменной*⟩@ заменяется в выходном файле значением переменной с соответствующим именем (пустой строкой, если переменная не определена).

— Аналогично, конструкция `${`⟨*имя_переменной*⟩`}`.

— Конструкция:

```
#cmakedefine ⟨имя_переменной⟩ ⟨строка⟩
```

будет заменена на:

```
#define ⟨имя_переменной⟩ ⟨преобразованная_строка⟩
```

если значение переменной истинно в соответствии с табл. 2.1, и на строку:

```
/* #undef ⟨имя_переменной⟩ */
```

если ложно. Правила преобразования также применяются к определению символа препроцессора (⟨*строка*⟩).

— Аналогично, в зависимости от истинности или ложности значения, конструкция:

```
#cmakedefine01 ⟨имя_переменной⟩
```

будет заменена на:

```
#define ⟨имя_переменной⟩ ⟨1_или_0⟩
```

При помощи необязательных аргументов, передаваемых команде, можно настроить способ преобразования файла:

— Передача аргумента `COPYONLY` приводит к обычному копированию файла без преобразований его содержимого.

— Передача ESCAPE_QUOTES приводит к добавлению символов «\» перед каждым символом двойной кавычки во вставляемые значения переменных. Это может быть необходимо для формирования в выходном файле на языках C/C++ строковых литералов.

— Передача @ONLY приводит к замене только конструкций вида @...@. Это может быть необходимо для генерирования по образцу сценариев CMake, где конструкции ${...} могут использоваться при работе с переменными.

— После аргумента NEWLINE_STYLE можно указать способ разделения строк: при помощи символов "\r\n" (CRLF, DOS, WIN32) или "\n" (LF, UNIX). Этот аргумент несовместим с аргументом COPYONLY.

Пример

Пусть требуется добавить возможность для разрабатываемой программы вывода на печать своей версии — той, которая указана после аргумента VERSION команды project() (п. 2.4.2) в файле описания проекта.

Этот номер версии хранится в специальной переменной CMake с именем PROJECT_VERSION. Передать эту информацию в исходный код программы можно при помощи команды configure_file(). Чтобы генерируемый файл был как можно меньше по объёму, можно реализовать конфигурирование заголовочного файла, который затем будет подключаться в коде программы. Таким образом, структура каталога проекта может быть такой, как представлено на рис. 2.10.

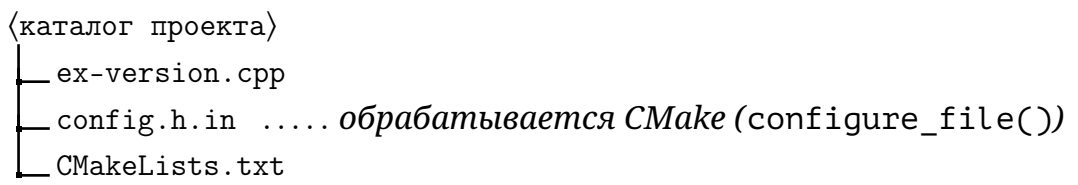

Рис. 2.10. Структура каталога проекта с шаблоном конфигурируемого файла

Содержимое файла `config.h.in`:

```
#cmakedefine PROJECT_VERSION "@PROJECT_VERSION@"
```

Если значение переменной `PROJECT_VERSION` ложно или не определено, в результате конфигурирования будет создан файл со следующим содержанием:

```
/* #undef PROJECT_VERSION */
```

Если же в переменной содержится правильный номер версии, например «`1.0`», будет сгенерировано следующее содержимое выходного файла:

```
#define PROJECT_VERSION "1.0"
```

Так как оставшаяся часть строки в файле шаблона после имени переменной тоже содержит конструкцию конфигурирования (`"@PROJECT_VERSION@"`), при формировании выходного файла она также будет обработана, заменившись значением переменной.

Содержимое файла `ex-version.cpp`:

```
#include "config.h"
```

```
#include <iostream>

const char g_acszVersion[] =
#ifdef PROJECT_VERSION
  PROJECT_VERSION;
#else
  "<unknown version>";
#endif

int main()
{
  std::cout <<
    "Version: " << g_acszVersion << std::endl;
}
```

Здесь в начале подключается файл config.h, который должен быть сгенерирован в выходном каталоге и содержать (или не содержать) определение символа PROJECT_VERSION. Далее при определении строкового константного массива g_acszVersion директивами препроцессора проверяется, был ли определён этот символ. Если да, то массив инициализируется литералом, который является определением PROJECT_VERSION, иначе — строкой «неизвестная версия».

Содержимое файла CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.0)
```

```
# project(ex-version)
project(ex-version VERSION 1.0)

set(CMAKE_INCLUDE_CURRENT_DIR ON)

configure_file(
  config.h.in config.h)
add_executable(
  ex-version ex-version.cpp config.h.in)
```

Здесь в команде `project()` устанавливается версия проекта. Чтобы эта возможность была доступной, выше устанавливается совместимость с минимальной версией CMake 3.0. Чтобы при компиляции добавляемой цели исполняемого файла был найден генерируемый в выходном каталоге файл `config.h`, специальной переменной `CMAKE_INCLUDE_CURRENT_DIR` устанавливается значение истины.

В последней строке файл `config.h.in` добавляется к списку исходных файлов цели исполняемого файла при помощи команды `add_executable()` (п. 2.5.1). Во время построения этот файл не будет обрабатываться инструментами компилятора, однако будет отображаться в списке исходных файлов при работе с интегрированными средами. *

2.10.2. `add_test()`, `enable_testing()`

```
add_test(
  NAME ⟨имя_теста⟩
```

```
COMMAND ⟨команда⟩ [⟨аргумент1⟩ ... ⟨аргументn⟩]
[CONFIGURATIONS ⟨конфигурация1⟩ ... ⟨конфигурацияn⟩]
[WORKING_DIRECTORY ⟨каталог⟩])
```

enable_testing()

Команда `add_test()` добавляет тест к проекту.

— При помощи аргумента `NAME` определяется имя теста.

— При помощи аргумента `COMMAND` определяется команда вместе с аргументами, которая реализует тест. В случае успешного прохождения теста команда должна передать системе код возврата, равный 0. Аргумент ⟨*команда*⟩ может быть либо путём к исполняемому файлу, либо именем цели, определённой ранее при помощи команды `add_executable()` (п. 2.5.1). В последнем случае путь к создаваемому при помощи цели исполняемому файлу будет использован в качестве команды, реализующей тест.

— После необязательного аргумента `CONFIGURATIONS` можно указать список конфигураций, которыми будет ограничено генерирование теста.

— После необязательного аргумента `WORKING_DIRECTORY` можно задать путь к рабочему каталогу исполняемой команды, который определяет одноимённое свойство теста. По умолчанию в качестве рабочего каталога выбирается подкаталог построения, соответствующий текущему каталогу подпроекта.

По умолчанию выполнение команды `add_test()` не вызывает никаких действий. Добавление тестов при помощи этой команды выполняется только после исполнения команды `enable_testing()`. Эта команда должна вызываться в файле `CMakeLists.txt` для проекта верхнего уровня. В этом случае в генерируемом описании проекта создаётся дополнительная цель, которая называется `test` при использовании системы make, проект `RUN_TESTS` для Microsoft Visual Studio и т. д. При запуске цели в случае неудачного исполнения какого-либо теста (с ненулевым кодом возврата) утилита make или подобная ей сама завершится с ненулевым кодом возврата. Это позволяет организовать автоматическое тестирование проекта при построении.

В определении правил для цели тестирования используется вызов утилиты `ctest` из состава CMake. Вместо исполнения цели тестирования системой построения можно вызвать эту команду непосредственно. В этом случае появляется возможность передать ей дополнительные аргументы командной строки, предоставляющие расширенные возможности (определение тайм-аута, выбор подмножества тестов и т. д.). При использовании инструмента make дополнительные аргументы утилите `ctest` можно также передать при помощи переменной окружения `ARGS` (см. пример ниже).

Тестирование является важной составной частью процесса разработки программного обеспечения. Автоматические тесты служат двум целям:

— пользователю программного пакета они предоставляют возможность проверить корректность построения пакета в его системе;

— разработчику программного проекта, вносящему изменения в исходный код, они позволяют убедиться, что эти изменения не нарушают прежней функциональности.

Автоматическое тестирование реализовано в описаниях проектов для многих библиотек с открытым исходным кодом, выполняющих математические расчёты (GMP[3], LAPACK[4] и т. д.). CMake облегчает создание подобных целей тестирования. При разработке многих сложных программных проектов оправданно внедрение практики, в соответствии с которой любые изменения от разработчиков попадают в центральный репозитарий кода только после успешного прохождения построения и тестирования версии с предлагаемыми изменениями. Для автоматизации этого процесса существуют различные средства, например система обзора кода Gerrit[5]. Таким образом, использование подобных систем совместно с CMake позволяет организовать автоматическое тестирование изменений кода перед помещением его в основную ветвь хранилища.

Пример

Пусть проект библиотеки имеет структуру, изображённую на рис. 2.11.

[3] https://gmplib.org/ (дата обращения: 11.01.2015).
[4] http://www.netlib.org/lapack/ (дата обращения: 11.01.2015).
[5] https://code.google.com/p/gerrit/ (дата обращения: 11.01.2015).

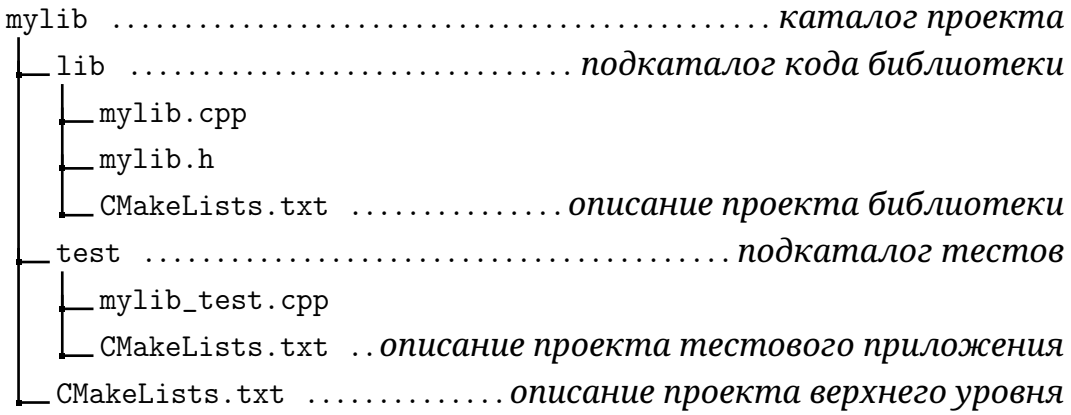

Рис. 2.11. Структура каталога проекта с библиотекой и тестом

Пусть код библиотеки для примера выглядит следующим образом:

Файл `mylib.cpp`:

```
int answer()
{
  return 42;
}
```

Файл `mylib.h`:

```
#ifndef MYLIB_H__
#define MYLIB_H__

int answer();

#endif    // MYLIB_H__
```

Файл `CMakeLists.txt` для подпроекта библиотеки:

```
add_library(mylib mylib.cpp mylib.h)
```

Как можно видеть, код библиотеки вместе с её описанием в системе CMake не имеет каких-либо существенных отличий от других похожих примеров, рассмотренных ранее.

Теперь перейдём к разработке теста. Файл `mylib_test.cpp` может выглядеть следующим образом:

```
#include "mylib.h"

int main()
{
  return (answer() == 42 ? 0 : 1);
}
```

Программа завершается с передачей вызывающей системе нулевого кода возврата, если вызываемая функция библиотеки (`answer()`) возвращает ожидаемое от неё значение, и ненулевого в противном случае.

Файл `CMakeLists.txt` для проекта теста:

```
add_executable(mylib_test mylib_test.cpp)

include_directories(../lib)

target_link_libraries(mylib_test mylib)

add_test(
  NAME mylib_test_1
```

```
  COMMAND mylib_test
  )
```

Здесь определяется цель mylib_test, которая генерирует исполняемый файл, связываемый с библиотекой. Команда add_test() создаёт тест с исполняемым файлом-результатом работы цели mylib_test.

Файл CMakeLists.txt описания проекта верхнего уровня:

```
cmake_minimum_required(VERSION 2.8)

project(mylib)

enable_testing()

add_subdirectory(lib)
add_subdirectory(test)
```

Здесь перед добавлением подпроектов lib и test вызывается команда enable_testing(), которая включает создание цели тестирования.

В POSIX-совместимой системе команды построения и тестирования проекта могут быть следующими:

```
cmake ../../work/cross/mylib
make
make test
```

При необходимости можно передать дополнительные аргументы утилите `ctest`, реализующей тестирование, при помощи переменной `ARGS`:

```
make test ARGS="--timeout 1"
```

Здесь утилите `ctest` передаётся аргумент, задающий таймаут для теста, равный 1 с.

В случае успешного прохождения теста вывод команды «`make test`» будет подобен следующему:

```
Running tests...
Test project /home/dubrov/_build/build_mylib
    Start 1: mylib_test_1
1/1 Test #1: mylib_test_1 .....................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.01 sec
```

Если же функция `answer()` из библиотеки вернёт какое-либо другое значение, вывод команды «`make test`» изменится на следующий:

```
Running tests...
Test project /home/dubrov/_build/build_mylib
    Start 1: mylib_test_1
1/1 Test #1: mylib_test_1 .....................***Failed    0.00 sec

0% tests passed, 1 tests failed out of 1

Total Test time (real) =   0.01 sec
```

```
The following tests FAILED:
          1 - mylib_test_1 (Failed)
Errors while running CTest
make: *** [test] Error 8
```

*

Замечания:

— После запуска тестов в подкаталоге `Testing/Temporary` каталога построения проекта будет создан файл `LastTest.log` с подробностями запуска тестов. В частности, там будет сохранена информация, выводимая тестовыми программами в стандартные потоки.

— Ограничить время работы теста можно также из сценария CMake установкой свойства теста `TIMEOUT` в заданное количество секунд (команда `set_property()`, п. 2.11.2).

— Для разработки сложных тестовых приложений удобно пользоваться сторонними каркасами модульного тестирования, такими как Google Test[6] и Qt Test[7]. ▲

2.10.3. `install()`

Команда `install()` добавляет правило установки сущности к специальной цели генерируемого проекта (цель `install` для системы make, проект `INSTALL` для среды Microsoft Visual Studio и т. д.). При помощи этой команды можно создавать правила для выполнения следующих действий:

[6] https://code.google.com/p/googletest/ (дата обращения: 11.01.2015).
[7] http://doc.qt.io/qt-5/qttest-index.html (дата обращения: 11.01.2015).

— установки результатов выполнения целей проекта (команды `add_executable()`, п. 2.5.1, `add_library()`, п. 2.5.2);

— установки файлов из каталога проекта;

— установки подкаталогов из каталога проекта с возможностью отбора содержащихся в них файлов по маске;

— запуска произвольных сценариев CMake;

— генерирования модулей CMake, облегчающих использование устанавливаемых библиотек в сторонних проектах.

После обработки описания системой CMake в выходном каталоге каждого подпроекта генерируется вспомогательный сценарий `cmake_install.cmake`, интерпретация которого и осуществляет фактическую установку. Вызов инструмента CMake для исполнения этого сценария записан в правилах `install` и т. д.

Команда `install()` способна принимать множество настроек при помощи своих аргументов, мы рассмотрим в данном руководстве только основные из них.

Для установки результатов целей предназначена следующая форма команды `install()`:

```
install(
  TARGETS ⟨имя_цели_1⟩ ... ⟨имя_цели_n⟩
  [EXPORT ⟨имя_экспорта⟩]
  [
    [ARCHIVE | LIBRARY | RINTIME]
    [DESTINATION ⟨каталог⟩]
```

```
    [CONFIGURATIONS [Debug | Release | ...]]
    [COMPONENT ⟨имя_компонента⟩]
  ]
  ...)
```

Команда добавляет к цели правила для установки результатов выполнения заданных целей.

— После необязательного аргумента `EXPORT` можно определить имя для устанавливаемых файлов, которое затем можно использовать для ссылки на них в команде определения правила генерирования сценария CMake (`install(EXPORT ...)`).

— При помощи необязательных аргументов `ARCHIVE`, `LIBRARY` и т. д. можно определить, к каким видам файлов относятся следующие настройки (`DESTINATION` и т. д.). По умолчанию последующие настройки относятся ко всем файлам. Например, исполняемые файлы и динамические библиотеки в системах Windows относятся к категории `RUNTIME`, статические библиотеки, включая библиотеки импорта, относятся к категории `ARCHIVE` (см. описание команды `add_library()`, п. 2.5.2).

— После необязательного аргумента `DESTINATION` можно указать путь к каталогу, в который должны быть установлены файлы. Относительные пути определяют каталоги относительно каталога установки. Каталог установки определяется содержимым специальной переменной

CMake `CMAKE_INSTALL_PREFIX`, в которой по умолчанию хранится путь «`/usr/local`» для POSIX-совместимых систем и путь к подкаталогу с именем проекта внутри каталога «`Program Files`» или «`Program Files (x86)`» для систем Windows.

— После необязательного аргумента `CONFIGURATIONS` можно указать конфигурации, которыми будет ограничено действие правил установки.

— После необязательного аргумента `COMPONENT` можно задать имя компонента, с которым будет связано правило установки. При помощи компонент можно организовать возможность выбора пользователем для установки разных частей программного пакета. Выбор компонента определяется значением переменной CMake `COMPONENT`. Пустое её значение по умолчанию приводит к установке всех компонент.

В одной команде можно указать несколько групп свойств, относящихся к различным видам файлов из перечисленных целей. При помощи нескольких команд `install()` можно установить одни и те же файлы в разные каталоги.

Замечание: все цели, обрабатываемые командной `install()`, должны быть определены в том же самом подпроекте, в описании которого вызывается эта команда. ▲

`install``(`

`FILES` | `PROGRAMS` ⟨*файл*$_1$⟩ `...` ⟨*файл*$_n$⟩

```
  DESTINATION ⟨каталог⟩
  [CONFIGURATIONS [Debug | Release | ...]]
  [COMPONENT ⟨имя_компонента⟩]
  [RENAME ⟨имя⟩])
```

Эта форма команды `install()` добавляет правила для установки заданных файлов. Относительные пути для устанавливаемых файлов интерпретируются по отношению к каталогу текущего (под)проекта.

— Указание аргумента `PROGRAMS` вместо `FILES` приводит к тому, что на POSIX-совместимых системах установленные файлы будут иметь права доступа на исполнение. Эта возможность предназначена прежде всего для установки сценариев из каталога исходных файлов проекта.

— Аргументы `DESTINATION` и т. д. имеют то же значение, что и для команды `install(TARGETS ...)`.

— После необязательного аргумента `RENAME` можно задать новое имя устанавливаемого файла, отличное от исходного. Этот аргумент допустим только в случае указания одного исходного файла.

```
install(
  DIRECTORY [⟨каталог_1⟩ ... ⟨каталог_n⟩]
  DESTINATION ⟨каталог⟩
  [CONFIGURATIONS [Debug | Release | ...]]
  [COMPONENT ⟨имя_компонента⟩]
```

```
[FILES_MATCHING]
[
  [PATTERN ⟨маска⟩ | REGEX ⟨регулярное_выражение⟩]
  [EXCLUDE]
]
[...])
```

Эта форма команды `install()` добавляет правила для установки файлов внутри заданных каталогов. Относительные пути для устанавливаемых каталогов интерпретируются по отношению к каталогу текущего (под)проекта. Если список каталогов пуст, в каталоге установки создаётся пустой каталог, имя которого указано при помощи аргумента `DESTINATION`.

В выходной каталог копируется структура каталогов-источников, при этом последние компоненты путей этих подкаталогов (после последнего символа «/») добавляются к пути каталога-приёмника.

Пример

Команда:

```
install(DIRECTORY dir1 dir2/)
```

добавит следующую структуру каталогов в выходном каталоге (рис. 2.12). *

— Аргументы `DESTINATION` и т. д. имеют то же значение, что и для команды `install(TARGETS ...)`.

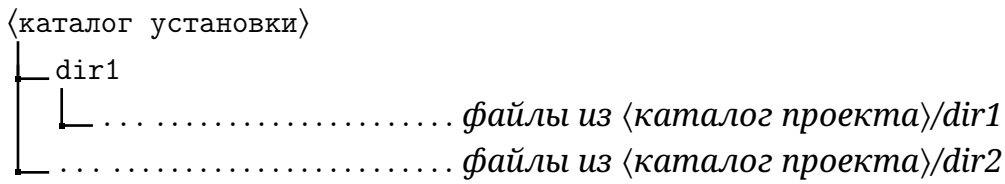

Рис. 2.12. Структура каталога установки

— В конце команды можно указать несколько групп свойств для отбора файлов и каталогов, удовлетворяющих заданной маске или регулярному выражению. При помощи необязательного аргумента `FILES_MATCHING` перед началом первой группы можно указать, что проверяться на соответствие будут только имена файлов, а не каталогов. Далее могут следовать несколько аргументов `PATTERN` и `REGEX`, устанавливающих соответствие на языке масок командных оболочек большинства систем или регулярных выражений (п. 2.2.5) соответственно. Аргумент `EXCLUDE` после образца соответствия означает, что заданные файлы или каталоги будут исключены из установки, а не включены в неё.

```
install(
  [
    [SCRIPT ⟨файл_сценария⟩] [CODE ⟨код_CMake⟩]
  ]
  [COMPONENT ⟨имя_компонента⟩]
  [...])
```

Эта форма команды `install()` добавляет правила для запуска сценариев CMake во время установки.

— После необязательного аргумента `SCRIPT` указывается путь к файлу сценария CMake. Относительные пути интерпретируются по отношению к каталогу проекта.

— После необязательного аргумента `CODE` определяется код CMake, заключённый в двойные кавычки.

— Необязательный аргумент `COMPONENT` имеет то же значение, что и для команды `install(TARGETS ...)`.

Групп аргументов, начинающихся со `SCRIPT` или `CODE`, может быть несколько.

---

install(
  EXPORT ⟨*имя_экспорта*⟩
  DESTINATION ⟨*каталог*⟩
  [NAMESPACE ⟨*имя_пространства_имён*⟩]
  [FILE ⟨*имя_файла*⟩.cmake]
  [CONFIGURATIONS [Debug | Release | ...]]
  [COMPONENT ⟨*имя_компонента*⟩])

Эта форма команды `install()` генерирует в выходном каталоге модуль CMake, который затем можно будет использовать для подключения установленной библиотеки в стороннем проекте. В файле CMake этого проекта можно подключить сценарий при помощи команды `include()` (п. 2.4.3), после чего связывать цели проекта с экспортируемыми целями.

— После аргумента `EXPORT` указывается имя экспортируемого набора файлов, которое должно быть ранее определено

при помощи необязательного аргумента EXPORT команды install(TARGETS ...).

— Аргумент DESTINATION, а также необязательные аргументы CONFIGURATIONS и COMPONENT имеют те же значения, что и для команды install(TARGETS ...).

— После необязательного аргумента NAMESPACE можно указать префикс, который будет добавляться к именам экспортируемых целей. При помощи этих имён в импортирующем проекте можно будет ссылаться на эти цели.

— После необязательного аргумента FILE можно указать имя файла сценария. По умолчанию используется имя «⟨*имя_экспорта*⟩.cmake».

Замечание: в зависимости от выходного каталога для исполнения цели установки могут потребоваться права администратора для данной системы. ▲

Пример

Вернёмся к примеру проекта на с. 87, который состоял из двух библиотек и использовавшего их исполняемого файла. Этот пример демонстрировал, как при помощи присваиваний специальным переменным можно добиться структуры подкаталогов в каталоге построения, совместимой с рекомендациями GNU. Попробуем теперь определить правила для установки проекта с подобной структурой в каталоге установки. Кроме этого, разделим файлы проекта на два компонента: для обычного пользователя и для разработчика. Очевидно, что для пользователя

достаточно установки исполняемого файла my_program и динамических (разделяемых) библиотек my_library_1/_2 (если тип библиотек — SHARED). Для разработчика кроме этих файлов также будут нужны статические библиотеки/библиотеки импорта, заголовочные файлы из подкаталогов my_library_1/_2 и сценарии CMake для подключения библиотек.

Файл CMakeLists.txt подкаталога my_library_1, решающий указанные задачи, может выглядеть следующим образом:

```
add_library(my_library_1 f.cpp f.h)

get_property(
  LIB_TYPE
  TARGET my_library_1
  PROPERTY TYPE)

if(LIB_TYPE STREQUAL SHARED_LIBRARY)
  install(
    TARGETS my_library_1
    COMPONENT user
    RUNTIME
      DESTINATION bin
    LIBRARY
      DESTINATION lib)
endif()

install(
  TARGETS my_library_1
```

```
  EXPORT my_library_1
  COMPONENT developer
  RUNTIME
    DESTINATION bin
  LIBRARY
    DESTINATION lib
  ARCHIVE
    DESTINATION lib)

install(
  DIRECTORY .
  DESTINATION include
  COMPONENT developer
  FILES_MATCHING
    PATTERN "*.h")

install(
  EXPORT my_library_1
  DESTINATION share
  COMPONENT developer)
```

Здесь первая команда `install()` отвечает за установку файлов библиотек, необходимых только для запуска программы. Если тип собираемых библиотек статический, при её исполнении система CMake выведет сообщение об ошибке из-за того, что в команде отсутствует группа настроек `STATIC`. Поэтому сначала проверяется тип библиотеки при помощи свой-

ства TYPE её цели. Свойство считывается при помощи команды get_property() (п. 2.11.2). При помощи условного оператора (п. 2.8.1) команда исполняется, только если тип библиотеки — SHARED. В системах Windows тип выходного файла динамической библиотеки — RUNTIME, и он устанавливается в подкаталог bin вместе с исполняемым файлом. В POSIX-системах разделяемая библиотека помещается в подкаталог lib. Группа настроек STATIC в команде пропущена, из-за чего библиотека импорта установлена не будет.

Файл CMakeLists.txt подкаталога my_library_2 может выглядеть аналогичным образом. Чтобы избежать стиля «copy-paste», а также упростить в будущем добавление новых библиотек, можно оформить код, начинающийся со второй команды, в виде функции в отдельном модуле CMake с использованием команд function() и endfunction() (п. 2.8.4):

```
function(my_install LIB_NAME)
  # Здесь помещается предыдущий код, начиная
  # с get_property(), при этом my_library_1
  # нужно везде заменить на ${LIB_NAME}
endfunction()
```

Файл CMakeLists.txt подкаталога my_program:

```
add_executable(my_program main.cpp)

include_directories(../my_library_1 ../my_library_2)

target_link_libraries(
```

```
  my_program my_library_1 my_library_2)

install(
  TARGETS my_program
  COMPONENT user
  DESTINATION bin)

install(
  TARGETS my_program
  COMPONENT developer
  DESTINATION bin)
```

Команды для построения и установки могут быть подобными следующим:

```
cmake^
  -G "MinGW Makefiles"^
  -D CMAKE_INSTALL_PREFIX=⟨каталог_установки⟩^
  ⟨каталог_проекта⟩

mingw32-make

cmake -D COMPONENT=developer -P cmake_install.cmake
```

Здесь вместо команды «`mingw32-make install`» используется непосредственный вызов сгенерированного сценария установки, который позволяет определить в командной строке значение переменной `COMPONENT` и тем самым выбрать компонент для установки.

После выполнения построения и установки компонента «developer» структура каталога установки, например, в системе Windows при построении динамических библиотек и использовании компилятора MinGW будет выглядеть так, как показано на рис. 2.13.

```
⟨каталог установки⟩
├─ bin
│  ├─ libmy_library_1.dll
│  ├─ libmy_library_2.dll
│  └─ my_program.exe
├─ include
│  ├─ f.h ..... заголовочный файл библиотеки my_library_1
│  └─ g.h ..... заголовочный файл библиотеки my_library_2
├─ lib
│  ├─ libmy_library_1.dll.a ... библиотека импорта my_library_1
│  └─ libmy_library_2.dll.a ... библиотека импорта my_library_2
└─ share
   ├─ my_library_1.cmake
   ├─ my_library_1-noconfig.cmake ..... вспомогательный файл
   ├─ my_library_2.cmake
   └─ my_library_2-noconfig.cmake ..... вспомогательный файл
```

Рис. 2.13. Структура каталога установки с исполняемым файлом и двумя библиотеками

Файл CMakeLists.txt проекта исполняемого файла, использующего библиотеку my_library_1, может выглядеть следующим образом:

```
cmake_minimum_required(VERSION 2.8)

project(my_use)
```

```
set(
  MY_LIBRARY_1_PATH ""
  CACHE PATH "Path to my_library_1 installation")

include(
  "${MY_LIBRARY_1_PATH}/share/my_library_1.cmake")

add_executable(my_use my_use.cpp)

include_directories("${MY_LIBRARY_1_PATH}/include")
target_link_libraries(my_use my_library_1)
```

Команды построения проекта могут быть подобны следующим:

```
cmake^
  -G "MinGW Makefiles"^
  -D MY_LIBRARY_1_PATH=⟨каталог_установки_библиотеки_1⟩^
  ⟨каталог_проекта_использующего_библиотеку_1⟩

mingw32-make
```

Здесь ⟨*каталог_установки_библиотеки_1*⟩ — путь к каталогу, куда ранее был установлен предыдущий проект с библиотекой `my_library_1`. *

Замечания:

— Использование команды `target_include_directories()` (п. 2.6.2) в примере выше затруднено тем, что в коман-

де указываются пути относительно каталога построения, а не установки. Исправить эту проблему можно при помощи выражений генераторов (п. 2.12.2).

— В дополнение к модулям подключения библиотек, генерируемым командой `install(EXPORT ...)`, CMake также поддерживает конфигурационные файлы, используемые командой `find_package()` (п. 2.11.1). Они предоставляют дополнительные возможности, например информацию о совместимости версий библиотек. ▲

2.10.4. add_custom_target()

---

add_custom_target(
⟨*логическое_имя_цели*⟩
[ALL]
[⟨*путь_к_команде*$_1$⟩ [⟨*аргумент*$_{1,1}$⟩ ... ⟨*аргумент*$_{1,m}$⟩]]
[
COMMAND
⟨*путь_к_команде*$_2$⟩ [⟨*аргумент*$_{2,1}$⟩ ... ⟨*аргумент*$_{2,n}$⟩]
...
]
[DEPENDS ⟨*файл*$_1$⟩ ... ⟨*файл*$_k$⟩]
[WORKING_DIRECTORY ⟨*каталог*⟩]
[VERBATIM]
[SOURCES ⟨*исходный_модуль*$_1$⟩ ... ⟨*исходный_модуль*$_p$⟩])

Команда добавляет к проекту цель, построение которой заключается в исполнении заданных инструментов. При этом ко-

нечная система построения считает, что цель должна перестраиваться всегда, вне зависимости от того, какие файлы создаются при исполнении этих инструментов (как будто дата последнего изменения цели всегда старше требуемого). Таким образом, цель, создаваемая при помощи этой команды, ведёт себя аналогично фальшивым целям системы make (п. 1.3.1).

Иногда команда `add_custom_target()` применяется для создания цели, построение которой приводит к последовательному построению нескольких других ранее определённых целей. В этом случае список команд на исполнение инструментов будет пустым, а зависимость данной цели от других устанавливается при помощи команды `add_dependencies()` (п. 2.6.8) или от файлов при помощи аргумента `DEPENDS` (см. пример на с. 328).

— По умолчанию, создаваемая цель не участвует в построении общего проекта (цель «`all`» в make и т. п.). Обычно это имеет смысл, поскольку фальшивые цели запускаются на построение каждый раз при их использовании. Они служат для выполнения некоторых специфических действий, используемых нечасто. Если всё же необходимо, чтобы данная цель участвовала в построении общей цели, можно использовать необязательный аргумент `ALL` команды `add_custom_target()`.

— Несколько последовательных команд оболочки, возможно, вместе с их аргументами командной строки указываются друг за другом, каждая последующая отделяется от предыдущей аргументом `COMMAND`. Перед первой командой аргу-

мент `COMMAND` указывать необязательно. При построении цели команды будут выполняться в порядке их указания в аргументах команды `add_custom_target()`.

— После необязательного аргумента `DEPENDS` можно указать файлы, от которых должна зависеть цель. Здесь также можно указывать выходные файлы для команд `add_custom_command()` (п. 2.10.5), вызванных в текущем подпроекте. Команда `add_custom_command()` похожа на команду `add_custom_target()`. Основное её отличие заключается в том, что в ней явно указываются выходные файлы, создаваемые в результате построения цели.

— После необязательного аргумента `WORKING_DIRECTORY` можно определить путь к рабочему каталогу для вызываемых инструментов. Относительный путь интерпретируется по отношению к подкаталогу построения для текущего (под)проекта.

— Некоторые из символов, встречающиеся в аргументах команд цели, могут иметь специальное значение в командных оболочках различных систем. При использовании необязательного аргумента `VERBATIM` все аргументы командной строки будут при необходимости дополнены зависящими от системы escape-символами так, чтобы они были переданы инструментам в неизменном виде. Например, строка «`$n`» будет интерпретирована оболочкой bash как подстановка значения переменной `n`. Чтобы передать такую строку в качестве аргумента командной

строки, необходимо добавить перед символом «$» escape-символ «\».

— После необязательного аргумента SOURCES можно указать исходные файлы для создаваемой цели. Эта настройка никак не влияет на генерируемые правила и может быть использована исключительно для удобства разработчика: указанные исходные файлы будут отображаться интегрированными средами как входящие в проект. Эти файлы могут иметь какое-либо отношение к построению цели. Например, они могут обрабатываться вызываемыми инструментами.

Пример

Пусть требуется определить цель проекта, исполнение которой приводило бы к созданию в каталоге построения архива 7-zip[8] с исходными файлами (содержимым каталога проекта).

В целях повторной используемости кода реализуем эту задачу в отдельном модуле CMake `7zip.cmake`:

```
set(BINDIR32_ENV_NAME "ProgramFiles(x86)")

find_program(
  7ZIP_EXECUTABLE
  NAMES
    7z 7za
  PATHS
    "$ENV{ProgramFiles}/7-Zip"
```

[8] http://7-zip.org/ (дата обращения: 27.03.2015).

```
    "$ENV{${BINDIR32_ENV_NAME}}/7-Zip"
    "C:/Program Files/7-Zip"
    "C:/Program Files (x86)/7-Zip"
  )

if(7ZIP_EXECUTABLE)
  add_custom_target(
    create_archive
    COMMAND
      "${7ZIP_EXECUTABLE}"
        a "${PROJECT_NAME}.7z" "${PROJECT_SOURCE_DIR}"
    WORKING_DIRECTORY
      "${PROJECT_BINARY_DIR}"
    )
else()
  message(
    WARNING
    "Could not find 7-zip archiver on this system. "
    "You can manually assign a path to it to "
    "7ZIP_EXECUTABLE variable.")
endif()
```

Здесь для получения пути к консольному архиватору 7-zip используется команда `find_program()` (п. 2.9.2), которая записывает результат в переменную кэша `7ZIP_EXECUTABLE`. После аргумента `PATHS` ей передаются пути к каталогам, где эта программа может находиться в системе Windows. Сначала

используются пути относительно каталогов «Program Files» и «Program Files (x86)». Пути к ним хранятся в переменных окружения ProgramFiles и ProgramFiles(x86). Так как последняя переменная содержит символы скобок, которые недопустимы с точки зрения CMake в именах переменных, это ограничение обходится присваиванием строки с именем переменной BINDIR32_ENV_NAME и разыменования последней в месте использования. На всякий случай также проверяются типичные пути к этим каталогам без ссылок на переменные окружения.

При определении цели create_archive используется вызов команды, передающей архиватору аргументы для добавления ко вновь создаваемому архиву с именем проекта (переменная PROJECT_NAME) каталога проекта (переменная PROJECT_SOURCE_DIR) вместе с содержимым. Архив будет создан в каталоге построения (переменная PROJECT_BINARY_DIR), который назначается рабочим для команды.

Пример файла CMakeLists.txt, использующего модуль:

```
cmake_minimum_required(VERSION 2.8)

project(ex-7zip)

add_executable(ex-7zip ex-7zip.cpp)

include(7zip.cmake)
```

Здесь модуль подключается командой include() (п. 2.4.3).*

2.10.5. add_custom_command()

---

add_custom_command(
  OUTPUT ⟨*файл*$_1$⟩ ... ⟨*файл*$_m$⟩
  ⟨*команды*⟩
  [MAIN_DEPENDENCY ⟨*зависимость*⟩]
  [DEPENDS [⟨*зависимость*$_1$⟩ ... ⟨*зависимость*$_n$⟩]]
  [WORKING_DIRECTORY ⟨*каталог*⟩]
  [VERBATIM] [APPEND])

add_custom_command(
  TARGET ⟨*имя_цели*⟩
  PRE_BUILD | PRE_LINK | POST_BUILD
  ⟨*команды*⟩
  [WORKING_DIRECTORY ⟨*каталог*⟩]
  [VERBATIM])

⟨*команды*⟩ ::=
  ⟨*команда*$_1$⟩ ... ⟨*команда*$_k$⟩

⟨*команда*⟩ ::=
  COMMAND ⟨*путь_к_команде*⟩ [⟨*аргумент*$_1$⟩ ... ⟨*аргумент*$_p$⟩]

Первая форма команды `add_custom_command()` добавляет к проекту цель, построение которой реализуется инструментами с заданными командами оболочки. В отличие от команды `add_custom_target()` (п. 2.10.4), создающей фальшивую цель, выходные файлы цели этой команды задаются явно. Также в от-

личие от неё команда `add_custom_command()` не определяет логического имени создаваемой цели. Предполагается, что цель должна запускаться на построение перед исполнением других целей, которые зависят от её выходных файлов.

— После аргумента `OUTPUT` перечисляются выходные файлы, которые должны быть сгенерированы при построении цели. Относительные пути указываются по отношению к выходному каталогу текущего проекта.

— После каждого из нескольких необязательных аргументов `COMMAND` указывается команда оболочки для вызова инструмента вместе с аргументами. Вместо пути можно указать имя цели, созданной при помощи команды `add_executable()` (п. 2.5.1). В этом случае будет запущен исполняемый файл, создаваемый этой целью, и будет установлена зависимость текущей цели от запускаемой.

— После необязательного аргумента `MAIN_DEPENDENCY` можно указать основную зависимость. Основная зависимость полностью аналогична остальным зависимостям, указываемым после аргумента `DEPENDS` (см. далее), но для генератора Visual Studio указывает, в какой проект необходимо добавить пользовательские команды.

— После необязательного аргумента `DEPENDS` можно указать список файлов и целей, от которых должна зависеть текущая цель. Файлы могут быть, например, выходными для других целей, созданных при помощи команды `add_custom_command()`. Если другая цель используется для

построения исполняемого файла или библиотеки, кроме зависимости на уровне целей также добавляется зависимость на уровне исходных файлов: повторное построение другой цели приводит к запуску текущей.

— Необязательный аргумент `WORKING_DIRECTORY` (вместе со следующим за ним путём), а также `VERBATIM` имеют такое же значение, как и для команды `add_custom_target()` (п. 2.10.4).

— Необязательный аргумент `APPEND` указывает на то, что команды вызова инструментов и зависимости должны быть добавлены к другим командам и зависимостям, ранее определённым другой командой `add_custom_command()` с теми же выходными файлами.

Таким образом, создаваемая командой цель будет запускаться, если хотя бы один из её выходных файлов используется другой целью, которую надо построить, и будет выполнено хотя бы одно из следующих условий:

— хотя бы один из файлов, перечисленных после аргумента `OUTPUT`, отсутствует на диске;

— время последнего изменения любого из файлов, перечисленных после аргумента `DEPENDS` (или выходных файлов перечисленных там же целей), если такие есть, позже времени изменения хотя бы одного из выходных файлов текущей цели.

Другая форма команды `add_custom_command()` предназначена для добавления команд вызова внешних инструментов

к уже существующим целям. Аналогичными возможностями обладает, например, интегрированная среда Visual Studio, в настройках проекта которой есть страницы «Событие перед сборкой», «Событие перед компоновкой» и «Событие после сборки». Команды будут исполняться, только если в процессе построения выходные файлы указанной цели должны быть повторно созданы.

— После аргумента `TARGET` указывается логическое имя цели, созданной ранее при помощи команды `add_executable()` (п. 2.5.1) и т. п.

— При помощи следующего аргумента указывается, в какой момент должны быть запущены команды:

`PRE_BUILD`: перед построением цели. Этот аргумент поддерживается только генераторами для Visual Studio версии 7.0 или выше. Для остальных генераторов этот аргумент эквивалентен аргументу `PRE_LINK` (см. далее).

`PRE_LINK`: после компиляции всех файлов, но перед запуском компоновщика/инструмента создания библиотеки. Это событие не поддерживается для целей, созданных командой `add_custom_target()` (п. 2.10.4).

`POST_BUILD`: после исполнения всех инструментов для построения текущей цели.

— Аргументы `COMMAND`, `WORKING_DIRECTORY` и `VERBATIM` имеют такое же значение, как и для первой формы команды `add_custom_command()`.

Пример

Рассмотрим упрощённую модель ситуации, когда часть исходных файлов проекта генерируются внешними инструментами. Как будет видно дальше, такая ситуация является типичной для проектов, использующих библиотеку Qt. В других проектах, например LLVM, исходные файлы создаются в результате запуска исполняемых файлов, являющихся результатами построения других целей.

Файл `CMakeLists.txt` проекта:

```
# ...

add_custom_command(
  OUTPUT out.cpp
  COMMAND echo int f1() { return 14\; } > out.cpp)

add_executable(
  my_prog1
  my_prog1.cpp "${CMAKE_CURRENT_BINARY_DIR}/out.cpp")
```

Здесь командой `add_custom_command()` создаётся цель, генерирующая текстовый файл `out.cpp` в текущем каталоге построения. Для этого цель вызывает команду оболочки `echo`, которая записывает в файл строку:

```
int f1() { return 14; }
```

(Escape-символ «\» перед точкой с запятой необходим в коде CMake, чтобы она не воспринималась как разделитель элементов списка.)

Так как текущий каталог для команды не указан, по умолчанию выбирается текущий каталог построения. Следующей командой определяется цель `my_prog1` для создания исполняемого файла. Одним из исходных файлов в ней указан генерируемый предыдущей целью файл в выходном каталоге. Таким образом, если этот файл на диске отсутствует, при построении автоматически запускается создающая его цель. Так как у неё нет других зависимостей, она будет запускаться только при отсутствии файла `out.cpp`.

Файл `my_prog1.cpp`:

```
#include <iostream>

int f1();

int main()
{
  std::cout << f1() << std::endl;
}
```

*

Замечание: использованный в примере вызов команды `echo` будет корректно работать в системе Windows. Чтобы эта команда записала правильное содержимое в выходной файл в POSIX-совместимой командной оболочке, формат её вызова должен быть следующим:

```
echo "int f1() { return 14; }" > out.cpp
```

Другое решение заключается в передаче аргумента `VERBATIM` команде `add_custom_command()`. ▲

2.11. Прочие команды

2.11.1. `find_package()`

Команда `find_package()` предназначена для поиска пакетов CMake. *Пакетом* (package) называется набор внешних библиотек и инструментов вместе с модулями CMake, предоставляющими информацию о зависимостях для их подключения. Результатом выполнения команды в случае успешного поиска является набор целей и переменных, которые можно использовать в описании проекта (команды `target_link_libraries()`, п. 2.6.7 и т. д.).

Для реализации логики поиска пакета нужно выполнить больше действий, чем просто сгенерировать командой `install(EXPORT ...)` (п. 2.10.3) модуль для подключения библиотеки, поэтому в данном учебнике эта тема не рассматривается. Зато пользоваться командой `find_package()` удобнее, особенно для подключения наборов библиотек со сложной организацией. Во многих известных и широко используемых библиотеках уже встроена поддержка их поиска при помощи команды `find_package()`, примеры использования можно найти в главе 3.

Команда `find_package()` поддерживает большое количество настроек при помощи необязательных аргументов, в настоящем руководстве будут рассмотрены только основные из них,

имеющие наибольшую ценность для конечного пользователя стороннего пакета. Стоит лишь отметить два режима работы команды:

При помощи конфигурационного файла: этот режим больше подходит для поиска тех программных пакетов, которые разрабатывались с использованием системы CMake. В этом случае разработчик стороннего пакета имеет возможность в описании проекта реализовать правила для генерирования конфигурационного файла, включая рассмотренную ранее команду `install(EXPORT ...)`, а также команды из вспомогательного стандартного модуля `CMakePackageConfigHelpers`. В этом случае во время исполнения команды `find_package()` система CMake берёт на себя ответственность за автоматическое выполнение таких задач, как проверка соответствия версий, заполнение стандартных переменных. Вместе с дистрибутивом файлов проекта, предназначенных для разработчиков, распространяется конфигурационный файл ⟨*имя_пакета*⟩`Config.cmake` или ⟨*имя_пакета_в_нижнем_регистре*⟩`-config.cmake`.

При помощи модуля поиска: этот режим может быть использован для тех пакетов, которые разрабатывались без использования системы CMake. В этом случае разработчик, не имеющий отношения к разработке стороннего пакета, может реализовать *модуль поиска* — модуль CMake с именем `Find`⟨*имя_пакета*⟩`.cmake`, выполняющий поиск пакета в системе по некоторым правилам. В этом случае модуль

поиска должен реализовать собственную логику таких задач, как проверка версий на совместимость и т. д. Модуль поиска, как и любой модуль CMake, может располагаться в любом из каталогов, перечисленных в специальной переменной `CMAKE_MODULE_PATH`. Система CMake поставляется с большим количеством (более 140) модулей поиска для основных популярных программных пакетов.

Синтаксис команды для выполнения в режиме конфигурационного файла:

---

find_package(
⟨*имя_пакета*⟩ [⟨*версия*⟩] [EXACT] [QUIET] [REQUIRED]
[[COMPONENTS] [⟨*компонент*$_1$⟩ ... ⟨*компонент*$_n$⟩]]
[CONFIG | NO_MODULE]
[NAMES ⟨*имя*$_1$⟩ ... ⟨*имя*$_n$⟩])

Здесь первый аргумент определяет имя пакета, после которого может быть указана версия. Команда найдёт пакет в системе, совместимый с указанной версией. Если таких пакетов окажется несколько, будет получена информация о произвольном из них.

— При помощи необязательного аргумента `EXACT` можно определить, что необходим поиск только указанной версии пакета.

— Указание необязательного аргумента `QUIET` приводит к тому, что в случае неудачного поиска не будет выведено сообщения об этом.

— Использование необязательного аргумента `REQUIRED` приводит к тому, что в случае неудачного поиска будет прервана работа CMake, как при использовании команды `message(FATAL_ERROR ...)` (п. 2.4.4). По умолчанию выполнение CMake продолжается, и факт успешности поиска пакета можно определить при помощи проверки значений специальных переменных (см. далее).

— После необязательного аргумента `COMPONENTS` можно указать имена компонент пакета, которые требуется загрузить. Крупные пакеты, такие как Qt, могут состоять из множества компонент, причём для построения конкретных приложений далеко не все из них нужны (см. пример на с. 300). Если указан аргумент `REQUIRED`, аргумент `COMPONENTS` можно пропустить.

— При помощи необязательного аргумента `CONFIG` или его синонима `NO_MODULE` можно явно указать, что команда `find_package()` должна работать в режиме конфигурационного файла. Если не указывать настройки, специфичные для этого режима, синтаксис команды полностью совпадает с синтаксисом для режима модуля поиска. По умолчанию в таком случае CMake сначала пытается запустить команду в режиме модуля поиска, и, если его найти не удаётся, запускается режим конфигурационного файла.

— После необязательного аргумента `NAMES` можно указать одно или несколько альтернативных имён, которые будут использованы вместо имени пакета (см. далее).

Для каждого указанного имени пакета (первый аргумент команды, а также все последующие за аргументом NAMES) команда `find_package()` ищет файлы с именами вида ⟨*имя*⟩`Config.cmake` и ⟨*имя_в_нижнем_регистре*⟩`-config.cmake` в ряде каталогов (аналогично командам `find_file()` и т. п., п. 2.9.2), среди которых основное значение имеют следующие:

1) Каталоги, перечисленные в специальной переменной CMake `CMAKE_PREFIX_PATH`. В системе OS X также используются специальные переменные `CMAKE_FRAMEWORK_PATH` и `CMAKE_APPBUNDLE_PATH` для поиска каркасов и пакетов приложений.

2) Каталоги, перечисленные в переменных окружения с именами вида ⟨*имя_пакета*⟩`_DIR`, а также в одноимённых к переменным CMake из предыдущего пункта.

3) Перечисленные в стандартной переменной окружения `PATH`.

4) Перечисленные в пользовательском *реестре пакетов* — специальной базе имён и путей установки пакетов. В Windows для этого используется ветвь реестра пользователя (внутри корня `HKEY_CURRENT_USER`), в POSIX-совместимых системах — подкаталог `.cmake/packages` домашнего каталога пользователя.

5) Перечисленные в системном реестре пакетов. В Windows для этого используется ветвь системного реестра (внутри корня `HKEY_LOCAL_MACHINE`), в других системах это хранилище не реализовано.

В каждом из этих каталогов команда ищет указанные файлы дополнительно в подкаталогах с фиксированными именами (например, CMake в Windows и т. д.). В случае нахождения файла он запускается на исполнение системой CMake. По результатам поиска системой CMake автоматически заполняется ряд переменных (табл. 2.13).

Таблица 2.13

Переменные, заполняемые по результатам поиска пакета

Переменная	Значение
⟨*имя_пакета*⟩_FOUND	Устанавливается в TRUE, если пакет найден, и в FALSE, если нет
⟨*имя_пакета*⟩_DIR	Переменная кэша, в которую записывается путь к каталогу, содержащему найденный файл конфигурации
⟨*имя_пакета*⟩_CONFIG	Содержит полный путь к файлу конфигурации
⟨*имя_пакета*⟩_CONSIDERED_CONFIGS	Содержит список путей к файлам конфигураций, которые были просмотрены для поиска подходящей версии

Окончание табл. 2.13

Переменная	Значение
⟨*имя_пакета*⟩_CONSIDERED_VERSIONS	Содержит список путей рассмотренных версий, соответствующий списку в предыдущей переменной. Значения этих переменных можно выводить на печать для отладки процесса поиска подходящей версии пакета

Эти переменные можно использовать, например, для проверки успешности поиска.

Пример

Пусть требуется обеспечить построение сложного проекта, часть целей которого использует библиотеки Qt для реализации пользовательского интерфейса. Если эти библиотеки отсутствуют в системе, требуется только выполнить построение остальных целей. Эту задачу можно решить при помощи следующего фрагмента кода:

```
# ...

find_package(Qt5Widgets)
```

```
set(MY_BUILD_GUI_APP ${Qt5Widgets_FOUND})
if(MY_BUILD_GUI_APP)
  add_executable(
    my_gui my_gui.cpp main_window.cpp main_window.h)
  target_link_libraries(my_gui Qt5::Widgets)
  # ...
else()
  message(
    WARNING "The GUI part will not be built")
endif()
```

Здесь используется команда `find_package()` для поиска пакета `Qt5Widgets`. Результат успешности поиска записывается в переменную `Qt5Widgets_FOUND`, значение которой сохраняется в переменную `MY_BUILD_GUI_APP` для дальнейшего использования. В случае успеха к проекту добавляется цель `my_gui`, которая связывается с библиотекой Qt Widgets (см. п. 3.3.1). *

Синтаксис команды `find_package()` для выполнения в режиме модуля поиска:

```
find_package(
  ⟨имя_пакета⟩ [⟨версия⟩] [EXACT] [QUIET] [MODULE]
  [REQUIRED]
  [[COMPONENTS] [⟨компонент1,1⟩ ... ⟨компонент1,m⟩]]
  [OPTIONAL_COMPONENTS ⟨компонент2,1⟩ ... ⟨компонент2,n⟩])
```

Большинство аргументов этой формы команды совпадает с аргументами команды в режиме конфигурационного файла. Ниже приведено описание аргументов, специфических для режима модуля поиска:

— Необязательный аргумент `MODULE` явно указывает, что команда должна запускаться в этом режиме и не запускаться в режиме конфигурационного файла, если соответствующий модуль поиска не будет найден.

— После необязательного аргумента `OPTIONAL_COMPONENTS` перечисляются имена необязательных компонент пакета.

В этом режиме модуль поиска отвечает за заполнение переменных, перечисленных в табл. 2.13, а также любых других, приведённых в документации к модулю.

Замечание: команде `find_package()` необходимо передавать имя пакета точно в тех же регистрах символов, как оно указано в имени файла модуля поиска или конфигурационного файла (если только он не имеет форму нижнего регистра, см. выше). Например, нужно указывать имя «`Git`» вместо «`git`» (п. 3.5.1). Эта ошибка не будет проявляться на платформах с нечувствительными к регистру имён файловыми системами. ▲

2.11.2. `get_property()`, `set_property()`

`get_property``(`

⟨*имя_переменной*⟩

⟨*сущность*⟩

PROPERTY ⟨*имя_свойства*⟩
[SET | DEFINED])

⟨*сущность*⟩ ::=
GLOBAL |
DIRECTORY [⟨*каталог*⟩] |
TARGET ⟨*имя_цели*⟩ |
SOURCE ⟨*файл*⟩ |
INSTALL ⟨*файл*⟩ |
TEST ⟨*имя_теста*⟩ |
CACHE ⟨*имя_переменной*⟩ |
VARIABLE

set_property(
⟨*сущности*⟩
[APPEND] [APPEND_STRING]
PROPERTY ⟨*имя_свойства*⟩
[⟨*значение*$_1$⟩ ... ⟨*значение*$_m$⟩])

⟨*сущности*⟩ ::=
GLOBAL |
DIRECTORY [⟨*каталог*⟩] |
TARGET [⟨*имя_цели*$_1$⟩ ... ⟨*имя_цели*$_n$⟩] |
SOURCE [⟨*файл*$_1$⟩ ... ⟨*файл*$_n$⟩] |
INSTALL [⟨*файл*$_1$⟩ ... ⟨*файл*$_n$⟩] |
TEST [⟨*имя_теста*$_1$⟩ ... ⟨*имя_теста*$_n$⟩] |
CACHE [⟨*имя_переменной*$_1$⟩ ... ⟨*имя_переменной*$_n$⟩]

Команды `get_property()` и `set_property()` используются для считывания и установки значений свойств (п. 2.2.4). При считывании значения обязательно указываются:

1) имя переменной, в которую будет считан результат выполнения команды;
2) тип и, если требуется, название сущности, для которой считывается свойство;
3) имя свойства после аргумента `PROPERTY`.

Тип и название сущности определяются с помощью следующих аргументов:

`GLOBAL`: глобальное свойство, имя объекта не требуется.

`DIRECTORY`: свойство каталога подпроекта (в котором располагается файл `CMakeLists.txt`). По умолчанию опрашивается свойство каталога, обрабатываемого CMake в настоящий момент, однако можно задать ранее обработанный каталог. Указывается полный или относительный путь.

`TARGET`: свойство цели. Указывается имя цели, ранее определённой при помощи команды `add_executable()` (п. 2.5.1) и т. п.

`SOURCE`: свойство исходного файла. Указывается путь к файлу (имеет смысл указывать относительный путь по отношению к каталогу текущего проекта). Файл должен существовать на диске, но не обязательно должен принадлежать какой-либо из определённых целей. Устанавливаемые свой-

ства влияют только на цели, добавляемые в текущем подпроекте.

INSTALL: свойство устанавливаемого файла. Указывается путь к файлу (по отношению к каталогу установки).

TEST: свойство теста. Указывается имя теста, ранее определённое при помощи команды `add_test()` (п. 2.10.2).

CACHE: свойство переменной кэша. Указывается имя переменной.

VARIABLE: этот аргумент не задаёт никакой сущности, для которой будет опрашиваться свойство. Он позволяет использовать команду `get_property()` для определения состояния (см. ниже) переменной, имя которой указывается в этом случае после аргумента `PROPERTY`.

— Необязательный аргумент команды `get_property()` (или его отсутствие) определяет, какую информацию о состоянии свойства нужно записать в переменную:

По умолчанию: возвращается значение свойства. Если свойство ранее не было установлено или его не существует, возвращается пустая строка.

SET: возвращается логическое значение, определяющее, было ли ранее установлено свойство.

DEFINED: возвращается логическое значение, указывающее на то, существует ли такое свойство.

При установке значений свойств обязательно указываются:

1) тип и, возможно, названия нескольких сущностей, для которых устанавливается свойство;
2) имя свойства после аргумента `PROPERTY`;
3) набор значений, формирующих список.

Особенностью команды `set_property()` является возможность установить одно значение для одного и того же свойства сразу нескольким сущностям одного типа (кроме каталога) при помощи одной команды. Передаваемые команде последние аргументы формируют список (п. 2.2.3), определяющий присваиваемое значение.

— Наличие или отсутствие остальных необязательных аргументов команды `set_property()` определяет способ добавления нового значения к существующим значениям свойств:

По умолчанию: значения будут присвоены свойствам вместо их предыдущих значений.

`APPEND`: значения будут добавлены в конец существующих значений свойств, рассматриваемых как списки (с разделением элементов точками с запятой).

`APPEND_STRING`: значения будут добавлены в конец существующих значений свойств, рассматриваемых как строки (конкатенацией строк).

Замечание: многие свойства предпочтительнее устанавливать не при помощи команды `set_property()`, а при помощи специализированных команд. Например, в графическом интерфейсе

утилит CMake есть возможность отображать не все переменные кэша, а лишь основные. За это отвечает свойство ADVANCED переменной кэша, определяющее, будет ли переменная по умолчанию скрыта. Для установки этого свойства значения истины удобнее использовать команду mark_as_advanced(), в аргументах которой перечисляются имена переменных. ▲

Примеры

1)

```
add_executable(exec main.cpp file1.cpp file1.h)

set_property(
  TARGET exec
  PROPERTY OUTPUT_NAME
  my_prog)
```

Здесь устанавливается свойство цели exec с именем OUTPUT_NAME, которое определяет основу имени для выходного файла (без возможного расширения «.exe» и т. д.). По умолчанию если значение свойства не установлено, в качестве базы имени используется логическое имя цели (exec).

2)

```
foreach(LANG C CXX)
  #
  get_property(
    FEATURES
    GLOBAL
    PROPERTY CMAKE_${LANG}_KNOWN_FEATURES)
```

```
  #
  message(STATUS "Known ${LANG} features:")
  foreach(FEATURE ${FEATURES})
    message(STATUS "  ${FEATURE}")
  endforeach()
  #
endforeach()
```

Здесь при помощи первой команды foreach() (п. 2.8.3) реализована обработка глобальных свойств с именами CMAKE_C_KNOWN_FEATURES и CMAKE_CXX_KNOWN_FEATURES (см. п. 2.6.6). Для каждого из этих свойств считывается его значение в переменную FEATURES, которая затем распечатывается поэлементно при помощи вложенного цикла.

3)

```
set(
  MY_SETTING_VALUES One Two Three)

set(
  MY_SETTING_DESCRIPTION
  "Sample setting, possible values: "
  "${MY_SETTING_VALUES}")

set(
  MY_SETTING One
  CACHE STRING "${MY_SETTING_DESCRIPTION}")

set_property(
```

```
  CACHE MY_SETTING
  PROPERTY STRINGS
  ${MY_SETTING_VALUES})

list(FIND MY_SETTING_VALUES ${MY_SETTING} INDEX)
if(INDEX EQUAL -1)
  set(
    MY_SETTING One
    CACHE STRING "${MY_SETTING_DESCRIPTION}"
    FORCE)
endif()
```

Здесь в кэше создаётся переменная MY_SETTING, для которой устанавливается свойство STRINGS. Оно определяет набор возможных значений для переменной, которые, например, будут отображаться в выпадающем списке утилиты CMake с графическим интерфейсом (рис. 2.14). Дальше выполняется проверка принадлежности текущего значения переменной списку. Если переменной кэша присвоено недопустимое значение (например, из командной строки), оно заменяется на одно из допустимых. *

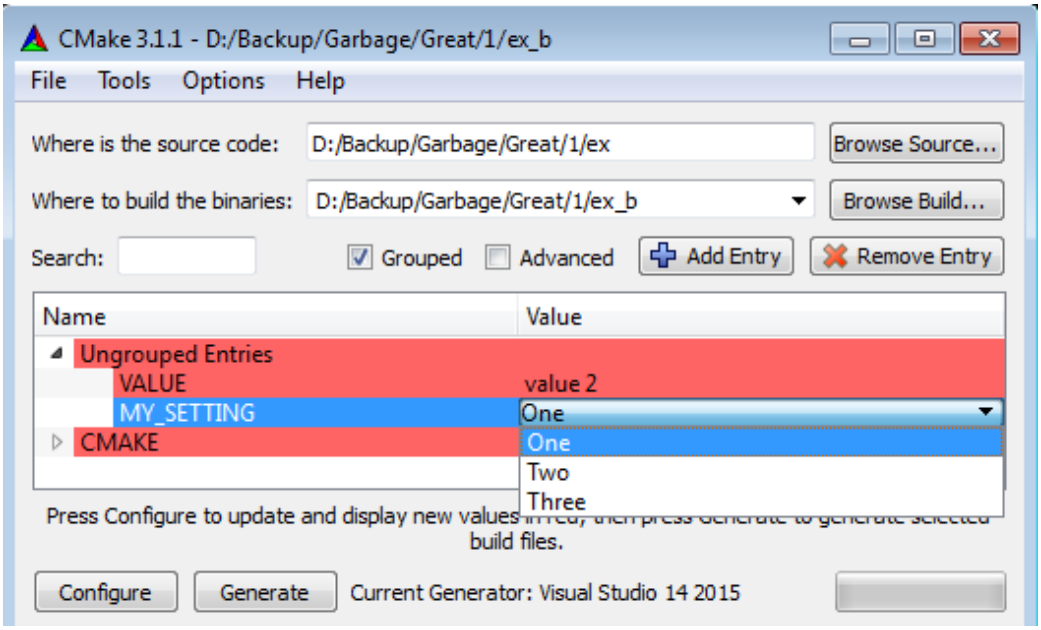

Рис. 2.14. Редактирование переменной кэша с набором допустимых значений при помощи выпадающего списка

2.12. Поддержка нескольких конфигураций построения

2.12.1. Виды конфигураций

В описании проекта многие настройки инструментов (компилятор, компоновщик и т. д.) зависят от операционной системы, используемого набора инструментов построения, версий найденных пакетов и т. д. Это реализуется при помощи команды `if()` (п. 2.8.1), проверяющей значения специальных переменных.

Пример

Пусть требуется использовать возможности стандарта C++11. Для этого компилятору gcc нужно передать специальный ключ

командной строки. Кроме того, в случае его использования нужно включить режим вывода всех предупреждений, а также подключить библиотеки `libpthread` и `libdl`. Это можно реализовать при помощи следующего фрагмента:

```
if(CMAKE_COMPILER_IS_GNUCXX)
  add_compile_options(-Wall -std=c++11)
  target_link_libraries(my_app -lpthread -ldl)
endif()
```

*

Также настройки инструментов могут зависеть и от используемой конфигурации. Для проверки конфигурации используется специальная переменная `CMAKE_BUILD_TYPE`.

Пример

```
if(CMAKE_COMPILER_IS_GNUCXX)
  if(CMAKE_BUILD_TYPE STREQUAL Debug)
    add_definitions(-D_DEBUG)
  else()
    add_definitions(-DNDEBUG)
  endif()
endif()
```

*

По умолчанию генерируется конфигурация с пустым именем. Переменной `CMAKE_BUILD_TYPE` можно присваивать значение, например, в командной строке при вызове CMake:

```
cmake -D CMAKE_BUILD_TYPE=Debug ...
```

Названия всех доступных конфигураций перечислены в специальной переменной CMAKE_CONFIGURATION_TYPES. В зависимости от конфигурации компилятору передаются различные ключи командной строки. По умолчанию в CMake определены 4 конфигурации, характеристики которых приведены ниже (наличие конкретных настроек зависит от компилятора).

Debug: конфигурация для отладки:

- Включена отладочная информация.
- Включены проверки на ошибки времени исполнения (выход за границы стека и т. п.).
- Отключены оптимизации.
- Отключена подстановка функций (inlining).
- Определён символ препроцессора _DEBUG, что позволяет условно включать в коде дополнительные проверки корректности.

Release: конфигурация для поставки конечному пользователю:

- Отключена отладочная информация.
- Отключены проверки на ошибки времени исполнения.
- Включены оптимизации, направленные на создание более быстродействующего кода.

- Компилятору позволяется выполнять подстановку любых функций, которые он сочтёт нужными, в дополнение к явно указанным (`inline`).
- Определён символ препроцессора `NDEBUG`, что отключает проверки при помощи стандартного макроса `assert()`.

`RelWithDebInfo`: конфигурация для поставки конечному пользователю с включённой отладочной информацией. Может быть полезной для получения от пользователя осмысленных отчётов об ошибках времени исполнения:

- Включена отладочная информация.
- Отключены проверки на ошибки времени исполнения.
- Включены оптимизации, направленные на создание более быстродействующего кода.
- Компилятору позволяется выполнять подстановку любых функций, которые он сочтёт нужными, в дополнение к явно указанным (`inline`).
- Определён символ препроцессора `NDEBUG`.

`MinSizeRel`: конфигурация для поставки конечному пользователю с кодом минимального размера:

- Отключена отладочная информация.
- Отключены проверки на ошибки времени исполнения.

— Включены оптимизации, направленные на создание более компактного кода.

— Компилятору позволяется выполнять подстановку только функций, указанных явно при помощи ключевых слов `inline` и т. п.

— Определён символ препроцессора `NDEBUG`.

Однако подход к настройке в зависимости от конфигурации, продемонстрированный в последнем примере, работает не всегда. Дело в том, что некоторые конечные системы построения (например, интегрированные среды разработки) поддерживают множественные конфигурации. Файлы проектов для них хранят настройки для всех конфигураций одновременно. Используемую конфигурацию можно выбирать в момент запуска построения. В процессе генерации проектов для таких систем CMake исполняет команды из описания проекта только один раз, а не отдельно для каждой из конфигураций[9]. Чтобы определить, какие из настроек должны действовать для какой из конфигураций, в CMake доступны следующие средства:

— Подстановки путей к целям вместо их логических имён. Команды `target_link_libraries()` (п. 2.6.7), `add_test()` (п. 2.10.2) и т. д. могут принимать в качестве аргументов логические имена целей, например созданные командой `add_library()` (п. 2.5.2). Во время генерирования файлов для конечных систем построения эти имена автоматиче-

[9] В отличие от инструмента qmake, который использует два прохода для генерирования make-файлов конфигураций Debug и Release и ещё один для создания make-файла, рекурсивно исполняющего два первых.

ски заменяются на полные пути к соответствующим выходным файлам. Например, выходные файлы для разных конфигураций могут помещаться в разные каталоги. Также можно настроить разные имена для выходных файлов разных конфигураций.

— Команде `target_link_libraries()` можно передавать аргументы `debug` и `optimized`, указывающие на то, к каким конфигурациям относятся подключаемые библиотеки.

— Для целого ряда специальных переменных и свойств CMake существуют их аналоги с суффиксами вида _⟨*имя_конфигурации_в_верхнем_регистре*⟩, которые относятся к конкретной конфигурации. Например, переменные `CMAKE_`⟨*язык*⟩`_FLAGS` (`CMAKE_C_FLAGS`, `CMAKE_CXX_FLAGS` и т. д.) содержат список ключей, передаваемых компилятору во всех конфигурациях. Кроме этого, переменные `CMAKE_C_FLAGS_DEBUG` и т. д. содержат дополнительные ключи для разных конфигураций.

— Выражения генераторов (см. далее).

2.12.2. Выражения генераторов

Выражения генераторов (generator expressions) могут встречаться внутри строковых значений CMake. Если эти значения передать в качестве аргументов некоторым командам, то во время генерации файлов для конечных систем построения эти выражения будут заменены на другие значения в зависимости от генерируемой конфигурации и других условий. Выражения

генераторов имеют вид «$<...>» и могут быть вложены друг в друга. В последнем случае сначала вычисляются внутренние выражения, затем внешние.

Пример

Попробуем проверить, как вычисляются выражения генераторов. Для примера рассмотрим выражение «$<CONFIG>», которое заменяется названием используемой конфигурации. Можно передать эту строку команде `message()`:

```
message(STATUS $<CONFIG>)
```

При запуске инструмента CMake команда выведет следующее сообщение:

```
-- $<CONFIG>
```

Причина того, что выражение не было вычислено, очевидна: команда `message()` (как и все остальные) выполняется во время работы инструмента CMake, в то время как значения выражений генераторов могут быть известны только на момент запуска построения, когда пользователь выберет конкретную конфигурацию.

Таким образом, команды CMake сами не вычисляют выражений генераторов. Однако они передают их генераторам, которые и выполняют вычисление для каждой из конфигураций (из чего и происходит их название).

Чтобы всё же получить возможность увидеть, как вычисляются выражения генераторов, нужно найти какой-либо способ распечатать строки с теми же выражениями, но уже

во время обработки проекта конечной системой построения. Для этого подходят команды add_custom_target() (п. 2.10.4) и add_custom_command() (п. 2.10.5). Воспользуемся для примера второй из них. В этой команде необходимо будет указать команду оболочки, выводящую в журнал построения заданную строку. В примере на с. 199 уже была использована команда echo, однако для обеспечения кроссплатформенности вместо неё можно воспользоваться самим инструментом CMake:

```
add_executable(my_prog my_prog.cpp)

set(CMD COMMAND "${CMAKE_COMMAND}" -E echo)

add_custom_command(
  TARGET my_prog
  PRE_BUILD
  ${CMD} "Target info:"
  ${CMD} "  Config:   $<CONFIG>"
  ${CMD} "  Platform: $<PLATFORM_ID>"
  ${CMD} "  Out file: $<TARGET_FILE:my_prog>"
  )
```

Здесь для удобства список повторяющихся аргументов команды add_custom_command() присвоен переменной CMD. Команда add_custom_command() добавляет к цели перед построением команды вывода сообщений. Специальная переменная CMAKE_COMMAND содержит путь к инструменту CMake. Если его

вызвать с аргументами «`-E echo` ⟨*строка*⟩», он выведет указанную строку в стандартный поток.

Часть содержимого журнала построения проекта интегрированной средой Visual Studio 2015 приведена ниже:

```
1>------ Сборка начата: проект: ZERO_CHECK, Конфигурация: Debug Win32
2>------ Сборка начата: проект: my_prog, Конфигурация: Debug Win32
2>  Target info:
2>    Config:   Debug
2>    Platform: Windows
2>    Out file: D:/Backup/Garbage/Great/1/ex_b/Debug/my_prog.exe
...
```

*

Замечания:

— Инструмент CMake с ключом «`-E`» является кроссплатформенной заменой основных команд оболочки (`mkdir`, `cp` и т. д.). Полный список операций можно получить, вызвав команду:

```
cmake -E help
```

— Некоторые из выражений генераторов (например, `C_COMPILER_ID`) нельзя вывести на печать подобным способом, так как их нельзя использовать для вызова инструментов при помощи команд `add_custom_target()` и `add_custom_command()` (см. далее). Вместо них можно использовать соответствующие специальные переменные.

▲

Выражения генераторов можно указывать внутри аргументов команд, определяющих следующие настройки:

— Исходные файлы в командах определения целей (`add_executable()`, п. 2.5.1 и т. д.).

— Настройки целей во всех соответствующих командах установки свойств целей (`add_compile_options()`, п. 2.6.3, `include_directories()`, п. 2.6.1 и т. д.).

— Команды оболочки для вызова инструментов и их рабочий каталог в командах `add_custom_target()` (п. 2.10.4), `add_custom_command()` (п. 2.10.5) и `add_test()` (п. 2.10.2).

— Файлы для установки в команде `install()` (п. 2.10.3).

— Имена и значения свойств файлов для установки в команде `set_property()` (п. 2.11.2).

Также выражения генераторов можно использовать внутри значений при установке командой `set_property()` следующих свойств:

— Все свойства каталогов проектов, целей и тестов, которые устанавливаются указанными выше командами (т. е. аргументы команд сами могут содержать выражения генераторов). Например, команда `target_compile_options()` (п. 2.6.5) устанавливает значения свойств `COMPILE_OPTIONS` (настройки компилятора для построения текущей цели) и `INTERFACE_COMPILE_OPTIONS` (для построения зависимых целей, подключающих библиотеку текущей цели).

— Некоторые другие свойства, например свойство каталога проекта ADDITIONAL_MAKE_CLEAN_FILES. Оно содержит список дополнительных файлов для удаления целью clean, генерируемой для системы make.

2.12.3. Информационные выражения

Результатом вычисления информационных выражений генераторов (табл. 2.14) является их замена строковой информацией о текущей (к которой относится команда с выражением) или заданной цели. Здесь и далее в таблицах выражения генераторов приведены без охватывающих символов «$<...>». При необходимости указаны ограничения на использование тех или иных выражений.

Таблица 2.14

Информационные выражения генераторов

Выражение	Описание
CONFIG	Название конфигурации
PLATFORM_ID	Целевая операционная система
Команды должны относиться к целям, созданным командой add_executable() или add_library()	
C_COMPILER_ID	Имя компилятора C
CXX_COMPILER_ID	Имя компилятора C++
C_COMPILER_VERSION	Версия компилятора C
CXX_COMPILER_VERSION	Версия компилятора C++

Продолжение табл. 2.14

Выражение	**Описание**
`TARGET_PROPERTY:`[⟨*цель*⟩`,`]⟨*св-во*⟩	Значение заданного свойства для заданной цели или (по умолчанию) цели, для которой исполняется команда
⟨*цель*⟩ должна быть создана командой `add_executable()` или `add_library()`	
`TARGET_FILE:`⟨*цель*⟩	Путь к основному файлу, создаваемому целью
`TARGET_FILE_NAME:`⟨*цель*⟩	Имя основного файла, создаваемого целью
`TARGET_FILE_DIR:`⟨*цель*⟩	Путь к каталогу основного файла, создаваемого целью
⟨*цель*⟩ должна быть создана командой `add_library()` или `add_executable()` со свойством `ENABLE_EXPORTS` *	
`TARGET_LINKER_FILE:`⟨*цель*⟩	Путь к выходному файлу, используемому для связывания с библиотекой
`TARGET_LINKER_FILE_NAME:`⟨*цель*⟩	Имя выходного файла, используемого для связывания с библиотекой

Продолжение табл. 2.14

Выражение	**Описание**
`TARGET_LINKER_FILE_DIR:`⟨*цель*⟩	Путь к каталогу выходного файла, используемого для связывания с библиотекой
⟨*цель*⟩ должна создавать разделяемую библиотеку для POSIX-совместимой системы	
`TARGET_SONAME_FILE:`⟨*цель*⟩	Путь к выходному файлу разделяемой библиотеки
`TARGET_SONAME_FILE_NAME:`⟨*цель*⟩	Имя выходного файла разделяемой библиотеки
`TARGET_SONAME_FILE_DIR:`⟨*цель*⟩	Путь к каталогу выходного файла разделяемой библиотеки
⟨*цель*⟩ должна быть создана командой `add_executable()` или `add_library()` и использовать компоновщик Visual Studio	
`TARGET_PDB_FILE:`⟨*цель*⟩	Путь к выходному файлу с отладочной информацией (`*.pdb`)
`TARGET_PDB_FILE_NAME:`⟨*цель*⟩	Имя выходного файла с отладочной информацией

Окончание табл. 2.14

Выражение	Описание
`TARGET_PDB_FILE_DIR:`⟨*цель*⟩	Путь к каталогу выходного файла с отладочной информацией
Используется в команде `install(EXPORT ...)`	
`INSTALL_PREFIX`	Путь к каталогу установки

* Если свойство `ENABLE_EXPORTS` цели исполняемого файла установлено в истину, CMake считает, что исполняемый файл можно использовать в качестве динамической библиотеки (такая возможность есть в большинстве систем). Во время построения такой цели при необходимости создаётся библиотека импорта, саму цель можно использовать в команде `target_link_libraries()` для подключения библиотеки к другой цели.

Пример

Пусть требуется создать проект разделяемой библиотеки для использования в качестве загружаемого модуля.

Ниже приведён исходный код разделяемой библиотеки (файл `my_plugin/my_plugin.cpp`):

```
extern "C" int my_plugin_f()
{
  return 31;
```

```
}
```

Описание цели библиотеки выглядит следующим образом (файл `my_plugin/CMakeLists.txt`):

```
add_library(my_plugin MODULE my_plugin.cpp)
```

Теперь возникает задача создания теста для библиотеки. Очевидно, тестовая программа должна загружать библиотеку по заданному пути к ней, используя для этого, например, функции интерфейса POSIX. Поскольку библиотека и тест могут собираться в разных подкаталогах, надёжным способом ссылки на библиотеку будет указание полного пути к ней, который можно передавать программе через аргумент командной строки.

Исходный код теста (файл `my_test/my_test.cpp`):

```
#include <dlfcn.h>
#include <iostream>
#include <cstddef>

using namespace std;

int main(int nArgC, char *apszArgV[])
{
  if (nArgC < 2)
  {
    cerr << "Wrong usage" << endl;
    return 1;
```

```
  }    // if (nArgC < 2)
  //
  void *pvHandle = dlopen(
    apszArgV[1], RTLD_LOCAL | RTLD_LAZY);
  if (NULL == pvHandle)
  {
    cerr << "Cannot open library" << apszArgV[1] <<
      ": " << dlerror() << endl;
    return 2;
  }    // if (NULL == pvHandle)
  //
  int nRet = 0;
  //
  typedef int (*PFN_T)();
  const char *pcszName = "my_plugin_f";
  PFN_T pfn = (PFN_T) dlsym(pvHandle, pcszName);
  if (NULL == pfn)
  {
    cerr << "Cannot get function " << pcszName <<
      ": " << dlerror() << endl;
    nRet = 3;
  }    // if (NULL == pfn)
  else
  {
    const int cn = (*pfn)();
    if (cn != 31)
    {
```

```
        cerr << "The return value is different: " <<
          cn << endl;
        nRet = 4;
      }    // if (cn != 31)
    }    // if (NULL == pfn) (else)
    //
    dlclose(pvHandle);
    //
    return nRet;
}    // main()
```

Здесь сначала проверяется, что программе был передан хотя бы один аргумент. Если это так, он передаётся функции POSIX `dlopen()`, которая выполняет динамическую загрузку модуля [13]. Если загрузка выполнена успешно, при помощи функции `dlsym()` в модуле выполняется поиск адреса функции с именем «`my_plugin_f`», который присваивается переменной-указателю на функцию `pfn`. Если эта функция найдена, выполняется её косвенный вызов и проверка возвращённого значения.

Описание цели теста (файл `my_test/CMakeLists.txt`):

```
add_executable(my_test my_test.cpp)

target_link_libraries(my_test -ldl)
```

Здесь цель связывается с системной библиотекой `libdl`, которая экспортирует функции `dlopen()`, `dlsym()` и т. д.

Описание проекта верхнего уровня имеет следующий вид (файл `CMakeLists.txt`):

```
cmake_minimum_required(VERSION 2.8)

enable_testing()

add_subdirectory(my_plugin)
add_subdirectory(my_test)

add_test(
  NAME my_plugin_test
  COMMAND my_test "$<TARGET_FILE:my_plugin>")
```

Здесь командой `add_test()` (п. 2.10.2) добавляется тест, который запускает приложение-результат сборки цели `my_test` с путём к библиотеке в аргументе командной строки. Последний получается вычислением выражения `$<TARGET_FILE:my_plugin>`. *

Замечания:

— В приведённом примере при определении теста невозможно обойтись без выражения генераторов, записав команду `add_test()` подобным образом:

```
add_test(
  NAME my_plugin_test
  COMMAND my_test my_plugin)
```

Хотя имя цели `my_test` заменяется на путь к исполняемому файлу, поскольку указано вместо имени команды, имя `my_plugin` уже указано в качестве её аргумента. Поэтому

при запуске теста оно будет передано программе в неизменном виде.

— Тестовое приложение можно также реализовать с использованием функций Windows API `LoadLibrary()`, `GetProcAddress()` и т. д. ▲

2.12.4. Логические выражения

Логические выражения генераторов (табл. 2.15) после вычисления заменяются на «1» или «0» в зависимости от истинности их условия.

Таблица 2.15

Логические выражения генераторов

Выражение	Условие
Основные	
`STREQUAL:`⟨*строка*$_1$⟩`,`⟨*строка*$_2$⟩	⟨*строка*$_1$⟩ = ⟨*строка*$_2$⟩
`EQUAL:`⟨*число*$_1$⟩`,`⟨*число*$_2$⟩	⟨*число*$_1$⟩ = ⟨*число*$_2$⟩
`VERSION_EQUAL:`⟨*версия*$_1$⟩`,`⟨*версия*$_2$⟩	⟨*версия*$_1$⟩ = ⟨*версия*$_2$⟩
`VERSION_LESS:`⟨*версия*$_1$⟩`,`⟨*версия*$_2$⟩	⟨*версия*$_1$⟩ < ⟨*версия*$_2$⟩
`VERSION_GREATER:`⟨*версия*$_1$⟩`,`⟨*версия*$_2$⟩	⟨*версия*$_1$⟩ > ⟨*версия*$_2$⟩
`BOOL:`⟨*строка*⟩	Истинность/ложность строки в соответствии с табл. 2.1
Составные	

Продолжение табл. 2.15

Выражение	**Условие**
`NOT:`⟨*знач*⟩	$\begin{cases} 0, & \text{если } \langle знач \rangle = 1 \\ 1, & \text{иначе} \end{cases}$
`AND:`⟨*знач*$_1$⟩$\{$`,`⟨*знач*$_n$⟩$\}$	$\begin{cases} 1, & \text{если } \forall k\ \langle знач_k \rangle = 1 \\ 0, & \text{иначе} \end{cases}$
`OR:`⟨*знач*$_1$⟩$\{$`,`⟨*знач*$_n$⟩$\}$	$\begin{cases} 0, & \text{если } \forall k\ \langle знач_k \rangle = 0 \\ 1, & \text{иначе} \end{cases}$
Системные *	
`CONFIG:`⟨*строка*⟩	Равенство используемой конфигурации строке без учёта регистра
`PLATFORM_ID:`⟨*строка*⟩	Равенство целевой платформы строке
`C_COMPILER_ID:`⟨*строка*⟩	Равенство имени компилятора C строке
`CXX_COMPILER_ID:`⟨*строка*⟩	Равенство имени компилятора C++ строке

Окончание табл. 2.15

Выражение	Условие
`C_COMPILER_VERSION:`⟨*версия*⟩	Равенство версии компилятора C заданной
`CXX_COMPILER_VERSION:`⟨*версия*⟩	Равенство версии компилятора C++ заданной
`COMPILE_FEATURES:`⟨*возм*$_1$⟩[`,`⟨*возм*$_k$⟩]	Доступность всех возможностей в используемом компиляторе (п. 2.6.6) **

* Эти выражения введены для удобства, их можно заменить комбинацией нескольких других выражений.

** При помощи этого выражения можно условно включать в проект реализацию участков кода на основе новых возможностей компилятора и без их использования, если компилятор их не поддерживает (например, при помощи шаблонов C++11 с переменным количеством параметров или с ограниченным, поддерживаемых C++98).

2.12.5. Преобразующие выражения

Преобразующие выражения (табл. 2.16) вычисляются в зависимости от входных аргументов.

Таблица 2.16

Преобразующие выражения генераторов

Выражение	Результат
`0:`⟨*строка*⟩	Пустая строка *
`1:`⟨*строка*⟩	⟨*строка*⟩ *
`LOWER_CASE:`⟨*строка*⟩	⟨*строка*⟩ в нижнем регистре
`UPPERR_CASE:`⟨*строка*⟩	⟨*строка*⟩ в верхнем регистре
`JOIN:`⟨*строка*$_1$⟩`,`⟨*строка*$_2$⟩	⟨*строка*$_1$⟩, в которой все вхождения точки с запятой заменены на ⟨*строка*$_2$⟩ **
`MAKE_C_IDENTIFIER:`⟨*строка*⟩	⟨*строка*⟩, преобразованная к виду, который может быть использован в качестве корректного идентификатора в языке C ***
Системные	
`LINK_ONLY:`⟨*строка*⟩	Пустая строка, если выражение вычисляется при определении транзитивных требований (п. 2.6.7), иначе ⟨*строка*⟩
`BUILD_INTERFACE:`⟨*строка*⟩	⟨*строка*⟩, если цель используется другой целью в пределах того же проекта, иначе пустая строка ****

Окончание табл. 2.16

Выражение	Результат
`INSTALL_INTERFACE:`⟨*строка*⟩	⟨*строка*⟩, если цель экспортируется командой `install(EXPORT ...)`, иначе пустая строка ****

* Предполагается, что эти выражения будут использоваться совместно с логическими выражениями (см. табл. 2.15), которые заменяются на `0` или `1` (пример: `$<$<CONFIG:debug>:-D_DEBUG>`).

** Предполагается, что это выражение будет использоваться для обработки свойств целей, представляющих списки (см. пример ниже).

*** Например, идентификатор CMake «`01-hello`» будет преобразован к «`_01_hello`».

**** Предполагается, что эти команды будут использоваться в командах `target_include_directories()` (п. 2.6.2) и т. п. для цели библиотеки, имеющей правило для установки (`install(EXPORT ...)`, п. 2.10.3).

Примеры

1)
```
set(
  DEFS
  "$<TARGET_PROPERTY:prog,COMPILE_DEFINITIONS>")
```

```
target_compile_options(
  prog
  PRIVATE "-D $<JOIN:${DEFS}, -D >")
```

Здесь для удобства строка с выражением разбита на две: первая в переменной DEFS содержит выражение, которое при вычислении заменяется на значение свойства COMPILE_DEFINITIONS цели prog. Это свойство содержит список определений символов препроцессора, заполняемый командами add_definitions() (п. 2.6.3) и target_compile_definitions() (п. 2.6.4). Внутри команды target_compile_options() (п. 2.6.5) эта строка подставляется внутрь другого выражения JOIN, которое при вычислении заменяет в этом списке все точки с запятой, разделяющие его элементы, на строки « -D ». Строка «-D » перед выражением добавляется перед первым определением. Таким образом, всё выражение после вычисления генератором развернётся в список аргументов формата компилятора gcc, определяющих заданные символы («-D MY_DEF1 -D MY_DEF2 ...») в предположении, что свойство COMPILE_DEFINITIONS содержит непустой список.

2) Пусть проект приложения реализует некоторые возможности средствами, зависящими от целевой операционной системы. Например, в Windows он может использовать Windows API, а в POSIX-совместимых системах — соответственно API POSIX. Разработчик выделил все зависящие

от системы участки кода в один или несколько заголовочных файлов. Для каждого интерфейса прикладных программ существует своя версия заголовочных файлов. Разные версии называются одинаково, но расположены в разных подкаталогах (рис. 2.15).

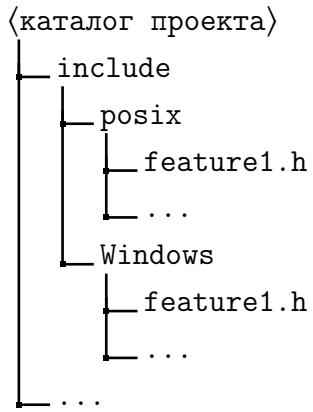

Рис. 2.15. Часть структуры каталога проекта с разными версиями заголовочных файлов

Теперь для правильной сборки приложения достаточно обеспечить передачу компилятору нужного каталога для поиска заголовочных файлов, чтобы в исходном коде директивы вида:

```
#include "feature1.h"
```

подключали нужную версию файла. Выражение генераторов $<PLATFORM_ID> заменяется на имя операционной системы, его можно использовать при определении каталогов поиска:

```
target_include_directories(
  prog
```

```
  PRIVATE
  "${CMAKE_SOURCE_DIR}/include/$<PLATFORM_ID>")
```

Тогда в каталоге проекта include нужно создать подкаталоги с именами операционных систем, для которых должен быть собран проект. Однако такой подход приведёт к ненужному дублированию заголовочных файлов, так как целый ряд операционных систем поддерживают интерфейс POSIX. Чтобы избежать использования символических ссылок в каталоге проекта, можно усложнить используемое выражение генераторов следующим образом:

```
set(
  POSIX_SYSTEMS
  Darwin FreeBSD Linux NetBSD OpenBSD)

set(EXPRESSION "")
foreach(SYSTEM IN LISTS POSIX_SYSTEMS)
  set(TEST $<PLATFORM_ID:${SYSTEM}>)
  if(EXPRESSION)
    set(EXPRESSION ${EXPRESSION},${TEST})
  else()
    set(EXPRESSION ${TEST})
  endif()
endforeach()

set(EXPRESSION $<OR:${EXPRESSION}>)
set(INC_DIR "${CMAKE_SOURCE_DIR}/include/")
```

```
target_include_directories(
  prog
  PRIVATE
  $<${EXPRESSION}:${INC_DIR}posix>
  $<$<NOT:${EXPRESSION}>:${INC_DIR}$<PLATFORM_ID>>)
```

Здесь для удобства сложное выражение формируется на основе списка в переменной `POSIX_SYSTEMS`. Цикл `foreach()` генерирует в переменной `EXPRESSION` выражение, проверяющее принадлежность имени системы заданному списку:

```
$<OR:$<PLATFORM_ID:Darwin>,$<PLATFORM_ID:FreeBSD>,
...>
```

Дальше это выражение используется в двух условных выражениях:

```
$<${EXPRESSION}:⟨каталог1⟩>
$<$<NOT:${EXPRESSION}>:⟨каталог2⟩>
```

Первое выражение после вычисления заменяется на строку ⟨*каталог*$_1$⟩, если выражение в `EXPRESSION` окажется истинным, и на пустую строку иначе. Второе выражение, наоборот, заменится на ⟨*каталог*$_2$⟩, если условие будет ложным. При этом первый путь будет указывать на подкаталог «`posix`», а второй — на подкаталог с именем целевой системы. *

Замечание: здесь команде `target_include_directories()` передаются абсолютные пути. Если ей передать относительные пути, инструмент CMake выведет сообщение об ошибке. Причиной её будет использование выражений генераторов в путях. Так как относительные пути преобразуются в абсолютные на этапе исполнения команд CMake, инструмент не может знать, во что преобразуются выражения генераторов на более позднем этапе, и не сможет выполнить корректное преобразование путей в абсолютные. ▲

Пример

В замечании к примеру на с. 188 утверждалось, что использование команды `target_include_directories()` (п. 2.6.2) для библиотеки, устанавливаемой командой `install(EXPORT ...)` (п. 2.10.3), осложнено тем, что каталог проекта библиотеки и каталог её установки не совпадают. При сборке подключающих библиотеку целей, определённых в том же проекте верхнего уровня, что и сама библиотека, нужно использовать один каталог поиска, а при сборке целей из других проектов — другой.

Рассмотрим решение этой проблемы с помощью выражений генераторов. Пусть структура каталогов проекта и установки библиотеки должна быть такой, как она изображена на рис. 2.16 (пусть для сборки библиотеки используется компилятор gcc или подобный).

Файл `CMakeLists.txt` проекта библиотеки:

```
cmake_minimum_required(VERSION 2.8)
```

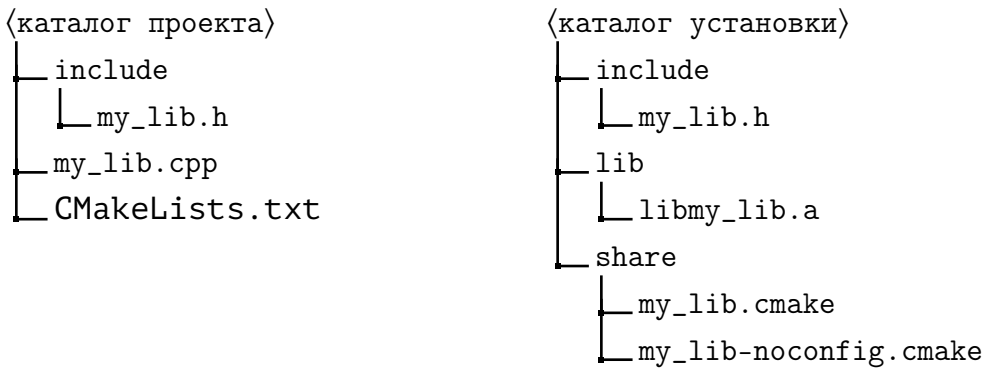

Рис. 2.16. Структура каталогов проекта и установки библиотеки

```
project(my_lib)

add_library(
  my_lib STATIC
  my_lib.cpp include/my_lib.h)

set(DIR "${CMAKE_CURRENT_SOURCE_DIR}")

target_include_directories(
  my_lib
  PUBLIC
    $<BUILD_INTERFACE:${DIR}/include>
    $<INSTALL_INTERFACE:include>)

install(
  TARGETS my_lib
  EXPORT my_lib
  RUNTIME
```

```
    DESTINATION bin
  LIBRARY
    DESTINATION lib
  ARCHIVE
    DESTINATION lib)

install(
  FILES include/my_lib.h
  DESTINATION include)

install(
  EXPORT my_lib
  DESTINATION share)
```

Здесь цель установки определена при помощи трёх команд `install()` аналогично примеру на с. 182. Основным отличием этого примера является использование команды `target_include_directories()` с аргументами, содержащими рассмотренные ранее выражения генераторов `$<BUILD_INTERFACE:...>` и `$<INSTALL_INTERFACE:...>`. Первое из них при вычислении заменяется на свой аргумент при подключении библиотеки к цели того же проекта, и на пустую строку — при подключении установленной библиотеки из внешнего проекта (при помощи сгенерированного целью установки сценария `share/my_lib.cmake`). Второе выражение действует противоположным образом. *

Замечание: как и в предыдущем примере, аргумент выражения генераторов `$<BUILD_INTERFACE:...>`, которое передаётся команде `target_include_directories()`, должен быть полным путём. Однако путь в аргументе выражения `$<INSTALL_INTERFACE:...>` уже должен указываться относительно каталога установки. Это необходимо для того, чтобы команда цели установки смогла корректно сгенерировать код сценария для подключения библиотеки. Также допустимо указание абсолютных путей с использованием ранее рассмотренного выражения `$<INSTALL_PREFIX>`. ▲

2.12.6. Вспомогательные выражения

Вспомогательные выражения генераторов (табл. 2.17) используются для обхода ограничений синтаксиса самих выражений генераторов: при помощи них можно вставить в вычисляемое выражение символы запятой, закрывающей угловой скобки и т. д.

Таблица 2.17

Вспомогательные выражения генераторов

Выражение	Результат
`ANGLE-R`	«>» («больше»)
`COMMA`	«,» («запятая»)
`SEMICOLON`	«;» («точка с запятой»)

Окончание табл. 2.17

Выражение	Результат
TARGET_NAME:⟨*строка*⟩	Отмечает заданную строку как имя цели *

* Имя цели в выражении может содержать символы запятой и двоеточия, не допустимые в аргументах других выражений генераторов. Однако имя цели здесь должно быть литералом (не вложенным выражением).

2.13. Упражнения

2.13.1. Тест рубежного контроля

1. При исполнении системой CMake следующего фрагмента кода:

```
set(DATA a b c)
message(${DATA})
```

будет выведено:

(a) abc; (b) a b c; (c) a;b;c; (d) a\;b\;c.

2. При исполнении системой CMake следующего фрагмента кода:

```
set(DATA a;b;c)
message(${DATA})
```

будет выведено:

(a) `abc;` (b) `a b c;` (c) `a;b;c;` (d) `a\;b\;c.`

3. При исполнении системой CMake следующего фрагмента кода:

```
set(DATA a\;b\;c)
message(${DATA})
```

будет выведено:

(a) `abc;` (b) `a b c;` (c) `a;b;c;` (d) `a\;b\;c.`

4. При исполнении системой CMake следующего фрагмента кода:

```
set(DATA "a b c")
message(${DATA})
```

будет выведено:

(a) `abc;` (b) `a b c;` (c) `a;b;c;` (d) `a\;b\;c.`

5. При исполнении системой CMake следующего фрагмента кода:

```
set(DATA "a;b;c")
message(${DATA})
```

будет выведено:

(a) `abc;` (b) `a b c;` (c) `a;b;c;` (d) `a\;b\;c.`

6. При исполнении системой CMake следующего фрагмента кода:

```
set(DATA "a\;b\;c")
message(${DATA})
```

   будет выведено:

   (a) `abc`; (b) `a b c`; (c) `a;b;c`; (d) `a\;b\;c`.

7. При исполнении системой CMake следующего фрагмента кода:

```
set(DATA a b c)
message("${DATA}")
```

   будет выведено:

   (a) `abc`; (b) `a b c`; (c) `a;b;c`; (d) `a\;b\;c`.

8. Пусть требуется в файле `CMakeLists.txt` корневого проекта подключить подпроект, находящийся в подкаталоге `library_1`. При помощи какой команды это можно сделать?

   (a) **include**(`library_1/CMakeLists.txt`);

   (b) **add_subdirectory**(`library_1`);

   (c) **include_directories**(`library_1`).

9. Пусть файл `CMakeLists.txt` для библиотеки имеет следующее содержимое:

```
add_library(my_lib file1.cpp file1.h)
```

   Пусть в проекте приложения, подключающем эту библиотеку, требуется настроить путь к заголовочному файлу

библиотеки, а также определить символ препроцессора `MY_LIB_USE`. Какой минимальный набор команд нужно использовать для этого в файле `CMakeLists.txt` проекта приложения?

(a) `target_link_libraries()`;

(b) `target_link_libraries()`, `include_directories()`, `add_definitions()`;

(c) `target_link_libraries()`, `include_directories()`, `add_compile_options()`.

10. Пусть теперь файл `CMakeLists.txt` для библиотеки из предыдущего вопроса был изменён следующим образом:

```
add_library(my_lib file1.cpp file1.h)

target_include_directories(
  my_lib INTERFACE .)

target_compile_definitions(
  my_lib PRIVATE -DMY_LIB_USE)
```

Какой теперь минимальный набор команд нужно использовать в файле `CMakeLists.txt` проекта приложения для подключения библиотеки этой цели?

(a) `target_link_libraries()`;

(b) `target_link_libraries()`, `add_definitions()`;

(c) `target_link_libraries()`, `include_directories()`;

(d) `target_link_libraries()`, `include_directories()`, `add_definitions()`;

(e) `target_link_libraries()`, `include_directories()`, `add_compile_options()`.

11. Пусть в файле `CMakeLists.txt` есть следующие команды для добавления целей библиотек:

```
add_library(lib1 MODULE lib1.cpp)

add_library(lib2 lib2.cpp)

set(BUILD_SHARED_LIBS FALSE)

add_library(lib3 lib3.cpp)
```

Пусть инструмент CMake вызывается для этого проекта при помощи следующей команды:

```
cmake -G ⟨генератор⟩ -D BUILD_SHARED_LIBS=1 ⟨путь⟩
```

Каких типов будут созданы цели библиотек `lib1`, `lib2` и `lib3` соответственно в результате исполнения инструмента CMake для данного проекта?

(a) `STATIC`, `STATIC`, `STATIC`.

(b) `SHARED`, `STATIC`, `STATIC`.

(c) `MODULE`, `STATIC`, `STATIC`.

(d) `MODULE`, `SHARED`, `STATIC`.

(e) `MODULE`, `SHARED`, `SHARED`.

12. *(Вопрос со множественным выбором.)* Пусть во входном файле CMake требуется выполнить некоторое действие, если переменная `CONDITION` содержит значение истины. Как можно корректно записать команду `if()` для проверки этого условия?

    (a) `if(CONDITION EQUAL TRUE)`

    (b) `if(CONDITION STREQUAL TRUE)`

    (c) `if(${CONDITION} STREQUAL TRUE)`

    (d) `if("${CONDITION}" STREQUAL TRUE)`

    (e) `if(CONDITION)`

    (f) `if(${CONDITION})`

    (g) `if("CONDITION")`

    (h) `if("${CONDITION}")`

13. Пусть цикл имеет в качестве заголовка следующую команду:

```
foreach(I RANGE 10)
```

    В этом случае тело цикла будет исполняться:

    (a) 1 раз; (b) 10 раз; (c) 11 раз.

14. При исполнении следующей команды:

```
get_property(
  SOME VARIABLE PROPERTY CMAKE_MODULE_PATH SET)
```

    значение переменной `SOME`:

    (a) не изменится;

    (b) установится в значение «истина»;

(c) установится в значение «ложь»;

(d) установится в текущее значение специальной переменной CMAKE_MODULE_PATH.

15. Исполнение системой CMake команды:

```
add_custom_command(
  TARGET prog
  PRE_BUILD
  COMMAND
    "${CMAKE_COMMAND}" -E echo "$<JOIN:a;b;c, - >")
```

приведёт к тому, что во время перестроения цели prog будет выведено:

(a) ничего;

(b) «$<JOIN:a;b;c, - >»;

(c) «a;b;c, - »;

(d) «a;b;c»;

(e) « - a - b - c»;

(f) «a - b - c - »;

(g) «a - b - c».

2.13.2. Проектное задание

Выполните задание п. 1.4.2 при помощи системы CMake. Требования к реализации:

— Части a и b задания должны быть оформлены в виде отдельных проектов (располагаться в разных каталогах системы).

— Проект библиотеки должен иметь возможность создавать как статический, так и динамический/разделяемый вариант.

— Проект библиотеки должен содержать описание целей тестирования и установки. Установка должна поддерживать компоненты для обычного пользователя и для разработчика. Структура каталога установки должна быть аналогичной структуре, приведённой на рис. 2.13.

— Проект библиотеки должен содержать описание интерфейсных настроек, применяемых к зависимым проектам.

— Проект приложения должен подключать библиотеку, используя её сценарий, сгенерированный во время установки. Для этого проект приложения должен иметь кэш-настройку, определяющую путь к каталогу установки библиотеки.

Проверьте работоспособность собираемой программы для различных доступных операционных систем и компиляторов.

3. Примеры использования пакетов

В этой главе будут приведены примеры использования некоторых популярных программных пакетов в проектах, которые используют систему CMake. Примеры подобраны таким образом, что демонстрируют решение различных задач, связанных с подключением внешних библиотек и инструментов, на разном уровне сложности.

3.1. OpenCV

Набор библиотек OpenCV[1] реализует основные алгоритмы анализа и обработки изображений («компьютерного зрения»), в том числе основанных на машинном обучении: рисование, фильтрация, трансформации, распознавание образов и т. д. Его название расшифровывается как Open Source Computer Vision Library. Библиотеки имеют открытый исходный код и предоставляются в использование по свободной лицензии BSD. Система сборки набора библиотек использует инструмент CMake.

[1] http://opencv.org/ (дата обращения: 09.02.2015).

Подключение библиотек в проектах, использующих для построения CMake, достаточно просто: нужно найти соответствующий пакет при помощи команды `find_package()` (п. 2.11.1) и установить использование библиотек при компоновке всех нужных целей. Минимальный код, добавляющий к цели функциональность OpenCV, будет следующим:

```
# ...

find_package(OpenCV REQUIRED)

# Требуется для OpenCV версии 3
include_directories(${OpenCV_INCLUDE_DIRS})

target_link_libraries(my_prog ${OpenCV_LIBS})
```

После аргумента `REQUIRED` команды `find_package()` можно указать отдельные компоненты, которые требуются для подключения (по умолчанию считается, что нужно подключить все компоненты). Названия компонент совпадают с названиями соответствующих модулей библиотек в справочной системе: `core`, `highgui`, `imgproc` и т. д. Полный список доступных модулей библиотек содержится в переменной `OpenCV_LIB_COMPONENTS` после выполнения команды `find_package()` (набор библиотек может быть собран без каких-то необязательных модулей, например реализующих поддержку технологий CUDA и OpenCL). В результате поиска создаётся переменная `OpenCV_LIBS`, содержащая список целей биб-

лиотек, которые можно передать в target_link_libraries(). Варианты библиотек выбираются в зависимости от конфигурации построения (для генераторов со множественными конфигурациями используются аргументы debug и optimized команды target_link_libraries(), п. 2.6.7) и от значения специальной переменной BUILD_SHARED_LIBS. Список каталогов поиска подключаемых файлов, который необходимо передать команде include_directories() и т. п. (выполняется автоматически для OpenCV версий 2.x.x), записывается в переменную OpenCV_INCLUDE_DIRS. Также после исполнения команды find_package() создаются другие переменные: OpenCV_VERSION (версия набора библиотек), OpenCV_SHARED (собраны ли библиотеки в виде разделяемых модулей) и т. д.

Пример

Рассмотрим тестовое приложение, которое выводит окно с текстовой надписью (рис. 3.1).

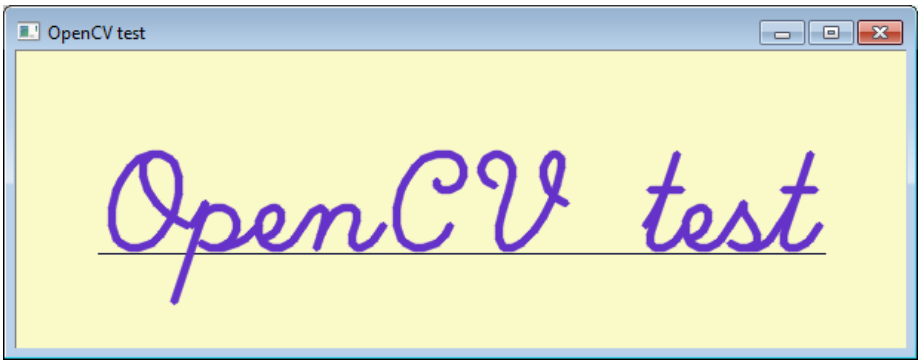

Рис. 3.1. Окно приложения, использующего библиотеки OpenCV

Файл test_opencv.cpp:

```
#include "opencv2/core/core.hpp"
#include "opencv2/highgui/highgui.hpp"
#if CV_MAJOR_VERSION >= 3
  #include "opencv2/imgproc/imgproc.hpp"
#endif

const std::string g_cText = "OpenCV test";
const int g_cnFontFace =
  cv::FONT_HERSHEY_SCRIPT_SIMPLEX;
const double g_cdFontScale = 3;
const int g_cnThickness = 4;

int main()
{
  cv::Mat image(
    200, 600, CV_8UC3, cv::Scalar(200, 250, 250));
  int nBaseLine = 0;
  cv::Size size = cv::getTextSize(
    g_cText, g_cnFontFace, g_cdFontScale,
    g_cnThickness, &nBaseLine);
  cv::Point textOrigin(
    (image.cols - size.width) / 2,
    (image.rows + size.height) / 2);
  cv::line(
    image, textOrigin + cv::Point(0, g_cnThickness),
    textOrigin + cv::Point(size.width, g_cnThickness),
    cv::Scalar(50, 0, 0));
```

```
  cv::putText(
    image, g_cText, textOrigin, g_cnFontFace,
    g_cdFontScale, cv::Scalar(200, 50, 100),
    g_cnThickness, CV_AA);
  cv::namedWindow(
    g_cText.c_str(), CV_WINDOW_AUTOSIZE);
  cv::imshow(g_cText.c_str(), image);
  cv::waitKey(0);
}
```

В реализации приложения использованы компоненты core для хранения изображения (класс cv::Mat) и рисования примитивов (линии, вывод текста, измерение размеров надписи) и highgui для использования простого оконного интерфейса. В OpenCV 3 функции рисования перенесены в библиотеку imgproc. Таким образом, описание проекта приложения может быть следующим (файл CMakeLists.txt):

```
cmake_minimum_required(VERSION 2.8)

project(test_opencv)

find_package(OpenCV REQUIRED core highgui imgproc)

add_executable(test_opencv WIN32 test_opencv.cpp)
include_directories(${OpenCV_INCLUDE_DIRS})
target_link_libraries(test_opencv ${OpenCV_LIBS})
```

```
if(MSVC)
  target_link_libraries(
    test_opencv -ENTRY:mainCRTStartup)
  include(correct_vc_static.cmake)
endif()
```

Здесь аргумент WIN32, который передаётся команде add_executable() (п. 2.5.1), используется для того, чтобы в системе Windows во время работы приложения не отображалась его основная консоль (она создаётся автоматически, если точкой входа в программу является функция main()). При использовании компоновщика Microsoft Visual C++ в процессе построения ему передаётся ключ «/SUBSYSTEM:WINDOWS». Кроме этого, ему также необходим ключ «/ENTRY:mainCRTStartup». Точка входа mainCRTStartup() из библиотеки поддержки выполнения программ компилятора Microsoft Visual C++ запускает функцию пользователя main(). Этот ключ передаётся во время компоновки в результате исполнения второй команды target_link_libraries() (п. 2.6.7). Знак «минус» вместо «/» в начале ключа необходим, так как указывает команде, что этот аргумент является ключом компоновщика, а не именем библиотеки. В процессе построения проекта он будет заменён на символ, соответствующий формату ключей используемого компоновщика.

Оставшаяся часть настроек проекта вынесена в отдельный модуль correct_vc_static.cmake:

```
if(MSVC AND NOT BUILD_SHARED_LIBS)
```

```
  set(VAR_NAMES)
  foreach(LANG C CXX)
    list(APPEND NAMES CMAKE_${LANG}_FLAGS)
    foreach(CFG IN LISTS CMAKE_CONFIGURATION_TYPES)
      string(TOUPPER ${CFG} CFG)
      list(APPEND VAR_NAMES CMAKE_${LANG}_FLAGS_${CFG})
    endforeach()
  endforeach()

  #
  foreach(VAR_NAME IN LISTS VAR_NAMES)
    string(REPLACE /MD /MT ${VAR_NAME} ${${VAR_NAME}})
  endforeach()
endif()
```

Причина, по которой эти команды вынесены в отдельный модуль, заключается в том, что аналогичные настройки будут необходимы нам при реализации других проектов. Модуль обеспечивает совместимость объектных файлов проекта со статическими версиями библиотек OpenCV. Компилятор Microsoft Visual C++ поддерживает создание файлов, которые зависят либо от динамических библиотек поддержки исполнения программ (например, `msvcr110d.dll` и т. д.), либо от их статических версий. Каждый из вариантов также имеет версии для отладки и поставки конечному пользователю (Debug/Release). При компоновке нескольких объектных модулей и библиотек необходимо, чтобы все они зависели от библиотек поддержки исполнения про-

грамм одного и того же типа. Тип используемой библиотеки определяется ключом компилятора (табл. 3.1).

Таблица 3.1

Влияние ключей компилятора Microsoft Visual C++ на тип используемой библиотеки поддержки исполнения программ

Ключ	Динамическая	Отладочная
`/MDd`	✔	✔
`/MD`	✔	✗
`/MTd`	✗	✔
`/MT`	✗	✗

Статические версии библиотек OpenCV скомпилированы с ключами «`/MTd`» и «`/MT`», в то время как генератор CMake для Visual Studio по умолчанию создаёт правила компиляции с ключами «`/MDd`» и «`/MD`». В приведённом выше примере сначала в переменной `VAR_NAMES` формируется список имён специальных переменных `CMAKE_C_FLAGS`, `CMAKE_C_FLAGS_DEBUG` и т. д., в которых содержатся ключи компиляции для соответствующего языка и конфигурации (или общие для всех конфигураций). Для этого используется специальная переменная `CMAKE_CONFIGURATION_TYPES`, в которой перечислены имена всех генерируемых конфигураций. Дальше в каждой из этих переменных происходит замена подстроки «`/MD`» на «`/MT`» при помощи команды `string(REPLACE ...)` (п. 2.7.4). *

Последнее, что необходимо обсудить, — это то, каким образом нужно запускать инструмент CMake, чтобы команда `find_packege()` смогла найти библиотеки OpenCV. В системах, основанных на Linux, если библиотеки установлены в стандартные каталоги (например, из системных репозитариев), никаких дополнительных настроек не требуется, команда вызова CMake может выглядеть подобным образом:

```
cmake ~/work/ex-opencv
```

Если библиотеки были собраны вручную и установлены в отдельный каталог, в таком случае можно указать системе CMake путь к каталогу их установки, например перечислив его среди прочих в переменной `CMAKE_PREFIX_PATH`:

```
cmake \
  -D CMAKE_PREFIX_PATH=$HOME/install/opencv-2.4.10 \
  ~/work/ex-opencv
```

При этом команда `find_package()` корректно определит путь к конфигурационному файлу `OpenCVConfig.cmake`, который находится в одном из подкаталогов каталога установки, в соответствии с процедурой, описанной в п. 2.11.1.

В системе Windows библиотеки разворачиваются из самораспаковывающегося архива в требуемый каталог. Внутри него находятся подкаталоги `build` и `sources`. В подкаталоге `build` есть конфигурационный файл `OpenCVConfig.cmake` и несколько вариантов сборки библиотек: для архитектур x86 и x86-64, среды Visual Studio версий 2010 (только в OpenCV версий 2.x.x),

2012 и 2013, и статического/динамического вариантов библиотек. Инструменту CMake необходим путь к каталогу `build`:

```
cmake^
  -G "Visual Studio 12"^
  -D CMAKE_PREFIX_PATH=d:\Tools\opencv\build^
  D:\Work\ex-opencv
```

Конфигурационный файл самостоятельно определит подкаталог с требуемым вариантом сборки библиотек в зависимости от используемого генератора и значения переменной `BUILD_SHARED_LIBS`, например:

```
cmake^
  -G "Visual Studio 12"^
  -D CMAKE_PREFIX_PATH=d:\Tools\opencv\build^
  -D BUILD_SHARED_LIBS=0^
  D:\Work\ex-opencv
```

Замечание: при использовании динамических вариантов библиотек для запуска собранных приложений необходимо, чтобы система могла найти эти библиотеки. Для этого к системной переменной окружения `PATH` можно добавить путь к библиотекам, например:

```
set PATH=d:\Tools\opencv\build\x86\vc12\bin;%PATH%
```

Другим решением будет копирование требуемых библиотек (например, `opencv_core2411d.dll` и `opencv_highgui2411d.dll`) в каталог исполняемого файла. ▲

Если требуется использовать другой компилятор, например gcc-MinGW, библиотеки необходимо предварительно собрать с его помощью. Для этого можно создать новый каталог рядом с каталогом build набора библиотек и впоследствии передавать его инструменту CMake через переменную CMAKE_PREFIX_PATH:

```
cd /d d:\Tools\opencv
mkdir build_mingw
cd build_mingw
cmake -G "MinGW Makefiles" -D WITH_IPP=OFF ..\sources
mingw32-make
```

Здесь установка переменной WITH_IPP в значение «ложь» в командной строке вызова CMake необходима для корректной сборки библиотек OpenCV 3 компилятором gcc-MinGW[2]. Также в этом случае необходимо найти и закомментировать строку:

```
  add_extra_compiler_option(-Werror=non-virtual-dtor)
```

в файле sources\cmake\OpenCVCompilerOptions.cmake[3].

Для сборки приложений необходимо использовать ту же версию компилятора, которая использовалась для сборки библиотек.

[2] http://answers.opencv.org/question/40159/cannot-compile-opencv-30/ (дата обращения: 29.04.2015).

[3] http://code.opencv.org/issues/4107#note-3 (дата обращения: 30.04.2015).

3.2. Boost

3.2.1. Интерфейс подключения библиотек

Boost[4] представляет собой большой набор библиотек, разрабатываемых широким сообществом пользователей языка C++. Библиотеки распространяются по свободной лицензии Boost Software License, которая допускает в том числе их коммерческое использование. В набор входят универсальные библиотеки, для которых предполагается возможность использования во многих приложениях вне зависимости от их области применения. В настоящее время Boost объединяет более 110 библиотек по следующим основным категориям:

— Алгоритмы обработки текста: регулярные выражения, лексический анализ и т. д.

— Контейнеры: графы, многомерные массивы, двунаправленные отображения и т. д.

— Метапрограммирование в шаблонах и макросах.

— Параллельное программирование: потоки, межпроцессное и сетевое взаимодействие, высокоуровневая обёртка над библиотекой MPI и т. д.

— Математические алгоритмы: эффективная реализация объектно-ориентированной библиотеки операций линейной алгебры, геометрия, целые числа, интервальная арифметика и т. д.

[4] http://www.boost.org/ (дата обращения: 17.02.2015).

— Управление ресурсами, интеллектуальные указатели.

— Системные механизмы: работа с датой и временем, кросс-платформенный уровень абстракции для файловой системы.

— Разбор аргументов командной строки.

— И т. д.

Основной упор в Boost делается на совместимость со стандартной библиотекой. Например, стандартные алгоритмы можно использовать для обработки данных контейнеров Boost и наоборот. Boost можно считать в некотором роде экспериментальной площадкой для языка C++, поскольку многие из библиотек, включённых комитетом по его стандартизации[5] в новые версии стандарта, перед этим были реализованы в Boost.

Boost использует собственную систему построения. Также существует экспериментальная реализация сборки библиотек при помощи системы CMake. В состав CMake включён модуль `FindBoost.cmake`, который используется для поиска библиотек при помощи команды `find_package()` (п. 2.11.1). В процессе поиска сначала происходит попытка обнаружения библиотек при помощи конфигурационного файла пакета (для случая, когда библиотеки собраны системой CMake). Если конфигурационный файл не обнаружен, модуль пытается найти библиотеки самостоятельно. Кроме переменной `CMAKE_PREFIX_PATH`, для поиска используются пути, записанные в следующих переменных окружения и CMake (табл. 3.2).

[5] http://www.open-std.org/jtc1/sc22/wg21/ (дата обращения: 17.02.2015).

Таблица 3.2

Переменные (CMake и окружения), в которых могут содержаться пути к библиотекам Boost, используемые модулем `FindBoost`

Переменная	Путь
`BOOST_ROOT`	Каталог установки (⟨*корень*⟩)
`BOOSTROOT`	То же самое
`BOOST_INCLUDEDIR`	Каталог заголовочных файлов (⟨*корень*⟩`/include`)
`BOOST_LIBRARYDIR`	Каталог библиотек (⟨*корень*⟩`/lib`)

Модуль `FindBoost.cmake` способен найти библиотеки Boost, если их версия находится в списке версий, известных модулю (от 1.33.0 до 1.58.0). Если версия установленных библиотек не входит в этот диапазон, дополнительные номера версий можно присвоить переменной CMake `Boost_ADDITIONAL_VERSIONS`.

Особенностью Boost по сравнению с другими библиотеками является то, что большинство его библиотек состоит только из заголовочных файлов (header-only libraries). Что касается оставшихся библиотек, то для части из них необходима сборка, чтобы их можно было использовать в сторонних проектах. Для другой части возможна необязательная сборка библиотечных модулей с целью использования дополнительных возможностей.

Для каждого из библиотечных модулей возможны варианты сборки, которые имеют следующие отличия:

— версия для отладки или для сборки приложений, предназначенных для конечного пользователя;

— код, скомпонованный с отладочными версиями стандартных библиотек C/C++ и библиотек поддержки выполнения программ или с их версиями для конечного пользователя;

— статическая или разделяемая библиотека;

— код, скомпонованный со статическими версиями стандартных библиотек или с их динамическими версиями (см. пример на с. 258);

— версия кода, обеспечивающая потокобезопасность в многопоточных приложениях или не обеспечивающая.

Также могут выбираться некоторые редко используемые свойства (сборка с библиотекой STLPort вместо стандартной, сборка с отладочными версиями библиотек Python). Варианты собранных библиотек для всех (или некоторых) комбинаций перечисленных свойств помещаются в один каталог при установке Boost. Возможна также установка в один и тот же каталог библиотек разных версий Boost, собранных разными компиляторами и т. д. Задача модуля CMake состоит в выборе правильных версий библиотек и заголовочных файлов в соответствии с настройками проекта пользователя. Следующие переменные (табл. 3.3) влияют на этот выбор.

Таблица 3.3

Переменные, влияющие на вариант выбираемых библиотек Boost

Переменная	Описание	Умолчание
Логические		
`Boost_USE_MULTITHREADED`	Многопоточные	Да
`Boost_USE_STATIC_LIBS`	Статические	Да
`Boost_USE_STATIC_RUNTIME`	Со статическими стандартными библиотеками	В зависимости от платформы
`Boost_USE_DEBUG_RUNTIME`	С отладочными стандартными библиотеками	Да
Строковые		
`Boost_COMPILER`	Суффикс компилятора (`-vc120`, `-gcc49` и т. д.)	В зависимости от компилятора

Замечание: возможны разные схемы именования библиотек Boost. Как правило, установленные в системах на основе ядра Linux версии из репозитариев не имеют в именах суффиксов, обозначающих вариант библиотеки. В этом случае библиотеки доступны в единственном варианте и настройки из табл. 3.3 не оказывают никакого влияния. ▲

После выполнения модуль `FindBoost.cmake` устанавливает в следующих основных переменных (табл. 3.4) результаты своей работы.

Таблица 3.4

Переменные с результатами поиска библиотек Boost

Переменная	Описание
`Boost_FOUND`	Истина, если заданные библиотеки найдены
`Boost_INCLUDE_DIRS`	Каталоги заголовочных файлов
`Boost_LIBRARIES`	Библиотеки для подключения к цели
`Boost_`⟨*компонент*⟩`_FOUND`	Истина, если найден заданный ⟨*компонент*⟩ *
`Boost_`⟨*компонент*⟩`_LIBRARY`	Библиотеки для подключения заданного компонента *
`Boost_VERSION`	Версия найденных библиотек
`Boost_LIB_VERSION`	Суффикс версии в файлах библиотек

* ⟨*компонент*⟩ пишется в верхнем регистре.

3.2.2. Подключение заголовочной библиотеки

Подключение библиотеки Boost, состоящей только из заголовочных файлов, выполняется проще всего, так как в этом случае нет необходимости заботиться о выборе нужного варианта

файлов библиотеки. Заголовочные файлы могут только различаться версией Boost, располагаясь в разных подкаталогах. Общая схема подключения выглядит следующим образом:

```
# ...

find_package(Boost ⟨версия⟩ REQUIRED)

include_directories(${Boost_INCLUDE_DIRS})

add_executable(⟨цель$_1$⟩ ⟨файл$_{1,1}$⟩ ... ⟨файл$_{n,1}$⟩)
# ...
```

Замечание: для выполнения поиска библиотек Boost команде `find_package()` (п. 2.11.1) передаются только имена тех компонент, которые имеют файлы библиотек, необходимые для подключения к целям, а не те, которые состоят только из заголовочных файлов. Поскольку в рассматриваемом случае таких библиотек нет, список компонент после аргумента `REQUIRED` тоже пуст. ▲

Пример

Пусть требуется скомпилировать следующую программу (файл `ex-mpl.cpp`), которая использует заголовочные файлы библиотек MPL и TypeTraits:

```
#include <boost/mpl/placeholders.hpp>
#include <boost/mpl/vector.hpp>
#include <boost/mpl/for_each.hpp>
```

```
#include <boost/mpl/filter_view.hpp>
#include <boost/mpl/transform_view.hpp>

#include <boost/type_traits.hpp>

#include <iostream>
#include <typeinfo>

namespace mpl = boost::mpl;
using namespace mpl::placeholders;

template <typename Types>
  struct get_pointees :
    mpl::transform_view
    <
      mpl::filter_view
      <
        Types,
        boost::is_pointer <_>
      >,
      boost::remove_pointer <_>
    >
{ };

template <typename T>
  struct wrap
{ };
```

```
struct print_type
{
  template <class T>
    void operator () (wrap <T>) const
  {
    std::cout << typeid (T).name() << " | ";
  }
};

template <class TSeq>
  void print_types()
{
  mpl::for_each <TSeq, wrap <_> > (print_type());
  std::cout << std::endl;
}

int main()
{
  typedef
    mpl::vector
    <
      char, short *,
      volatile int **,
      long,
      long long [10]
    >
```

```
    types_ptrs;
  //
  print_types <types_ptrs> ();
  print_types <get_pointees <types_ptrs>::type> ();
}
```

В этой программе на печать выводится сначала список названий типов, содержащихся в контейнере `types_ptrs` (предполагается, что в нём нет ссылочных типов), а затем названия типов, для которых в исходном списке присутствуют типы-указатели.

Для генерирования проекта можно использовать файл `CMakeLists.txt` со следующим содержимым:

```
cmake_minimum_required(VERSION 2.8)

project(ex-mpl)

find_package(Boost 1.32.0 REQUIRED)

include_directories(${Boost_INCLUDE_DIRS})
add_executable(ex-mpl ex-mpl.cpp)
```

Здесь указана версия Boost 1.32, так как это последняя версия, в которой были внесены изменения в библиотеку MPL, а также после неё не происходило изменений в заголовочных файлах библиотеки TypeTraits, которые были бы существенны для этой программы. Установка минимально возможной вер-

сии Boost позволяет собрать проект на системах с давно не обновлявшимся ПО. *

Замечание: если требуется поддержка статических вариантов сборки проекта при помощи компилятора Microsoft Visual C++, можно использовать модуль `correct_vc_static.cmake` из примера на с. 258. ▲

Команда вызова инструмента CMake для проекта, использующего только заголовочные библиотеки Boost, может быть следующей:

```
cmake^
  -D CMAKE_PREFIX_PATH=⟨путь_к_Boost⟩^
  -D Boost_ADDITIONAL_VERSIONS=1.59.0;1.59^
  ⟨другие_настройки⟩^
  ⟨каталог_проекта⟩
```

Как обычно, в переменной `CMAKE_PREFIX_PATH` могут быть перечислены и иные пути для поиска других пакетов (через точку с запятой). Путь может указывать на тот каталог, в который был установлен Boost во время сборки при помощи ключа «`--prefix=...`» его системы построения. Однако установка выполняется только после сборки всех библиотек Boost, а это может занять много времени. Если требуется использовать только «заголовочные» библиотеки, можно разархивировать исходные коды Boost в отдельный каталог и передать CMake путь к нему. Модуль поиска `FindBoost.cmake` корректно определит в этом случае значение переменной `Boost_INCLUDE_DIRS`.

3.2.3. Подключение библиотеки с модулем компоновки

Подключение скомпилированных файлов библиотек Boost кроме приведённых ранее команд дополнительно требует использования команды `target_link_libraries()`:

```
# ...

find_package(
  Boost ⟨версия⟩ REQUIRED ⟨компонент_1⟩ ... ⟨компонент_m⟩)

include_directories(${Boost_INCLUDE_DIRS})

add_executable(⟨цель_1⟩ ⟨файл_{1,1}⟩ ... ⟨файл_{n,1}⟩)
target_link_libraries(⟨цель_1⟩ ${Boost_LIBRARIES})
# ...
```

В переменную `Boost_LIBRARIES` модуль поиска Boost запишет пути к файлам библиотечных модулей, при необходимости вместе с аргументами `debug` и `optimized` (см. п. 2.6.7).

При подключении библиотеки, имеющей в своём составе скомпилированный модуль, следует учитывать, что необходимо правильно выбрать нужный вариант его сборки в зависимости от конфигурации построения проекта и других условий. За выбор отвечают переменные модуля `FindBoost.cmake` из табл. 3.3.

Пример

Пусть требуется собрать следующую программу, которая использует библиотеку Program Options (файл `ex-boost.cpp`):

```
#include <boost/program_options.hpp>

#include <iostream>

using namespace boost::program_options;

void usage(const options_description &rcDesc)
{
  std::cout <<
    "\nUsage:\n  ex-boost [<options>]\n\n" <<
    rcDesc << std::endl;
}

int main(int nArgC, char *apszArgV[])
{
  options_description desc("Allowed options");
  variables_map vars;
  try
  {
    desc.add_options()
      ("help,h", "print help message")
      ("version,v", "print version string")
      ("number,n", value <int> (), "set some number")
      ;
```

```
    store(
      parse_command_line(
        nArgC, apszArgV, desc), vars);
    notify(vars);
  }
  catch (const error &rcError)
  {
    std::cout <<
      std::endl << "Command line error: " <<
      rcError.what() << std::endl;
    usage(desc);
    return 1;
  }    // catch (const error &)
  //
  if (vars.count("help"))
    usage(desc);
  //
  for (auto pair : vars)
  {
    std::cout << pair.first;
    if (int *pn = boost::any_cast <int> (
      &pair.second.value()))
      std::cout << ": " << *pn;
    std::cout << std::endl;
  }    //  for (auto pair : vars)
}    // main()
```

Эта программа ожидает один или несколько аргументов командной строки: «-h», «-v», «-n ⟨*число*⟩» (или их длинные версии). Если командная строка не соответствует этому формату, выводится сообщение о правильном использовании команды, и программа завершается. Иначе, если указан аргумент «-h», выводится справка об аргументах. Также выводятся названия всех указанных в командной строке аргументов, и, если какой-то из них задаёт целочисленный аргумент, выводится его значение.

Пусть требуется возможность построения проекта в конфигурациях «Debug», «Release» и т. д. Пусть также необходима возможность выбора сборки со статическими или разделяемыми библиотеками. Что касается многопоточности, анализ программы показывает, что библиотека Program Options не используется в нескольких потоках, поэтому её потокобезопасность не требуется.

Таким образом, Файл `CMakeLists.txt` для проекта может быть следующим:

```
cmake_minimum_required(VERSION 3.2)

project(ex-boost)

if(BUILD_SHARED_LIBS)
  add_definitions(-DBOOST_ALL_DYN_LINK)
else()
  set(Boost_USE_STATIC_LIBS ON)
  set(Boost_USE_MULTITHREADED OFF)
```

```
  if(MSVC)
    set(Boost_USE_STATIC_RUNTIME ON)
  endif()
endif()

find_package(Boost 1.40 REQUIRED program_options)

include_directories(${Boost_INCLUDE_DIRS})
add_definitions(-DBOOST_ALL_NO_LIB)
add_executable(ex-boost ex-boost.cpp)
target_link_libraries(
  ex-boost ${Boost_LIBRARIES})

target_compile_features(
  ex-boost
  PRIVATE
    cxx_auto_type
    cxx_range_for)

include(correct_vc_static.cmake)
```

Модуль `correct_vc_static.cmake`, подключаемый последней командой, имеет такое же содержимое, что и одноимённый модуль из примера подключения библиотек OpenCV на с. 258.

Здесь перед вызовом команды `find_package()` (п. 2.11.1) устанавливаются значения переменных из табл. 3.3. Переменной `Boost_USE_STATIC_LIBS` присваивается «истина», ес-

ли специальная переменная `BUILD_SHARED_LIBS` установлена в истину. При использовании компилятора Microsoft Visual C++ в этом случае также устанавливается в истину переменная `Boost_USE_STATIC_RUNTIME`, чтобы объектный модуль приложения и библиотечный модуль Boost использовали один и тот же тип стандартных библиотек времени исполнения (иначе будут ошибки при компоновке). Дополнительно модуль `correct_vc_static.cmake` в этом случае устанавливает ключи компилятора, обеспечивающие зависимость компилируемого объектного модуля от статических библиотек. Значение переменной `Boost_USE_MULTITHREADED` устанавливается в «ложь» только в статическом варианте сборки, поскольку библиотеки Boost не собираются в разделяемом однопоточном варианте. Значение переменной `Boost_USE_DEBUG_RUNTIME` подходит по умолчанию при любом варианте сборки.

По умолчанию в заголовочных файлах Boost включены зависящие от компилятора директивы подключения библиотек при компоновке («`#pragma comment(lib, ⟨`*имя_библиотеки*`⟩)`»). Эти директивы доступны не для всех компиляторов. Чтобы они не вступали в конфликт с правилами, определяемыми CMake, они отключаются установкой символа препроцессора `BOOST_ALL_NO_LIB` при помощи команды `add_definitions()`. Однако в случае использования разделяемых библиотек Boost это имеет побочный эффект: из объявлений функций и классов исчезает объявление `__declspec (dllimport)` (см. пример на с. 110). Чтобы избежать этого, при использовании разделяе-

мых библиотек также устанавливается символ препроцессора BOOST_ALL_DYN_LINK. *

3.2.4. Частичная подмена стандартной библиотеки

В процессе развития программного проекта может возникнуть задача по его портированию на некоторую устаревшую архитектуру, для которой не существует современного компилятора языка C++. Это могут быть старые сборки систем на основе ядра Linux (например, на вычислительных кластерах), версии Windows, поддержка которых прекращена со стороны компиляторов и т. д. Одной из задач портирования является замена используемых возможностей стандартной библиотеки, которых нет в составе старых компиляторов. Чтобы избежать их реализации вручную, можно использовать тот факт, что большинство их сначала появилось в наборе библиотек Boost, прежде чем быть утверждёнными в новом стандарте (новые контейнеры, интеллектуальные указатели, средства многопоточности и т. д.). Часто интерфейс этих средств мало различается между стандартной библиотекой и Boost: отличия могут быть только в путях к заголовочным файлам и в названиях пространств имён.

Если стандартная библиотека компилятора включает встроенную поддержку требуемых возможностей, лучше использовать её. Если же необходимых средств в ней нет, тогда на систему, где выполняется сборка, можно установить Boost. Он занимает не так много места, и большинство его библиотек имеет хорошую совместимость с большим набором компи-

ляторов, в том числе устаревших. Таким образом, задача заключается в определении того, какие стандартные заголовочные файлы из числа требуемых присутствуют в компиляторе, и при необходимости использовать вместо них заголовочные файлы и библиотечные модули Boost. Эта задача аналогична одной из тех, которые решаются сценарием `configure`, генерируемым инструментами Autotools (п. 1.3.2). Чтобы проверить, имеется ли в наличии та или иная возможность в компиляторе, сценарий генерирует тексты коротких программ и пытается их скомпилировать.

В CMake подобные задачи решаются при помощи стандартных модулей, поставляемых вместе с инструментом. Целая серия модулей, имена которых начинаются с «`Check`», предназначена для проверки наличия тех или иных возможностей экспериментальным путём. Один из таких модулей под названием `CheckIncludeFileCXX` может быть использован для проверки возможности подключения заголовочного файла с заданным путём. В модуле определена команда:

`check_include_file_cxx(`
⟨*имя_файла*⟩ ⟨*имя_переменной*⟩ [⟨*флаги_компилятора*⟩])

Она запускает на компиляцию генерируемый файл C++ с директивой «`#include <`⟨*имя_файла*⟩`>`». Компиляция выполняется при помощи временного файла `CMakeLists.txt`, в котором цель определяется командой `add_executable()`. Компиляция выполняется тем набором инструментов, который соответствует выбранному генератору. Результат (истина/ложь)

записывается в переменную с заданным именем, которая создаётся в кэше (для ускорения последующих запусков CMake, так как при этом каждый раз не нужно выполнять проверку компилируемости кода). Вывод компилятора на тестовом запуске записывается в файл `CMakeOutput.log`, находящийся в подкаталоге с именем, которое хранится в специальной переменной `CMAKE_FILES_DIRECTORY` (как правило, содержит строку «`/CMakeFiles`»). Этот подкаталог находится внутри каталога построения проекта и предназначен для файлов, генерируемых CMake.

Модуль `CheckCXXSourceCompiles` решает более общую задачу: он проверяет компилируемость заданного исходного кода текущим набором инструментов. В нём определена команда:

```
check_cxx_source_compiles(
  ⟨код⟩ ⟨имя_переменной⟩
  [FAIL_REGEX ⟨регулярное_выражение⟩])
```

Необязательное регулярное выражение может задавать соответствие вывода компилятора, которое будет означать неудачу.

Процессом компиляции, запускаемым этими командами, можно управлять при помощи переменных (табл. 3.5).

Таблица 3.5

Переменные, влияющие на компиляцию командами `check_...`

Переменная	Значение
`CMAKE_REQUIRED_FLAGS`	Дополнительные флаги компилятора *
`CMAKE_REQUIRED_DEFINITIONS`	Дополнительные определения символов препроцессора (в формате `-D`⟨*имя*⟩[`=`⟨*значение*⟩])
`CMAKE_REQUIRED_INCLUDES`	Дополнительные пути поиска заголовочных файлов
`CMAKE_REQUIRED_LIBRARIES`	Библиотеки для компоновки **

* Также могут передаваться в качестве третьего аргумента команды `check_include_file_cxx()`.

** Используется только `check_cxx_source_compiles()`.

Пример

Пусть требуется скомпилировать программу, которая использует разделяемые интеллектуальные указатели (файл `ex-shared.cpp`):

```
#include "compatibility.h"

#include <iostream>
```

```
struct Test
{
  int m_n;
  Test() : m_n(99) { }
};

int main()
{
  my::shared_ptr <Test> ptr(new Test);
  std::cout << ptr->m_n << std::endl;
}
```

В более новых компиляторах шаблон `shared_ptr <>` определён в пространстве имён `std` в заголовочном файле `<memory>` (если включена совместимость с C++11 или выше). В тех версиях компиляторов, которые были выпущены до утверждения стандарта C++11, предварительные версии новых возможностей библиотек определены в файлах `<tr1/...>` в пространстве `std::tr1` (по названию «Technical Report 1», неформально обозначающему расширения стандартной библиотеки, принятые в черновом варианте после выхода стандарта C++03). В ещё более старых версиях этот тип указателей отсутствует в стандартных заголовках.

Заголовочный файл `compatibility.h` определяет пространство имён `my`:

```
#ifdef HAS_STD_SHARED
```

```
  #include <memory>
  namespace my = std;
#elif defined (HAS_TR1_SHARED)
  #include <tr1/memory>
  namespace my = std::tr1;
#else
  #include <boost/shared_ptr.hpp>
  namespace my = boost;
#endif
```

На описание проекта возлагается задача поиска нужных заголовочных файлов и определения символов препроцессора HAS_STD_SHARED и HAS_TR1_SHARED в нужных ситуациях (файл CMakeLists.txt):

```
cmake_minimum_required(VERSION 2.4)

project(ex-shared)

set(CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS TRUE)

include(CheckCXXSourceCompiles)
if(CMAKE_COMPILER_IS_GNUCXX)
  set(CMAKE_REQUIRED_FLAGS -std=c++11)
endif()
check_cxx_source_compiles(
  "#include <memory>
   int main()
```

```
    {
      std::shared_ptr <int> p;
    }"
  STD_SHARED_EXISTS)

if(STD_SHARED_EXISTS)
  message(STATUS "Found std::shared_ptr <>")
  add_definitions(-DHAS_STD_SHARED)
  if(CMAKE_COMPILER_IS_GNUCXX)
    add_definitions(-std=c++11)
  endif()
else()
  set(CMAKE_REQUIRED_FLAGS "")
  include(CheckIncludeFileCXX)
  check_include_file_cxx(tr1/memory TR1_SHARED_EXISTS)
  if(TR1_SHARED_EXISTS)
    message(STATUS "Found std::tr1::shared_ptr <>")
    add_definitions(-DHAS_TR1_SHARED)
  else()
    find_package(Boost 1.30)
    if(Boost_FOUND)
      message(STATUS "Found boost::shared_ptr <>")
      include_directories(${Boost_INCLUDE_DIRS})
    else()
      message(
        FATAL_ERROR
        "Neither std::shared_ptr, std::tr1::shared_ptr,"
```

```
        " nor Boost library was found. Please install "
        "Boost.")
    endif()
  endif()
endif()

add_executable(ex-shared ex-shared.cpp compatibility.h)
```

Здесь значение истины присваивается специальной переменной CMAKE_ALLOW_LOOSE_LOOP_CONSTRUCTS для того, чтобы можно было не повторять условия в аргументах команд (elseif() и т. д., старое требование CMake). Начиная с CMake версии 2.6 такое присваивание излишне, однако в этом описании запрашивается совместимость с версией 2.4. Также по причине совместимости используется команда add_definitions() вместо add_compile_options() для передачи флага «-std=c++11». Перед запуском первой тестовой компиляции в случае компилятора gcc ему передаётся этот флаг (включение совместимости со стандартом C++11) через переменную CMAKE_REQUIRED_FLAGS. Перед второй проверкой значение переменной очищается, так как старые компиляторы gcc не понимают флагов «-std=...» и реагируют на них ошибкой.

Тестирование приведённого кода показывает, что он выбирает пространство имён std в операционной системе Windows 7 с компиляторами Microsoft Visual C++ 2012 и TDM-GCC 4.9.2, а также в Windows XP с компилятором Microsoft Visual C++ 2010. В системе Slackware 12.0 с компилятором gcc 4.1.2 он использует про-

странство `std::tr1`. И наконец, в системе Windows XP с компиляторами Microsoft Visual C++ 6.0, 2008 и gcc-MinGW 3.4.5 и библиотекой Boost 1.35.0 он использует пространство имён `boost`.

*

Замечания:

— В более сложных ситуациях вместо использования условных директив препроцессора в файле `compatibility.h` может оказаться удобнее генерировать файл в каталоге построения при помощи команды `configure_file()` (п. 2.10.1).

— Другим способом проверки соответствия компилятора стандарту C++11 является проверка значения макроса `__cplusplus`. Следует, однако, учитывать, что старые компиляторы могут его не поддерживать либо устанавливать его значениями, отличными от стандартного. ▲

3.3. Qt

3.3.1. Интерфейс подключения библиотек

Qt[6] содержит большой набор библиотек и инструментов для реализации каркасов приложений с поддержкой графического интерфейса пользователя, а также других возможностей, востребованных в крупных проектах (многопоточность, сетевые взаимодействия, интерфейс с различными СУБД и т. д.). Основной задачей Qt является обеспечение переносимости кода,

[6] http://www.qt.io/ (дата обращения: 25.02.2015).

написанного с его применением, на большое количество настольных, встроенных и мобильных платформ. При этом быстродействие и размеры программного кода остаются на приемлемом уровне по сравнению с эквивалентным ему кодом, реализованным на местных API (при существенном выигрыше в скорости разработки и переносимости). Qt доступен в нескольких вариантах лицензий — от свободных (GNU GPL v2, LGPL v2.1 и 3) до коммерческих с дополнительными возможностями.

Современные версии Qt (5.x.x) распространяются вместе с конфигурационными файлами CMake. Кроме этого, в CMake есть стандартные модули для подключения Qt версий 3 и 4. В настоящем учебнике будут рассмотрены вопросы, связанные с использованием Qt 5[7].

На сайте Qt доступны предварительно собранные библиотеки в виде разделяемых модулей для различных платформ и компиляторов. Статические варианты можно собрать самостоятельно из исходных кодов. При одновременном использовании для разработки статических и динамических библиотек их следует устанавливать в разные каталоги. Таким образом, выбор варианта используемых библиотек Qt можно определять, например, значением переменной `CMAKE_PREFIX_PATH`, записывая в неё нужный каталог. Кроме этого, может понадобиться настройка проекта для использования статических библиотек аналогично примеру с подключением OpenCV на с. 258. В дальнейшем будем рассматривать использование динамических (разделяемых) библиотек.

[7] В [14] отдельная глава посвящена использованию CMake совместно с Qt 4.

Использовать нужно тот же самый компилятор, которым были собраны библиотеки. При установке версии для gcc-MinGW можно дополнительно установить сам компилятор в подкаталог Qt.

Замечание: при одновременно установленных в системе нескольких вариантах сборки Qt необходимо следить за тем, чтобы инструмент CMake при исполнении команды `find_package()` находил ту версию, которая соответствует выбранному генератору. В противном случае можно получить ошибки компоновки, из которых будет трудно понять причину возникновения проблемы. ▲

Способ подключения Qt зависит от версии CMake. Минимальной поддерживаемой версией является 2.8.3. При использовании 2.8.11 или выше интерфейс подключения выглядит проще всего — за счёт доступных в ней аргументов `INTERFACE`/`PUBLIC` в командах `target_include_directories()` (п. 2.6.2) и т. д. Общая схема в таком случае будет следующей:

```
# ...

find_package(
  Qt5 [⟨версия⟩] REQUIRED ⟨компонент_1⟩ ... ⟨компонент_m⟩)
# или если подключается только один компонент:
# find_package(Qt5⟨компонент⟩ [⟨версия⟩] REQUIRED)

add_executable(⟨цель⟩ WIN32 ⟨файл_1⟩ ... ⟨файл_n⟩)
target_link_libraries(⟨цель⟩ Qt5::⟨компонент_k⟩ ...)
```

```
# ...
```

При необходимости в команде поиска указывается минимальная версия, с которой должен связываться проект.

Замечания:

— Передача имени компонента `Qt5` команде `find_package()` (вместо `Qt5`⟨*компонент*⟩) также требует указания хотя бы одного компонента, иначе исполнение конфигурационного файла Qt приведёт к выводу сообщения об ошибке.

— Не со всеми используемыми компонентами требуется связывание при помощи команды `target_link_libraries()`.

— Поиск и подключение некоторых компонент автоматически подключает другие, от которых они зависимы. Например, подключение библиотеки Widgets также приводит к подключению библиотек Core и Gui.

— Если приложение использует консоль, аргумент `WIN32` команды `add_executable()` необходимо не указывать. ▲

Если используется CMake версии 2.8.9 или выше, вместо команды `target_link_libraries()` можно использовать команду, которая определяется в конфигурационном файле компонента Core (становится доступной после вызова `find_package()`):

```
qt5_use_modules(
  ⟨имя_цели⟩
  [LINK_PUBLIC | LINK_PRIVATE] ⟨компонент_1⟩
  ...
```

```
  [LINK_PUBLIC | LINK_PRIVATE] ⟨компонент_n⟩)
```

— Необязательные аргументы LINK_PUBLIC/LINK_PRIVATE имеют такой же смысл, как аргументы PUBLIC/PRIVATE команд target_link_libraries() (п. 2.6.7) и т. п.

Эта команда используется следующим образом:

```
# ...

find_package(Qt5Widgets)

add_executable(⟨цель⟩ WIN32 ⟨файл_1⟩ ... ⟨файл_n⟩)

qt5_use_modules(⟨цель⟩ Widgets)

target_link_libraries(
  ⟨цель⟩
  ${Qt5Core_QTMAIN_LIBRARIES})
```

Здесь и в следующем фрагменте кода в переменной Qt5Core_QTMAIN_LIBRARIES содержится имя цели библиотеки qtmain, в которой определяется точка входа в программу для компилятора Microsoft Visual C++. Эта библиотека автоматически подключается, если приложение должно работать в системе Windows без отображения консоли (см. также пример подключения библиотек OpenCV на с. 258). Если приложению требуется выводить информацию на консоль, обращение к этой перемен-

ной необходимо убрать из команды `target_link_libraries()`, как и аргумент `WIN32` команды `add_executable()`.

Наконец, если версия CMake ниже 2.8.9 (но не ниже 2.8.3), необходимо устанавливать отдельными командами все требуемые настройки проекта в соответствующих переменных, которые инициализируются в результате вызова `find_package()`. В этом случае необходимо вызывать `find_package()` для каждого компонента по отдельности, в результате будут создаваться аналогичные переменные `Qt5`⟨*компонент*⟩`_INCLUDE_DIRS` и т. д.

```
# ...

find_package(Qt5Widgets)

set(FLAGS "${Qt5Widgets_EXECUTABLE_COMPILE_FLAGS}")
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${FLAGS}")

include_directories(${Qt5Widgets_INCLUDE_DIRS})
add_definitions(${Qt5Widgets_DEFINITIONS})

add_executable(⟨цель⟩ WIN32 ⟨файл_1⟩ ... ⟨файл_n⟩)

target_link_libraries(
  ⟨цель⟩
  ${Qt5Widgets_LIBRARIES}
  ${Qt5Core_QTMAIN_LIBRARIES})
```

В дальнейшем будем предполагать, что используется CMake версии 2.8.11 или выше.

Примеры

1) Рассмотрим простейшую программу с пользовательским интерфейсом на основе Qt [1] (ex-qt-hello.cpp):

```
#include <QApplication>
#include <QLabel>

int main(int nArgC, char *apszArgV[])
{
  QApplication app(nArgC, apszArgV);
  QLabel *pLabel = new QLabel("Hello Qt!");
  pLabel->show();
  //
  return app.exec();
}
```

Файл CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8.11)

project(ex-qt-hello)

find_package(Qt5Widgets)

add_executable(ex-qt-hello WIN32 ex-qt-hello.cpp)
target_link_libraries(ex-qt-hello Qt5::Widgets)
```

Здесь командой `cmake_minimum_required()` устанавливается совместимость с версией 2.8.11, начиная с которой поддерживается связывание с библиотеками Qt при помощи команды `target_link_libraries()`.

В Windows команды запуска CMake из каталога построения могут быть следующими:

```
set PATH=C:\Qt5.5.0\Tools\mingw492_32\bin;%PATH%
set QT_PATH=C:\Qt5.5.0\5.5\mingw492_32

cmake^
  -G "MinGW Makefiles"^
  -D CMAKE_PREFIX_PATH=%QT_PATH%^
  D:\Work\ex-qt-hello
```

Первая команда может пригодиться в случае использования компилятора gcc-MinGW, чтобы генератор CMake создал правила с вызовами инструментов из каталога установки библиотеки.

Собранное для платформы Windows приложение будет зависеть от библиотеки Qt `Qt5Widgets.dll`, которая, в свою очередь, зависит от `Qt5Core.dll` и `Qt5Gui.dll` (или, если сборка происходила в отладочной конфигурации, от их отладочных версий `Qt5Widgetsd.dll` и т. д.). Библиотека Core зависит от библиотек ICU (International Components for Unicode)[8]. Также, если используется компилятор MinGW, приложение будет зависеть от его библиотек поддержки

[8] http://site.icu-project.org/ (дата обращения: 02.03.2015).

исполнения программ. Таким образом, для работоспособности приложения необходимо поместить в каталог с ним (или поместить путь их расположения в переменную окружения PATH) библиотеки, перечисленные в табл. 3.6.

Таблица 3.6

Библиотеки, от которых зависит приложение, использующее компонент Qt Core

Библиотека	Примечание
icudt53.dll icuin53.dll icuuc53.dll	International Components for Unicode
libgcc_s_dw2-1.dll *	Поддержка способа обработки исключений DW2[9]
libstdc++-6.dll *	Стандартная библиотека C++
Qt5Core.dll Qt5Gui.dll Qt5Widgets.dll	Или их отладочные варианты

* Только для компилятора gcc-MinGW.

Все эти библиотеки расположены в подкаталоге bin каталога установки Qt.

2) Рассмотрим консольное приложение, использующее компонент Qt Concurrent:

```
#include <QApplication>
```

[9] http://tdm-gcc.tdragon.net/quirks (дата обращения: 02.03.2015).

```
#include <QtConcurrent>

#include <iostream>
#include <vector>
#include <functional>
#include <algorithm>

typedef std::vector <int> IntVector;

void print(const IntVector &rcVector)
{
  std::copy(
    rcVector.begin(), rcVector.end(),
    std::ostream_iterator <int> (std::cout, " "));
  std::cout << std::endl;
}

int main(int nArgC, char *apszArgV[])
{
  QApplication app(nArgC, apszArgV);
  //
  IntVector v1;
  for (int i = 0; i < 30; ++ i)
    v1.push_back(i);
  print(v1);
  //
  IntVector v2 =
```

```
    QtConcurrent::blockingMapped <IntVector> (
      v1, std::negate <int> ());
  print(v2);
}
```

Файл CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8.11)

project(ex-qt-concurrent)

find_package(Qt5 REQUIRED Widgets Concurrent)

add_executable(
  ex-qt-concurrent ex-qt-concurrent.cpp)
target_link_libraries(
  ex-qt-concurrent Qt5::Widgets Qt5::Concurrent)
```

Собранное приложение будет зависеть от библиотеки `Qt5Concurrent.dll` (или `Qt5Concurrentd.dll`), помимо перечисленных в табл. 3.6. *

Замечание: при подключении других библиотек Qt для запуска скомпилированного приложения могут также понадобиться файлы разделяемых модулей этих библиотек (`Qt5Multimedia`, `Qt5Network` и т. д.), а также библиотека `libwinpthread-1.dll`, которая реализует в компиляторе gcc-MinGW интерфейс POSIX Threads. ▲

3.3.2. Использование инструментов Qt

В процессе построения проектов, в которых используется Qt, приходится выполнять вызовы вспомогательных инструментов, генерирующих исходные коды [1; 8]. Одним из таких инструментов является *компилятор метаобъектов* (Meta-Object Compiler, moc), который реализует технологию метапрограммирования, используемую в Qt. Он анализирует заголовочные файлы проекта, в которых описаны классы, производные от `QObject` (базы иерархии классов Qt), и записывает в выходной cpp-файл код, который реализует поддержку отсутствующей в языке C++ интроспекции. На её основе реализуются различные механизмы Qt, такие как технология обмена сообщениями при помощи сигналов и слотов. Если разработчику нужна поддержка каких-либо из этих возможностей в его классах, он оформляет их соответствующим образом (добавляет макрос `Q_OBJECT`, распознаваемый инструментом moc, в объявление класса и т. д.):

```
#include <QObject>

class MyObject : public QObject
{
  Q_OBJECT
  // ...
private slots:
  //
  void slot1();
```

```
  // ...
};
```

Кроме этого, необходимо обеспечить запуск инструмента moc для заголовочного файла с его объявлением во время построения проекта (рис. 3.2).

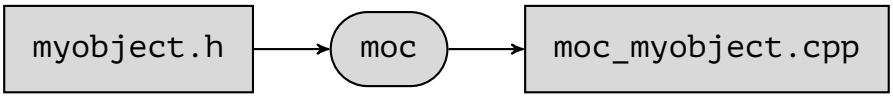

Рис. 3.2. Генерирование метаинформации при помощи инструмента moc

В системе CMake для этих целей используется команда `qt5_wrap_cpp()`, которая становится доступной после нахождения библиотеки `Core` при помощи команды `find_package()`:

```
qt5_wrap_cpp(
  ⟨имя_переменной⟩
  ⟨заголовочный_файл1⟩ ... ⟨заголовочный_файлm⟩
  [OPTIONS ⟨аргумент1⟩ ... ⟨аргументn⟩])
```

Команда создаёт правила для генерирования файлов в выходном каталоге из заданных заголовочных файлов проекта. Список путей к генерируемым файлам в каталоге построения записывается в заданную переменную, которую затем можно использовать в команде `add_executable()` для их добавления к цели. После необязательного аргумента `OPTIONS` можно указать дополнительные ключи командной строки для передачи инструменту moc. Команда также добавляет для каталога текущего подпроекта все необходимые определения препроцессора.

Пример

Пусть требуется создать приложение с оконным пользовательским интерфейсом, в главном меню которого находятся команды «Открыть файл» и «Выход» (рис. 3.3). Выбор пользователем первой команды должен приводить к отображению диалогового окна для запроса имени текстового файла, второй — к закрытию программы.

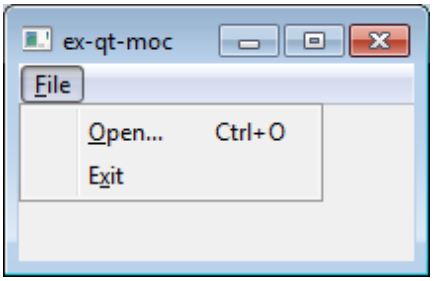

Рис. 3.3. Оконное приложение с меню, использующее библиотеки Qt

Заголовочный файл с объявлением класса главного окна приложения (`main-window.h`):

```
#ifndef MAIN_WINDOW_H__
#define MAIN_WINDOW_H__

#include <QMainWindow>

class MainWindow : public QMainWindow
{
  Q_OBJECT
  //
public:
```

```
  //
  MainWindow(QWidget *pParent = 0);
  //
private slots:
  //
  void onFileOpen();
};

#endif    // MAIN_WINDOW_H__
```

Файл реализации главного окна (`main-window.cpp`):

```
#include "main-window.h"

#include <QMenuBar>
#include <QKeySequence>
#include <QFileDialog>

MainWindow::MainWindow(QWidget *pParent)
  : QMainWindow(pParent)
{
  QMenu *pMenuFile = menuBar()->addMenu(tr("&File"));
  pMenuFile->addAction(
    tr("&Open..."), this,
    SLOT(onFileOpen()), QKeySequence::Open);
  pMenuFile->addAction(
    tr("E&xit"), this,
    SLOT(close()), QKeySequence::Quit);
```

```
}

void MainWindow::onFileOpen()
{
  QString fileName = QFileDialog::getOpenFileName(
    this,
    tr("Open File"), "", tr("Text Files (*.txt)"));
}
```

Поскольку в классе основного окна определена функция `onFileOpen()`, работающая в качестве слота, в определение класса добавляется макрос `Q_OBJECT`. В описании проекта необходимо указать правила, по которым при помощи инструмента `moc` в каталоге построения должен генерироваться файл мета-информации о классе `MainWindow`, который должен участвовать в построении приложения.

Точка входа в программу (файл `ex-qt-moc.cpp`):

```
#include "main-window.h"

#include <QApplication>

int main(int nArgC, char *apszArgV[])
{
  QApplication app(nArgC, apszArgV);
  MainWindow window;
  window.show();
  return app.exec();
```

```
}
```

Файл CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8.11)

project(ex-qt-moc)

find_package(Qt5Widgets REQUIRED)

set(HEADERS main-window.h)
qt5_wrap_cpp(MOC_FILES ${HEADERS})

add_executable(
  ex-qt-moc WIN32
  ex-qt-moc.cpp main-window.cpp
  ${HEADERS} ${MOC_FILES})
target_link_libraries(
  ex-qt-moc Qt5::Widgets)
```

Во время построения в выходном каталоге создаётся файл moc_main-window.cpp по содержимому приведённого выше файла main-window.h. Полный путь к генерируемому файлу записывается в переменную MOC_FILES и затем используется в списке входных файлов в команде add_executable(). *

Аналогичным образом в CMake реализована поддержка *компилятора ресурсов* Qt (Resource Compiler, rcc). Этот инструмент используется для встраивания в исполняемые файлы дво-

ичных данных, таких как изображения и пиктограммы, отображаемые пользовательским интерфейсом. В отличие от похожего механизма в системе Windows, Qt реализует поддержку ресурсов кроссплатформенным способом: преобразованием двоичных файлов в текст на языке C++ с объявлениями массивов. Для описания ресурсов применяются текстовые файлы с расширением «.qrc», использующие язык XML и редактируемые средой разработки Qt Creator (рис. 3.4).

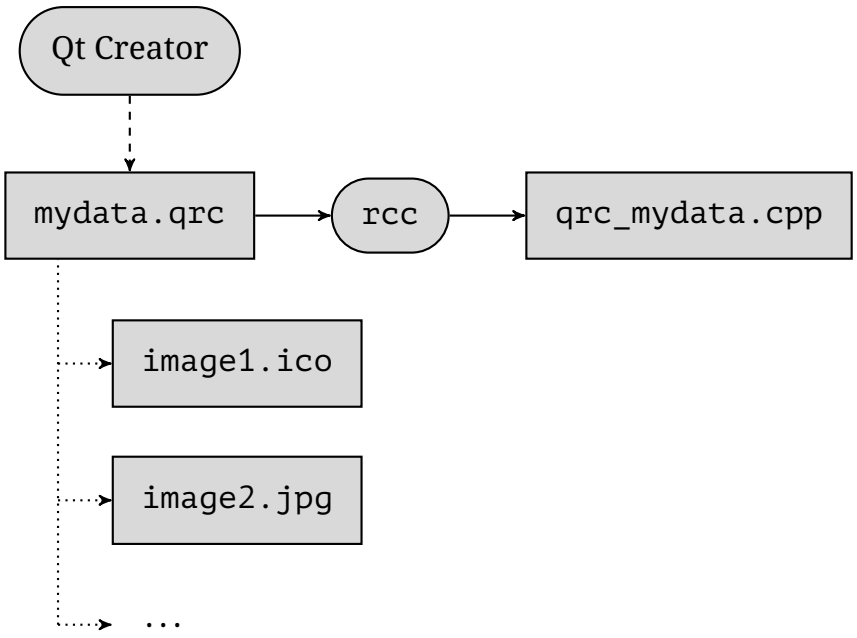

Рис. 3.4. Генерирование ресурсов при помощи инструмента rcc

Для добавления вызовов компилятора ресурсов в набор правил построения в CMake используется команда:

```
qt5_add_resources(
  ⟨имя_переменной⟩
  ⟨файл_описания_ресурсов1⟩ ... ⟨файл_описания_ресурсовm⟩
  [OPTIONS ⟨аргумент1⟩ ... ⟨аргументn⟩])
```

Пример

Пусть требуется добавить отображаемую пиктограмму к основному окну приложения из примера на с. 298. Для этого сначала необходимо поместить файл с изображением в каталог проекта, допустим в подкаталог `resources`. Далее следует создать файл описания ресурсов, например использовав инструмент Qt Creator (рис. 3.5).

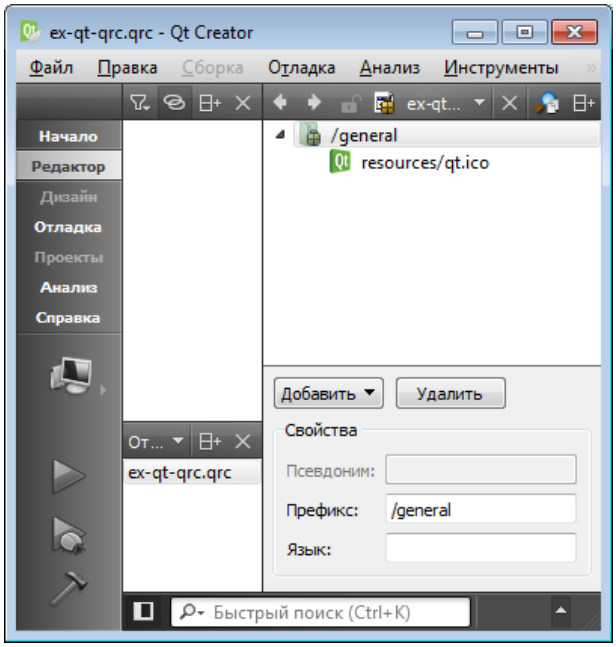

Рис. 3.5. Редактирование файла описания ресурсов при помощи среды разработки Qt Creator

Таким образом, структура каталога проекта будет такой, как изображена на рис. 3.6.

Содержимое сгенерированного инструментом Qt Creator файла `ex-qt-qrc.qrc`:

```
<RCC>
```

```
⟨каталог проекта⟩
├─ resources
│  └─ qt.ico
├─ ex-qt-qrc.cpp
├─ ex-qt-qrc.qrc  ... описание ресурсов, редактируется Qt Creator
└─ CMakeLists.txt
```

Рис. 3.6. Структура каталога проекта с ресурсами

```
    <qresource prefix="/general">
        <file>resources/qt.ico</file>
    </qresource>
</RCC>
```

Содержимое файла `ex-qt-qrc.cpp`:

```
#include <QApplication>
#include <QLabel>
#include <QIcon>

int main(int nArgC, char *apszArgV[])
{
  QApplication app(nArgC, apszArgV);
  QApplication::setWindowIcon(
    QIcon(":/general/resources/qt.ico"));
  //
  QLabel label("Hello Qt!");
  label.show();
  //
  return app.exec();
```

```
}
```

Здесь статический метод `QApplication::setWindowIcon()` устанавливает пиктограмму изображения, которая загружается из файла с заданным путём. Двоеточие в начале означает, что путь указывается относительно дерева ресурсов, хранимого внутри исполняемого файла.

Файл `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 2.8.11)

project(ex-qt-qrc)

find_package(Qt5Widgets REQUIRED)

qt5_add_resources(
  QRC_WRAPPERS ex-qt-qrc.qrc
  OPTIONS -no-compress)

add_executable(
  ex-qt-qrc WIN32
  ex-qt-qrc.cpp ${QRC_WRAPPERS})
target_link_libraries(
  ex-qt-qrc Qt5::Widgets)
```

Здесь в команде `qt5_add_resources()` определяются дополнительные аргументы командной строки, которые передаются инструменту qrc. Ключ `-no-compress` отключает сжатие

данных, выполняемое по умолчанию (алгоритм zlib[10]). Это может быть полезным для экономии времени построения, особенно если ресурсы уже находятся в сжатом виде (например, изображения в формате PNG).

В результате построения в выходном каталоге создаётся файл `qrc_ex-qt-qrc.cpp`. Полный путь к нему на этапе исполнения CMake записывается в переменную `QRC_WRAPPERS`, которая затем используется в списке входных файлов цели в команде `add_executable()`. *

Замечание: при помощи других ключей инструмента qrc можно настроить уровень сжатия от 1 до 9 (`-compress` ⟨*уровень*⟩) и пороговое отношение сжатого размера к исходному (в процентах), начиная с которого файл будет помещён в ресурсы в несжатом виде (`-threshold` ⟨*отношение*⟩). ▲

Другим инструментом Qt, для которого также есть поддержка в CMake, является *компилятор пользовательского интерфейса* (User Interface Compiler, uic). Текст реализации главного окна в примере на с. 305 показывает, что даже отображение меню из двух команд вынуждает разработчика писать множество строк кода, не говоря уже о программировании сложных диалоговых окон с большим количеством графических элементов управления. Для облегчения этих задач в Qt используется визуальный редактор Qt Designer (или Qt Creator), при помощи которого можно «нарисовать» окна с требуемыми элементами управления, размерами и другими свойствами. Вся информация сохраняется в текстовом файле с расширением «`.ui`», имею-

[10] http://www.zlib.net/ (дата обращения: 05.03.2015).

щем структуру XML. В процессе построения компилятор пользовательского интерфейса по этому файлу генерирует в выходном каталоге заголовочный файл, содержащий весь код инициализации внешнего вида окна (рис. 3.7).

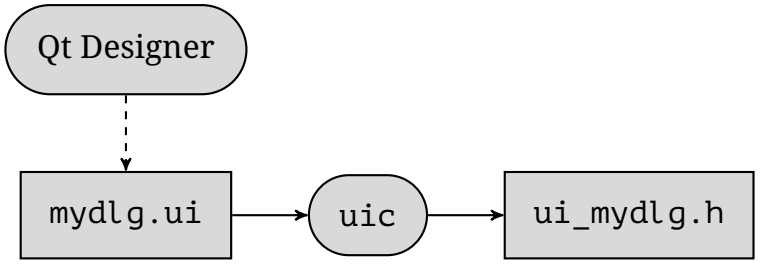

Рис. 3.7. Преобразование описания ресурса в код на C++ при помощи инструмента uic

В CMake для создания правил вызова компилятора пользовательского интерфейса существует команда `qt5_wrap_ui()`, которая определяется после успешного поиска пакета `Qt5Widgets`. Её синтаксис аналогичен двум предыдущим командам:

```
qt5_wrap_ui(
  ⟨имя_переменной⟩
  ⟨файл_интерфейса1⟩ ... ⟨файл_интерфейсаm⟩
  [OPTIONS ⟨аргумент1⟩ ... ⟨аргументn⟩])
```

Пример

Пусть при помощи инструмента Qt Designer создан макет диалогового окна (рис. 3.8), которое необходимо использовать в качестве основного модального окна приложения.

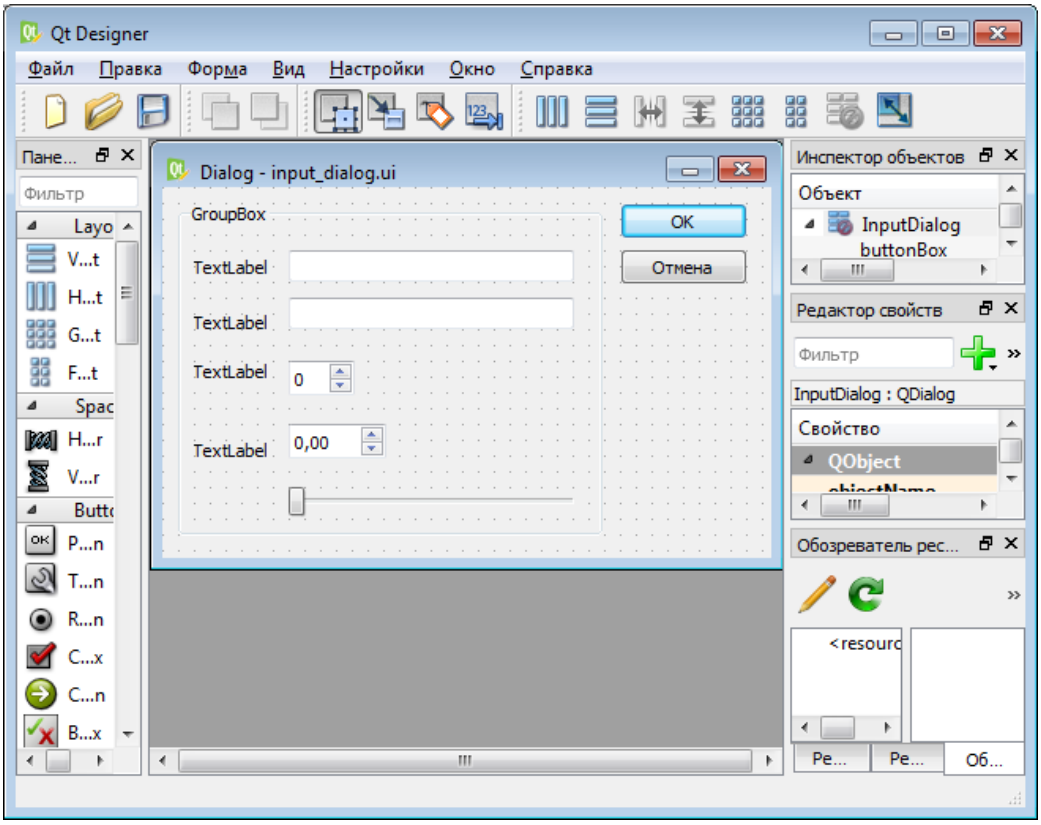

Рис. 3.8. Редактирование файла пользовательского интерфейса при помощи инструмента Qt Designer

Для удобства дальнейшей работы над проектом можно создать заголовочный файл с описанием класса окна (`InputDialog`). Таким образом, структура каталога проекта будет такой, как на рис. 3.9.

Содержимое файла `input_dialog.h`:

```
#ifndef INPUT_DIALOG_H__
#define INPUT_DIALOG_H__

#include "ui_input_dialog.h"

#include <QDialog>
```

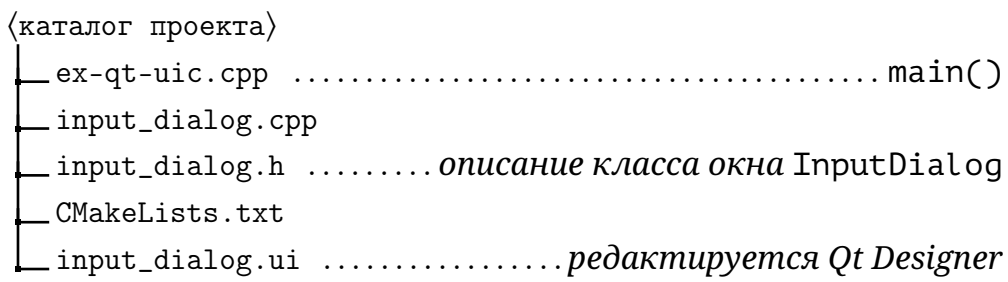

Рис. 3.9. Структура каталога проекта с описанием пользовательского интерфейса

```
class InputDialog :
  public QDialog, public Ui::InputDialog
{
public:
  //
  InputDialog(QWidget *pParent = 0);
};

#endif    // INPUT_DIALOG_H__
```

Здесь заголовочный файл `ui_input_dialog.h` генерируется в каталоге построения инструментом uic на основе содержимого файла пользовательского интерфейса `input_dialog.ui`. В нём определён класс `Ui::InputDialog`, отвечающий за внешний вид окна.

Файл `input_dialog.cpp` с реализацией конструкцией класса `InputDialog`:

```
#include "input_dialog.h"
```

```
InputDialog::InputDialog(QWidget *pParent)
  : QDialog(pParent)
{
  setupUi(this);
}
```

Здесь метод `setupUi()`, унаследованный от сгенерированного класса `Ui::InputDialog`, исполняет весь код инициализации окна.

Файл `ex-qt-uic.cpp`:

```
#include "input_dialog.h"

#include <QApplication>

int main(int nArgC, char *apszArgV[])
{
  QApplication app(nArgC, apszArgV);
  InputDialog dialog;
  dialog.exec();
}
```

Файл `CMakeLists.txt`:

```
cmake_minimum_required(VERSION 2.8.11)

project(ex-qt-uic)
```

```
set(CMAKE_INCLUDE_CURRENT_DIR ON)

find_package(Qt5Widgets REQUIRED)

qt5_wrap_ui(
  UI_WRAPPERS input_dialog.ui)

add_executable(
  ex-qt-uic WIN32
  ex-qt-uic.cpp input_dialog.cpp input_dialog.h
  ${UI_WRAPPERS})
target_link_libraries(
  ex-qt-uic Qt5::Widgets)
```

Здесь значение истины присваивается специальной переменной CMake `CMAKE_INCLUDE_CURRENT_DIR`, чтобы каталог построения добавлялся в список путей для поиска заголовочных файлов. Это необходимо для того, чтобы директива `#include`, которая подключает файл `ui_input_dialog.h` и которая использована в файле `input_dialog.h`, сработала правильно. *

Замечание: в целях сокращения объёма примера в нём отсутствует код обработки событий пользовательского интерфейса на основе сигналов и слотов. В реальных проектах в классах окон определяются слоты. Следовательно, эти классы также нужно передавать компилятору метаинформации при помощи команды `qt5_wrap_cpp()`, аналогично примеру на с. 305 (см. также следующий пример). ▲

В качестве альтернативы приведённым выше командам `qt5_...()` можно присвоить значения истины специальным переменным `CMAKE_AUTOMOC` (доступна в CMake начиная с версии 2.8.6), `CMAKE_AUTORCC` и `CMAKE_AUTOUIC` (доступны начиная с 3.0). Этими переменными инициализируются соответствующие свойства целей, которые включают генерирование специальных правил построения, автоматически определяющих необходимость запуска инструментов для нужных файлов. Анализ исходных текстов учитывает директивы подключения заголовочных файлов с именами `ui_....h` и т. д. Поддерживаются Qt версий 4 и 5. При использовании этих режимов упрощается описание проекта и добавление новых исходных файлов, однако несколько замедляется процесс построения. Для передачи инструментам moc и т. д. дополнительных ключей командной строки можно использовать специальные переменные `CMAKE_AUTOMOC_MOC_OPTIONS`, `CMAKE_AUTORCC_OPTIONS` и `CMAKE_AUTOUIC_OPTIONS`.

Пример

Пусть требуется добавить к предыдущему примеру (с. 314) обработку событий ввода: реализовать для пользователя возможность нажать кнопку «OK» только в том случае, если оба текстовых поля ввода заполнены непустыми значениями. Пусть также требуется добавить к диалоговому окну пиктограмму, аналогично примеру на с. 310. Структура каталога проекта, таким образом, будет выглядеть так, как на рис. 3.10.

Файл `input_dialog.h`:

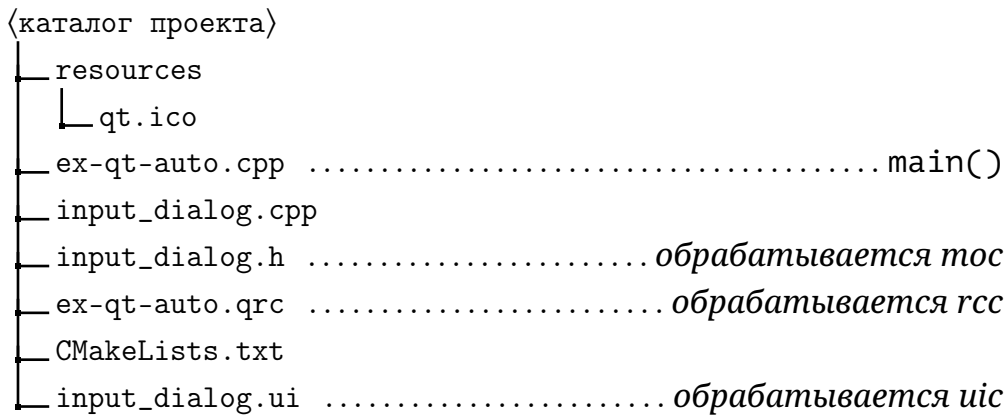

Рис. 3.10. Структура каталога проекта с автоматическим запуском инструментов Qt

```
#ifndef INPUT_DIALOG_H__
#define INPUT_DIALOG_H__

#include "ui_input_dialog.h"

#include <QDialog>

class InputDialog :
  public QDialog, public Ui::InputDialog
{
  Q_OBJECT
  //
public:
  //
  InputDialog(QWidget *pParent = 0);
  //
private slots:
```

```
  //
  void on_lineEdit_textChanged();
};

#endif    // INPUT_DIALOG_H__
```

Если указатель на первое поле ввода формы (рис. 3.8) называется lineEdit (определяется в редакторе Qt Designer), по правилам Qt объявленный в классе окна слот с именем on_lineEdit_textChanged() будет автоматически связан с сигналом textChanged(), поступающим от объекта, на который указывает lineEdit, всякий раз, когда содержимое поля ввода изменяется.

Файл input_dialog.cpp:

```
#include "input_dialog.h"

#include <QPushButton>

InputDialog::InputDialog(QWidget *pParent)
  : QDialog(pParent)
{
  setupUi(this);
  //
  connect(
    lineEdit_2, SIGNAL(textChanged(const QString &)),
    this, SLOT(on_lineEdit_textChanged()));
  //
```

```
  on_lineEdit_textChanged();
}

void InputDialog::on_lineEdit_textChanged()
{
  QPushButton *pButton = buttonBox->button(
    QDialogButtonBox::Ok);
  pButton->setEnabled(
    !lineEdit->text().isEmpty() &&
    !lineEdit_2->text().isEmpty());
}
```

Здесь в конструкторе сначала происходит связывание того же слота `on_lineEdit_textChanged()` с тем же сигналом `textChanged()`, но уже от второго поля ввода (`lineEdit_2`). Дальше этот метод вызывается непосредственно, чтобы состояние диалогового окна соответствовало его начальному содержимому с момента его создания. В самом обработчике происходит обращение к кнопке «OK» в группе `buttonBox` (создаётся в редакторе), для которой состояние доступности для нажатия определяется в зависимости от непустоты обоих полей ввода.

Файл `ex-qt-auto.cpp`:

```
#include "input_dialog.h"

#include <QApplication>
#include <QIcon>
```

```
int main(int nArgC, char *apszArgV[])
{
  QApplication app(nArgC, apszArgV);
  QApplication::setWindowIcon(
    QIcon(":/general/resources/qt.ico"));
  InputDialog dialog;
  dialog.exec();
}
```

Содержимое файла описания ресурсов (ex-qt-auto.qrc) и пользовательского интерфейса (input_dialog.ui) здесь имеют то же содержание, что и соответствующие файлы в двух предыдущих примерах.

Файл CMakeLists.txt:

```
cmake_minimum_required(VERSION 3.0)

project(ex-qt-auto)

set(CMAKE_INCLUDE_CURRENT_DIR ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)
set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTORCC_OPTIONS -no-compress)

find_package(Qt5Widgets REQUIRED)

add_executable(
```

```
  ex-qt-auto WIN32
  ex-qt-auto.cpp ex-qt-auto.qrc
  input_dialog.cpp input_dialog.h)
target_link_libraries(
  ex-qt-auto Qt5::Widgets)
```

Здесь вначале запрашивается совместимость с CMake версии 3.0, так как именно с неё доступны переменные CMAKE_AUTORCC и CMAKE_AUTOUIC. Для того чтобы вызов инструмента rcc мог выполняться автоматически, необходимо, чтобы файл описания ресурсов (ex-qt-auto.qrc) был добавлен к цели исполняемого файла (команда add_executable()). *

3.3.3. Локализация приложения

Qt имеет встроенную поддержку простой в использовании системы многоязычного пользовательского интерфейса с возможностью переключения языка приложения «на лету». Чтобы заменить в работающей программе все зависящие от языка сообщения для заданного перевода, необходимо при помощи специального программного интерфейса Qt загрузить соответствующий языковой модуль из двоичного файла с расширением «.qm». Схема подготовки таких файлов для приложения приведена на рис. 3.11.

Первым используется инструмент lupdate, задача которого заключается в сборе из исходных текстов всех строк, которые видны пользователю. На вход ему подаются файлы описания пользовательского интерфейса (с расширением «.ui»), а также

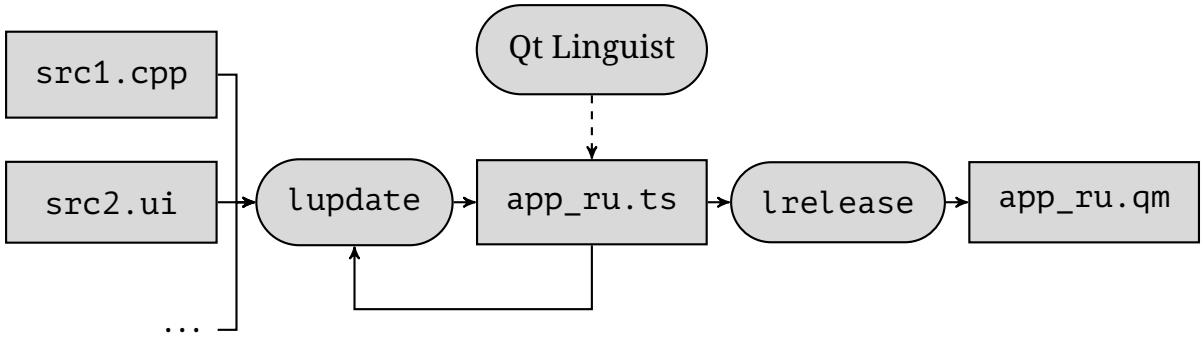

Рис. 3.11. Создание файлов локализации при помощи инструментов lupdate и lrelease

исходные модули и заголовочные файлы. Поддерживаются языки C++, Java и т. д. В исходных текстах все необходимые строковые литералы помещаются в вызовы функции `QObject::tr()` и других подобных функций и макросов, которые служат двум целям. Во-первых, с их помощью инструменту lupdate, который проводит синтаксический анализ кода, указывается, какие строки требуют перевода. Во-вторых, во время выполнения программы они реализуют саму подмену строк.

Результатом работы инструмента lupdate является текстовый файл с расширением «`.ts`» (от «Translation Source»). Этот файл имеет формат XML и содержит все собранные строки вместе с их местоположением в исходном коде, контекстом использования и комментариями для переводчика. Дальше переводчик интерфейса может использовать инструмент Qt Linguist для визуального редактирования этого файла (см. пример далее). Изначально каждая строка в нём имеет состояние «не завершено», которое меняется после заполнения поля перевода. Что важно, файл «`.ts`» является одновременно выходным и входным для инструмента lupdate: если повторно запустить

его для изменившихся исходных кодов, он добавит в файл новые данные, оставив нетронутыми уже переведённые строки.

Последним в цепочке инструментов используется lrelease, который компилирует текстовый файл «`.ts`» в двоичный «`.qm`».

Для автоматизации построения файлов перевода в проекте средствами CMake инструментарий Qt предоставляет конфигурационный файл модуля `LinguistTools`, который определяет следующую команду:

`qt5_create_translation(`
⟨*имя_переменной*⟩
⟨*путь*$_1$⟩ `...` ⟨*путь*$_m$⟩
[`OPTIONS` ⟨*аргумент*$_1$⟩ `...` ⟨*аргумент*$_n$⟩]`)`

Эта команда определяет в текущем подпроекте правила (при помощи команды `add_custom_command()`, п. 2.10.5) для генерирования файлов TS и QM. В переменную, имя которой передаётся первым аргументом, записываются полные пути к создаваемым результирующим QM-файлам. Эти пути можно передавать затем командам создания целей (`add_executable()` и т. д.) в качестве исходных файлов, чтобы построение цели при необходимости вызывало и правила создания этих файлов. В качестве альтернативы можно использовать эти пути как зависимости для цели, создаваемой командой `add_custom_target()` (п. 2.10.4), чтобы файлы перевода можно было генерировать запуском отдельной фальшивой цели. Особенностью команды по сравнению с другими подобными является то, что часть генерируемых файлов (TS) создаётся в каталоге проекта рядом с ис-

ходными файлами вместо каталога построения. Это может потенциально создавать некоторые проблемы (см. пример ниже).

В качестве аргументов путей команде передаются:

— исходные файлы проекта, которые нужно обработать командой lupdate;

— каталоги проекта, которые передаются команде lupdate для поиска исходных файлов для обработки;

— пути к генерируемым промежуточным файлам TS.

Относительные пути к исходным файлам и файлам TS интерпретируются по отношению к каталогу подпроекта. Конечные файлы QM создаются в каталоге построения текущего подпроекта или, если для соответствующего ему TS-файла определено значение свойства `OUTPUT_LOCATION`, в определяемом им каталоге.

После необязательного аргумента `OPTIONS` можно указать дополнительные аргументы командной строки для инструмента lupdate.

Следующая команда добавляет только правила для построения QM-файлов из существующих TS-файлов при помощи инструмента lrelease:

```
qt5_add_translation(
  ⟨имя_переменной⟩
  ⟨TS-файл_1⟩ ... ⟨TS-файл_m⟩)
```

Здесь передаваемое имя переменной имеет тот же смысл, что и для предыдущей команды.

Пример

Пусть требуется реализовать приложение, которое выводит сообщение в простом диалоговом окне, при этом заголовок окна и само сообщение должны быть выведены на языке, установленном в качестве системного. Эту задачу выполняет следующий код (файл ex-qt-translate.cpp):

```
#include <QApplication>
#include <QTranslator>
#include <QMessageBox>

int main(int nArgC, char *apszArgV[])
{
  QApplication app(nArgC, apszArgV);
  //
  QTranslator translator;
  const bool cbLoaded = translator.load(
    QLocale::system(), "ex-qt-translate_");
  if (cbLoaded)
    app.installTranslator(&translator);
  //
  QMessageBox::information(
    0,
    QApplication::translate(
      "main", "Application"),
    QApplication::translate(
      "main", "A localized message"));
}    // main()
```

Здесь вначале создаётся объект типа `QTranslator`, при помощи которого производится попытка загрузить файл перевода с именем «`ex-qt-translate_`⟨*язык*⟩`.qm`» в рабочем каталоге. В случае успеха этот объект устанавливается в качестве объекта локализации приложения. Дальше выводится сообщение функцией `QMessageBox::information()`. Передаваемые ей сообщение и заголовок обёрнуты в вызовы функции `QCoreApplication::translate()`, которая и выполняет подмену строк в соответствии с установленным объектом `QTranslator`.

Файл `CMakeLists.txt`, который определяет правила создания файла перевода для русского языка (приводимая схема в целом соответствует рекомендациям на Wiki-странице проекта CMake[11]):

```
cmake_minimum_required(VERSION 2.8.11)

project(ex-qt-translate)

find_package(Qt5 REQUIRED Widgets LinguistTools)

set(
  FILES_TO_TRANSLATE
  ex-qt-translate.cpp)

set(
  TS_FILES
```

[11] http://www.cmake.org/Wiki/CMake:How_To_Build_Qt4_Software (дата обращения: 10.03.2015).

```
  translations/ex-qt-translate_ru_RU.ts)

option(
  UPDATE_TRANSLATIONS
  "Update source translation (translations/*.ts)."
  ON)

if(UPDATE_TRANSLATIONS)
  message(
    STATUS "UPDATE_TRANSLATIONS option is set.")
  qt5_create_translation(
    QM_FILES ${FILES_TO_TRANSLATE} ${TS_FILES})
else()
  qt5_add_translation(
    QM_FILES ${TS_FILES})
endif()

add_custom_target(
  translations
  DEPENDS ${QM_FILES})

set_property(
  DIRECTORY
  PROPERTY CLEAN_NO_CUSTOM TRUE)

add_executable(
  ex-qt-translate WIN32
```

```
  ${FILES_TO_TRANSLATE})
target_link_libraries(
  ex-qt-translate Qt5::Widgets)
```

Здесь при помощи команды `find_package()` запрашивается использование модуля `LinguistTools`, в котором определены команды, создающие правила генерирования файлов перевода. Переменной `FILES_TO_TRANSLATE` присваивается список исходных файлов, которые должны обрабатываться инструментом lupdate. По мере развития проекта сюда можно будеть добавлять другие файлы с расширениями «`.cpp`», «`.h`», «`.ui`» и т. д. Переменной `TS_FILES` присваивается список путей к промежуточным TS-файлам, создаваемым в каталоге проекта. Их имена должны иметь приведённый выше формат («`ex-qt-translate_`⟨*язык*⟩`.qm`», где ⟨*язык*⟩ является именем, возвращаемым функцией `QLocale::name()` для тех языков, для которых требуется локализация). При помощи логической настройки проекта `UPDATE_TRANSLATIONS` определяется, нужно ли генерировать правила для создания промежуточных TS-файлов в каталоге проекта или только результирующих QM-файлов. Последнее может пригодиться для ускорения процесса сборки, если разработчик в настоящее время не занят обновлением пользовательского интерфейса.

Для удобства запуска правил создания файлов перевода при помощи команды `add_custom_target()` создаётся фальшивая цель с именем `translations`, которая зависит от QM-файлов.

Вызов команды qt5_create_translation() имеет один нежелательный побочный эффект. Дело в том, что цель clean, создаваемая генераторами для make-файлов, удаляет все сгенерированные в результате выполнения правил файлы. То же самое относится к цели translations, результатом выполнения которой является создание файлов TS в каталоге проекта. После выполнения цели clean эти файлы будут также удалены, что может привести к потере результатов долгой работы по локализации. Чтобы обойти эту проблему, у каталога текущего подпроекта устанавливается в истину свойство CLEAN_NO_CUSTOM, что приводит к исключению файлов, генерируемых правилами команд add_custom_command(), из числа удаляемых целью clean. В результате, правда, при исполнении этой цели также не удаляются файлы QM в каталоге построения.

Чтобы правила создания TS-файлов сработали без ошибок, необходимо в самом начале в каталоге проекта вручную создать пустой каталог translations. Далее можно выполнить следующие команды (на примере системы Linux Mint/Ubuntu):

```
cmake ~/work/ex-qt-translate    # каталог проекта
make                    # сборка исполняемого файла
make translations       # генерирование TS и «пустых» QM
/usr/lib/x86_64-linux-gnu/qt5/bin/linguist # правка TS
make translations       # генерирование QM
```

После выполнения третьей команды каталог проекта будет иметь структуру, представленную на рис. 3.12.

```
⟨каталог проекта⟩
├─ translations
│  └─ ex-qt-translate_ru_RU.ts  .... создаётся инструментом lupdate
├─ ex-qt-translate.cpp
└─ CMakeLists.txt
```

Рис. 3.12. Структура каталога проекта с файлом перевода, сгенерированным инструментом lupdate

Четвёртая команда запускает редактор Qt Linguist (рис. 3.13).

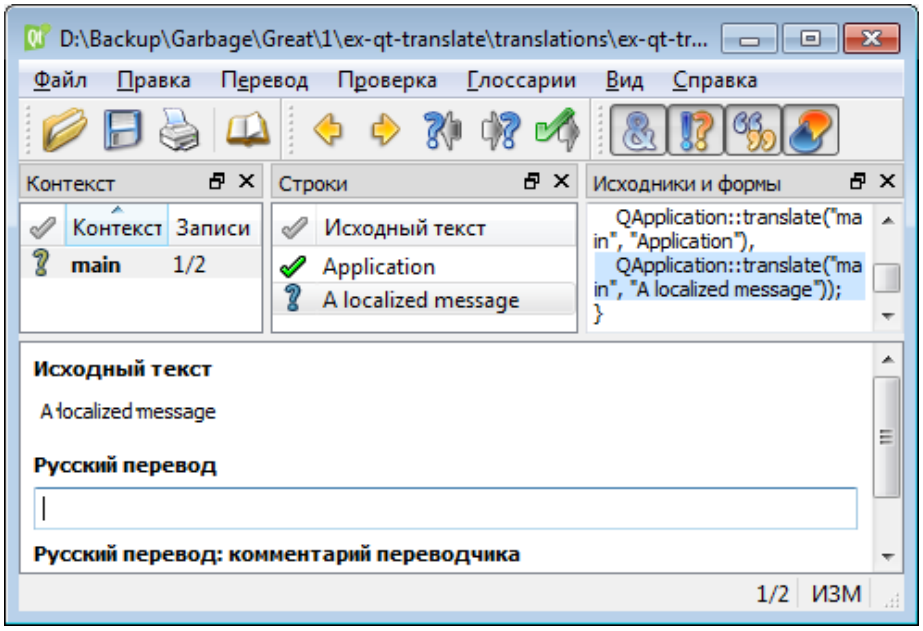

Рис. 3.13. Редактирование исходного файла перевода при помощи инструмента Qt Linguist

После завершения редактирования последняя команда создаёт в каталоге построения файл `ex-qt-translate_ru_RU.qm`.

*

Замечания:

— Пакет `LinguistTools` не определяет целей для связывания с библиотеками, поэтому для его использования

не нужно передавать дополнительных аргументов командам `target_link_libraries()`.

— При запуске приложения этого примера из интегрированной среды Microsoft Visual Studio для него в качестве рабочего каталога устанавливается каталог построения (в котором расположены файлы проекта). Так как QM-файл генерируется в этом же каталоге, запускаемое приложение находит его, хотя оно и расположено в подкаталоге с именем конфигурации («Debug» и т. д.). Если запустить приложение, например из проводника Windows, оно уже не сможет найти его. ▲

3.3.4. Установка приложения

Приложения, использующие библиотеки Qt, имеют зависимости от некоторых вспомогательных файлов. Это могут быть разделяемые библиотеки Qt, загружаемые модули (plugins), библиотеки поддержки времени исполнения компиляторов, файлы локализации и т. д. В таких условиях становится актуальной задача автоматизации установки приложения или выгрузки его вместе со всеми необходимыми для его работы файлами в отдельный каталог для подготовки пакета установки. Для облегчения этой задачи в CMake, помимо команды `install()` (п. 2.10.3), также может быть использован стандартный модуль `DeployQt4`. Несмотря на название, он вполне работоспособен по отношению к приложениям, разрабатываемым с помощью Qt версии 5, если для этого приложить небольшие усилия (см. пример ниже). В нём определено несколько команд, однако

основной, из которой вызываются все остальные, является команда `install_qt4_executable()`. Она при помощи команды `install(CODE ...)` добавляет к цели `install` код, который выполняет копирование в каталог установки требуемых библиотечных модулей. Команда использует другой стандартный модуль — `BundleUtilities`. Он предназначен для подготовки автономных пакетов приложений (application bundles) системы OS X, однако также способен создавать в других системах директории, имеющие свойства пакетов (автономность, переносимость на другие компьютеры).

Замечание: как станет понятно из приведённого далее примера, на самом деле модуль `DeployQt4` можно использовать для установки разделяемых модулей, от которых зависит любое приложение, а не только использующее библиотеки Qt. ▲

Синтаксис команды `install_qt4_executable()`:

```
install_qt4_executable(
  ⟨исполняемый_файл⟩ [⟨модули_Qt⟩ [⟨библиотеки⟩
  [⟨каталоги⟩ [⟨каталог_модулей_Qt⟩ [⟨писать_qt.conf⟩
  [⟨имя_компонента⟩]]]]]])
```

Если в аргументах команды со второго по четвёртый необходимо передать несколько элементов, они отделяются точками с запятой и помещаются в двойные кавычки, чтобы CMake не воспринял их как несколько отдельных аргументов (п. 2.2.2).

Команда выполняет копирование всех несистемных разделяемых и загружаемых модулей, от которых зависит заданный

исполняемый модуль. В результате в каталог приложения копируются все нужные библиотеки Qt, если только они не установлены вместе с системой (например, установлены из репозиториев Linux). Затем команда выполняет (зависящую от системы) правку исполняемого модуля и разделяемых модулей так, чтобы их можно было копировать и запускать на других системах, для которых эти модули совместимы на уровне системных библиотек. Зависимости определяются средствами стандартного модуля CMake `GetPrerequisites`, который пытается использовать для этого внешние утилиты `dumpbin` (поставляется вместе со средствами разработки Microsoft, также может быть установлена отдельно), `objdump` (входит в состав MinGw), `ldd` (Linux и т. п.) и `otool` (OS X).

— В качестве аргумента ⟨*исполняемый_файл*⟩ команде передаётся относительный путь к исполняемому файлу, который требуется проанализировать на зависимости. Путь задаётся относительно каталога установки (переменная окружения `DESTDIR` + специальная переменная CMake `CMAKE_INSTALL_PREFIX`). На момент установки файл должен быть уже скопирован в указанное место, т. е. перед командной `install_qt4_executable()` нужно вызвать команду `install(TARGETS ...)` для цели приложения.

— В необязательном аргументе ⟨*модули_Qt*⟩ перечисляются загружаемые модули Qt, от которых зависит приложение и которые также должны быть установлены в соответствующий каталог. В отличие от разделяемых модулей библиотек, зависимость от которых определена

в исполняемом файле, загружаемые модули активизируются во время работы программы вызовами специальных API (см. пример на с. 230), поэтому команда `install_qt4_executable()` не может самостоятельно определить эти зависимости. В аргументе перечисляются либо пути к библиотечным файлам, либо их имена (для модулей из состава Qt: `qjpeg`, `qsqlite` и т. д.). В последнем случае пути должны храниться в переменных CMake «`QT_`⟨*имя_в_верхнем_регистре*⟩`_PLUGIN_(DEBUG|RELEASE)`». Такие переменные создаются модулем поиска пакета Qt4, но не Qt5. При использовании Qt5 их можно создавать самостоятельно.

— В необязательном аргументе ⟨*библиотеки*⟩ в дополнение к модулям передаются пути к другим загружаемым модулям, разделяемым библиотекам и исполняемым файлам, которые также нужно исправить для возможности автономной работы.

— В необязательном аргументе ⟨*каталоги*⟩ перечисляются дополнительные пути к каталогам, в которых будет выполняться поиск библиотечных модулей, представляющих зависимости. Также для поиска будут использоваться каталоги, указанные в системных настройках (например, перечисленные в переменной окружения `PATH` в Windows), а также перечисленные в переменных CMake `QT_LIBRARY_DIR` и `QT_BINARY_DIR`. Эти переменные создаются модулем поиска пакета Qt4. Если используется Qt5, их можно создать самостоятельно (см. пример ниже).

— В необязательном аргументе ⟨*каталог_модулей_Qt*⟩ передаётся путь к каталогу, куда будут скопированы разделяемые модули (по умолчанию в каталог приложения или специальный каталог в OS X). Этот путь будет добавлен в файл настроек `qt.conf` в каталоге приложения — так, чтобы он использовался Qt для поиска загружаемых модулей.

— Передачей значения «ложь» в качестве необязательного логического аргумента ⟨*писать_qt.conf*⟩ можно запретить команде сохранять путь к загружаемым модулям в файл настроек `qt.conf`. В этом случае приложение может, например, самостоятельно передать нужный ему путь с модулями функции `QCoreApplication::addLibraryPath()`.

— Необязательный аргумент ⟨*имя_компонента*⟩ может быть передан команде `install()` в качестве имени компонента после аргумента `COMPONENT` (п. 2.10.3).

Пример

Пусть требуется к проекту из предыдущего примера (с. 328) добавить цель установки с каталогом, имеющим структуру, представленную на рис. 3.14.

Файл `ex-qt-deploy.cpp` аналогичен CPP-файлу из предыдущего примера за исключением указания пути, по которому объект `translator` должен искать файл локализации:

```
#include <QApplication>
#include <QTranslator>
#include <QMessageBox>
```

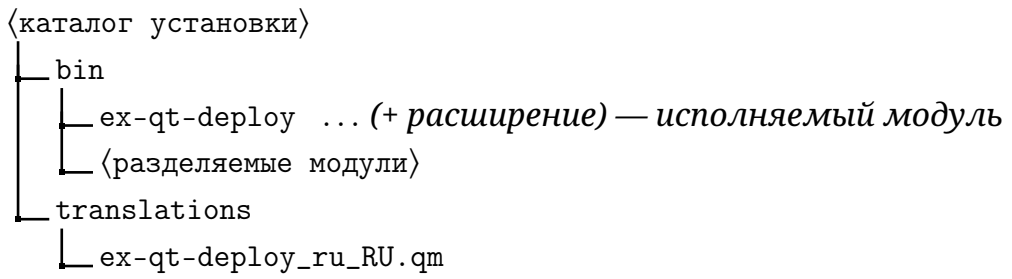

Рис. 3.14. Структура каталога установки программы, использующей библиотеки Qt

```
int main(int nArgC, char *apszArgV[])
{
  QApplication app(nArgC, apszArgV);
  //
  QTranslator translator;
  const bool cbLoaded = translator.load(
    QLocale::system(), "ex-qt-deploy_", "",
    app.applicationDirPath() + "/../translations/");
  //
  // ... (как в предыдущем примере)
}    // main()
```

Здесь функции `QTranslator::load()` последним аргументом передаётся путь к каталогу `translations` относительно каталога, где расположен сам исполняемый файл (возвращается функцией `QCoreApplication::applicationDirPath()`). Символы «/», разделяющие компоненты пути, библиотека `QtCore` при необходимости заменит на нужные символы для текущей платформы. Таким образом, каталог файлов локализации определяется независимо от текущего каталога программы.

Файл CMakeLists.txt отличается от предыдущего командами определения цели установки:

```
cmake_minimum_required(VERSION 2.8.11)

project(ex-qt-deploy)

# ... (аналогично предыдущему примеру)

# Installation

install(
  TARGETS ex-qt-deploy
  RUNTIME DESTINATION bin)

install(
  FILES ${QM_FILES}
  DESTINATION translations)

get_property(
  LIB_QT5CORE_PATH
  TARGET Qt5::Core
  PROPERTY IMPORTED_LOCATION_RELEASE)

get_filename_component(
  QT_LIBRARY_DIR "${LIB_QT5CORE_PATH}" DIRECTORY)

set(
```

```
  EXE_PATH bin/ex-qt-deploy${CMAKE_EXECUTABLE_SUFFIX})

include(DeployQt4)

install_qt4_executable(
  "${EXE_PATH}"            # исполняемый файл
  ""                       # модули
  ""                       # библиотеки
  "${QT_LIBRARY_DIR}")     # каталоги
```

Здесь вызываются команды `install()` (п. 2.10.3), которые добавляют к цели установки правила копирования исполняемого файла — результата выполнения цели `ex-qt-deploy` и скомпилированных файлов локализации (список их путей в переменной `QM_FILES` возвращается командой `qt5_create_translation()` или `qt5_add_translation()`). Дальше вызывается команда `install_qt4_executable()`, которая добавляет к цели установки правило копирования разделяемых библиотек. Ей передаётся путь к исполняемому файлу в переменной `EXE_PATH` относительно каталога установки. На момент выполнения правила исполняемый файл уже должен находиться по указанному пути благодаря командам `install()`. Имя файла состоит из имени цели (предполагается, что имя исполняемого файла не изменяется установкой свойства `OUTPUT_NAME` для цели) и зависящего от системы расширения, которое хранится в специальной переменной `CMAKE_EXECUTABLE_SUFFIX`.

Кроме относительного пути к исполняемому файлу, команде `install_qt4_executable()` передаётся путь поиска библиотечных файлов. Эта передача использована исключительно для наглядности: она необязательна, так как путь записывается в переменную с именем `QT_LIBRARY_DIR`, и команда `install_qt4_executable()` будет её также использовать для получения путей библиотек. Путь поиска библиотек Qt получается выделением каталога (команда `get_filename_component()`, п. 2.9.1) из пути к библиотеке `QtCore`, который содержится в свойстве `IMPORTED_LOCATION_RELEASE` импортируемой цели `Qt5::Core`. Следует отметить, что конфигурация «Release» предоставляется всеми сборками инструментария Qt, в отличие от конфигурации «Debug», которой, например, нет в сборках для систем на основе Linux.

Для удобства можно создать в каталоге построения следующий сценарий (`build.cmd`, на примере использования системы MinGW):

```
@echo off

set PATH=C:\Qt5.5.0\Tools\mingw492_32\bin;%PATH%
set GEN="MinGW Makefiles"
set QT_PATH=C:\Qt5.5.0\5.5\mingw492_32
set PREFIX_PATH=D:\install\ex-qt-deploy
set PROJECT_PATH=D:\Work\ex-qt-deploy

cmake^
```

```
  -G %GEN%^
  -D CMAKE_PREFIX_PATH=%QT_PATH%^
  -D CMAKE_INSTALL_PREFIX=%PREFIX_PATH%^
  %PROJECT_PATH%
```

Здесь в переменную окружения PATH добавляется путь поиска инструментов компилятора MinGW, поставляемого вместе с Qt, в переменную QT_PATH — путь к библиотекам Qt, необходимый для команды find_package() (п. 2.11.1), в переменные PREFIX_PATH и PROJECT_PATH — пути к каталогам установки и проекта. Используя этот сценарий, можно выполнить сборку и установку приложения при помощи следующих команд:

```
build.cmd
mingw32-make
mingw32-make translations
```

Дальше, если файл перевода в каталоге проекта ещё не был заполнен, необходимо сделать это при помощи инструмента Qt Linguist и повторно запустить последнюю команду:

```
mingw32-make translations
```

В конце необходимо выполнить цель установки:

```
mingw32-make install
```

Приведённый здесь код CMake способен создать цель установки приложения и разделяемых библиотек вне зависимости от того, какие именно библиотеки (Qt или какие-либо другие) используются приложением. *

3.4. Crypto++

Crypto++[12] является объектно-ориентированной и шаблонной библиотекой, реализующей популярные криптографические алгоритмы и схемы, а именно:

— схемы аутентифицированного шифрования;

— потоковые и блочные шифры, вместе с режимами их применения;

— коды аутентификации сообщений;

— хеш-функции;

— криптосистемы с открытым ключом;

— схемы обмена ключами;

— алгоритмы эллиптической криптографии;

— вспомогательные алгоритмы, включая арифметику целых чисел, многочленов и конечных полей, генерирование псевдослучайных чисел, схему разделения секрета, функции выведения ключей и т. д.

Библиотека распространяется по лицензии Boost Software License и широко используется в открытых, коммерческих и научно-образовательных проектах. Библиотека совместима с большим количеством платформ и компиляторов.

Существенным отличием библиотеки Crypto++ от рассмотренных ранее является то, что как сама библиотека не имеет

[12] http://www.cryptopp.com/ (дата обращения: 21.07.2015).

поддержки CMake, так и в CMake отсутствует модуль поиска этой библиотеки. Вместе с исходными кодами библиотеки поставляется make-файл для инструмента GNU make, который совместим с POSIX-системами и компиляторами gcc, clang, Intel C++ Compiler и т. д., а также (после правок) — с gcc-MinGW. Кроме этого, с библиотекой поставляются файлы проектов для среды Microsoft Visual Studio. При помощи этих систем поддерживаются следующие варианты сборки:

— В POSIX-совместимых системах возможна сборка в виде статической библиотеки (файл `libcryptopp.a`) или разделяемой (файл `libcryptopp.so`). Результирующий файл создаётся в каталоге исходных файлов. Цель установки `install` позволяет установить файлы библиотеки, необходимые для разработчика, в требуемый каталог, со структурой, изображённой на рис. 3.15. Кроме того, эти файлы могут быть установлены из системных репозитариев в стандартные каталоги (например, `/usr/include/crypto++` и `/usr/lib`).

— Аналогичным образом в системе Windows при помощи того же самого make-файла и инструментов gcc-MinGW можно получить такие же файлы с той разницей, что вместо разделяемой библиотеки `libcryptopp.so` будет создана динамическая `cryptopp.dll` вместе с библиотекой импорта `libcryptopp.dll.a` (рис. 3.15).

— В системе Windows при использовании среды Microsoft Visual Studio также доступны два варианта сборки библиоте-

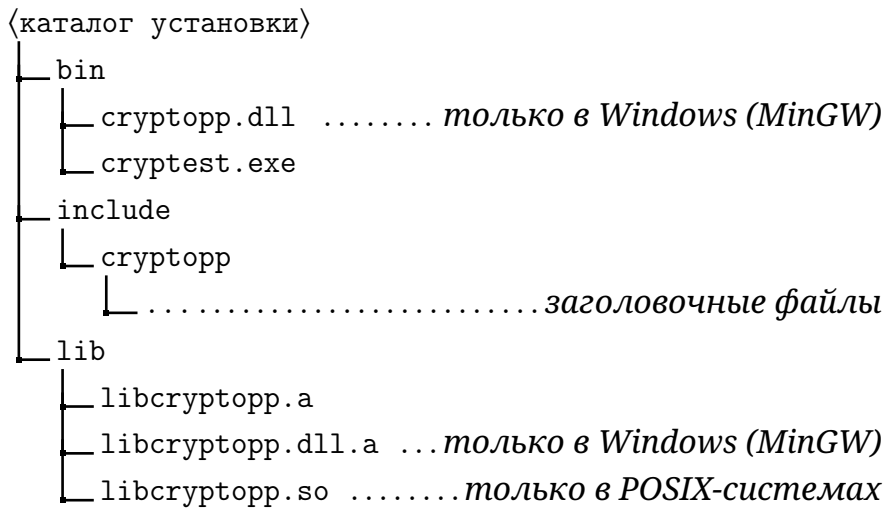

Рис. 3.15. Структура каталога с установленными файлами библиотеки Crypto++

ки. В первом вся библиотека создаётся в виде одного статически подключаемого модуля (`cryptlib.lib`). Во втором часть алгоритмов выносится в динамическую библиотеку `cryptopp.dll` с соответствующей библиотекой импорта `cryptopp.lib`. Оставшиеся алгоритмы хранятся в статической библиотеке с тем же именем `cryptlib.lib`. Динамическую библиотеку в этом случае можно использовать как совместно со статической, так и отдельно. Оба варианта сборки помещаются соответственно в подкаталоги `Output` и `DLL_Output` каталога исходных файлов библиотеки (рис. 3.16). Целью такого разделения является возможность использования двоичного файла `cryptopp.dll`, прошедшего процедуру проверки Национальным институтом стандартов и технологий США (NIST) на соответствие федеральным стандартам обработки информации FIPS 140-2.

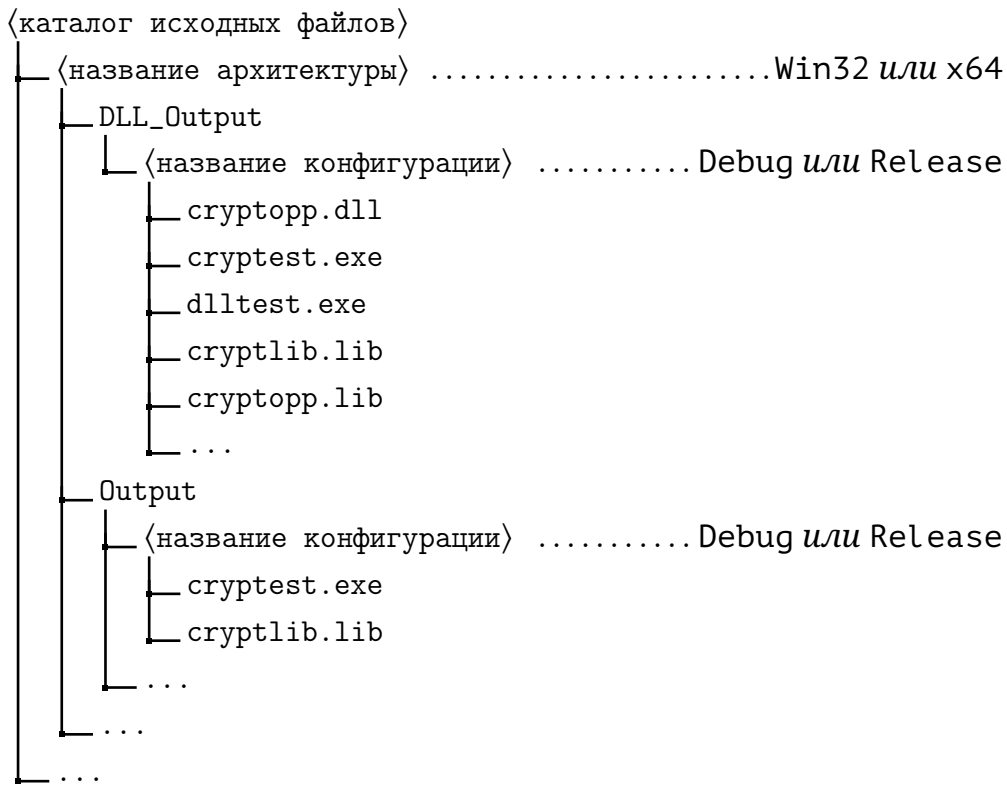

Рис. 3.16. Структура каталога с собранными файлами библиотеки Crypto++ при помощи среды Microsoft Visual Studio

В случае использования динамического варианта библиотеки, собранного при помощи Visual C++, необходимо также учитывать ещё одну связанную с этим проблему. Дело в том, что проект библиотеки для этой среды содержит настройки для её построения с ключами компилятора `/MTd` и `/MT` в режимах Debug и Release соответственно (подробнее об этих ключах см. в описании примера использования библиотеки OpenCV на с. 258). Из этого следует, что библиотека Crypto++ будет связана со статическими версиями функций диспетчера динамической памяти стандартной библиотеки поддержки выполнения программ. Это означает, что библиотека и вызывающее

её приложение будут использовать разные области динамической памяти. А это представляет собой серьёзную проблему, так как часто возникает потребность выделения области памяти в приложении с последующим её освобождением в библиотеке[13] (см. пример далее). Библиотека Crypto++ предлагает три возможных варианта решения проблемы, выполняя попытки их использования при отображении в адресное пространство работающего процесса в следующей последовательности:

1) В каждом загруженном в адресное пространство процесса исполняемом модуле ищется экспортируемая функция с именем `GetNewAndDeleteForCryptoPP()`. Если она там есть, она вызывается, возвращая адреса функций `operator new ()` и `operator delete ()` этого модуля для дальнейшего использования библиотекой. В итоге приложение и библиотека будут использовать одну и ту же область динамической памяти. Этот вариант больше подходит для случая, когда приложение использует нестандартный диспетчер памяти. Однако он неработоспособен, если приложение связано со статической версией стандартной библиотеки поддержки выполнения программ (в этом случае динамическая память инициализируется позже инициализации библиотеки).

2) Если функция из пункта 1 не экспортируется модулем, но при этом им экспортируется функция с именем `SetNewAndDeleteFromCryptoPP()`, то она вызывается,

[13] http://stackoverflow.com/questions/1634773/freeing-memory-allocated-in-a-different-dll (дата обращения: 27.07.2015).

при этом ей передаются адреса функций `operator new ()` и `operator delete ()` библиотеки Crypto++. Предполагается, что приложение сохранит эти адреса для дальнейшего использования.

3) Если функции из пунктов 1 и 2 отсутствуют во всех загруженных в процесс модулях, библиотека Crypto++ пытается найти функции `operator new ()` и `operator delete ()`, экспортируемые динамической версией стандартной библиотеки поддержки исполнения программ. Для работоспособности данного варианта требуется обеспечить загрузку этой библиотеки до библиотеки Crypto++.

Анализ приведённых возможностей решения проблемы показывает, что наиболее универсальным и простым в реализации из них является вариант 2.

Поскольку, как уже было отмечено, в системе CMake отсутствует модуль поиска библиотеки Crypto++, как и в самой библиотеке отсутствует конфигурационный файл для системы CMake, логику подключения этой библиотеки необходимо реализовывать самостоятельно. Сформулируем требования для нашей реализации:

— Поддержка должна быть реализована в виде, удобном для повторного использования.

— Для связывания цели с библиотекой должно быть достаточно подключения реализуемого модуля CMake и вызова команды `target_link_libraries()`, аналогично набору библиотек Qt.

— Реализация должна быть кроссплатформенной.

— Реализация должна иметь возможность подключать библиотеку, как собранную при помощи make-файла, так и при помощи среды Visual Studio.

— Реализация должна уметь находить файлы библиотеки, как расположенные непосредственно в каталоге исходных файлов после построения, так и установленные целью `install` или из стандартных репозитариев системы. Каталоги поиска файлов должны зависеть от значений специальных переменных `CMAKE_PREFIX_PATH`, `CMAKE_LIBRARY_PATH` и других, влияющих на поведение команд `find_library()` и т. д. (п. 2.9.2).

— Реализация должна выбирать статический или динамический/разделяемый вариант сборки библиотеки в зависимости от значения специальной переменной `BUILD_SHARED_LIBS`.

— При использовании динамической версии библиотеки, собранной в среде Visual Studio, к проекту должен автоматически подключаться исходный модуль для поддержки совместного использования приложением и библиотекой общей области динамической памяти, реализованный в варианте 2.

— Реализация должна выбирать правильную версию библиотеки для разных конфигураций построения проекта.

Пример

Пусть требуется зашифровать тестовое сообщение, после чего результат расшифровать и вывести на печать. В качестве метода шифрования необходимо использовать блочный алгоритм AES в режиме счётчика со случайными вектором инициализации и ключом размера 128 бит. Реализация построения проекта должна удовлетворять перечисленным выше требованиям. Решением этой задачи является код, расположенный в следующих файлах.

Файл `ex-cryptopp.cpp`:

```
#include "aes.h"
#include "ccm.h"
#include "osrng.h"
#include "filters.h"
#include "secblock.h"

#include <iostream>
#include <string>

typedef CryptoPP::CTR_Mode <CryptoPP::AES> AESWithCTR;

const char g_acszPlainText[] =
  "0123456789ABCDEF Hello World! 0123456789ABCDEF";

template <class TCryptor>
  std::string crypt(
    const CryptoPP::SecByteBlock &rcKey,
```

```
    const CryptoPP::SecByteBlock &rcIV,
    const std::string &rcInput)
{
  TCryptor cryptor;
  cryptor.SetKeyWithIV(
    rcKey.data(), rcKey.size(), rcIV.data());
  std::string s_result;
  CryptoPP::StringSource source(
    rcInput, true,
    new CryptoPP::StreamTransformationFilter(
      cryptor, new CryptoPP::StringSink(s_result)));
  //
  return s_result;
}

int main()
{
  CryptoPP::SecByteBlock key(
    CryptoPP::AES::DEFAULT_KEYLENGTH);
  CryptoPP::SecByteBlock iv(
    CryptoPP::AES::BLOCKSIZE);
  CryptoPP::AutoSeededRandomPool generator;
  generator.GenerateBlock(key, key.size());
  generator.GenerateBlock(iv, iv.size());
  //
  std::string s_encrypted =
    crypt <AESWithCTR::Encryption> (
```

```
      key, iv, g_acszPlainText);
  std::string s_decrypted =
    crypt <AESWithCTR::Decryption> (
      key, iv, s_encrypted);
  //
  std::cout << s_decrypted << std::endl;
}
```

Здесь в первой строке после подключения заголовочных файлов выполняется конкретизация библиотечного шаблона `CTR_Mode <>`, реализующего схему со счётчиком, классом, реализующим интерфейс симметричного шифра (класс `AES`), который используется схемой в процессе шифрования блоков. Внутри шаблона `CTR_Mode <>` определены типы `Encryption` и `Decryption`, реализующие интерфейс `SymmetricCipher` и предназначенные соответственно для шифрования и дешифрования. Эти классы используются в дальнейшем в функции `main()`.

Дальше приводится объявление константной строки `g_acszPlainText`, которая представляет собой исходное сообщение. Длина строки выбрана таким образом, что она, с одной стороны, больше длины блока алгоритма AES (16 байт), а с другой — не кратна ему. Это необходимо для проверки корректности работы схемы `CTR_Mode <>`.

Следующим идёт определение шаблонной функции `crypt <> ()`, которая предназначена одновременно для шифрования и дешифрования — в зависимости от передаваемого

ей шаблонного аргумента (типы `Encryption` или `Decryption`, о которых шла речь выше). Параметрами функции являются ключ, вектор инициализации и сообщение, которое требуется обработать. Первые два параметра имеют тип «`const SecByteBlock &`», который отвечает за автоматическое обнуление памяти после её использования в целях безопасности. В реальных приложениях исходное сообщение нужно передавать таким же способом.

В теле функции `crypt <> ()` объявляется переменная типа `Encryption` или `Decryption`, которой далее передаются ключ и вектор инициализации. Для ключа также передаётся его размер, так как один и тот же блочный шифр может работать с ключами нескольких размеров (как в случае с AES).

Далее в функции `crypt <> ()` объявляется переменная типа `StringSource`, задача которой состоит в передаче данных из источника на вход связанного с ней алгоритма преобразования. Класс `StringSource` является реализацией абстрактного класса `Source` (источник данных в последовательности преобразований), которая отвечает за чтение данных из строки (`std::string`) или массива символов. Другие реализации могут считывать данные из файла, по сети и т. д. Класс `Source`, в свою очередь, является реализацией абстрактного класса `Filter`, производного от интерфейса `BufferedTransformation`. Последний обобщает понятие преобразователя потока данных, которые считываются, возможно, порциями, и результат преобразования которых накапливается во внутреннем буфере. Конструктору переменной типа `StringSource` передают-

ся строка-источник (сообщение для шифрования или расшифрования), параметр `pumpAll` (`true`), указывающий на то, что в конструкторе необходимо считать столько данных из источника, сколько возможно, а также прикрепляемый объект преобразования следующего этапа (тоже производный от `BufferedTransformation`). В данном случае это объект типа `StreamTransformationFilter`, который является адаптером производных классов от `StreamTransformation`, реализуя интерфейс `Filter`. Конструктору объекта передаются адаптируемый объект (переменная `cryptor`) и прикрепляемый объект преобразования следующего этапа. В качестве него используется объект типа `StringSink`, который является реализацией абстрактного класса `Sink` (последний этап цепочки преобразований, «сток»), также производного от `BufferedTransformation`. Класс `StringSink` отвечает за добавление приходящих ему на вход данных в конец заданной строки. В рассматриваемом примере эта строка в итоге возвращается функцией `crypt <> ()`.

Следует отметить, что в приведённом здесь исходном коде объекты типов `StreamTransformationFilter` и `StringSink` создаются в динамической памяти внутри приложения, а удаляются владеющими ими объектами внутри библиотеки Crypto++. Это является примером описанной выше проблемы использования разных куч в приложении и динамической библиотеке Crypto++, собранной при помощи Visual C++.

Работа функции `main()` заключается в заполнении случайными данными ключа и вектора инициализации, вызова функ-

ции `crypt <> ()`, конкретизированной алгоритмом шифрования, с исходным сообщением, затем вызовом её же, конкретизированной алгоритмом расшифрования, с результатом её предыдущего вызова, и выводе на печать результата последнего вызова.

Файл `cryptopp-dll.cpp`, предназначенный для исправления проблемы различных куч в приложении и динамической библиотеке Crypto++, собранной при помощи Visual C++:

```
#ifdef _MSC_VER

#include <new>

typedef void * (*PNew)(size_t);
typedef void (*PDelete)(void *);
typedef std::new_handler (*PSetNewHandler)(
  std::new_handler);

static PNew g_pNew = nullptr;
static PDelete g_pDelete = nullptr;

extern "C" __declspec (dllexport)
  void SetNewAndDeleteFromCryptoPP(
    PNew pNew, PDelete pDelete,
    PSetNewHandler pSetNewHandler)
{
  g_pNew = pNew;
  g_pDelete = pDelete;
```

```
}

void *operator new (size_t size)
{
  return g_pNew(size);
}

void operator delete (void *p)
{
  g_pDelete(p);
}

#endif    // _MSC_VER
```

Здесь в начале и конце файла находится пара директив условного включения содержимого, если определён макрос `_MSC_VER`, предопределённый в компиляторе Visual C++. В файле определяются типы указателей на функции, совместимые со стандартными `operator new ()`, `operator delete ()` и `std::set_new_handler()`. Эти объявления скопированы из заголовочного файла `dll.h` библиотеки Crypto++. Сам файл не подключается из кода примера, так как он содержит директивы компилятора Visual C++ для связывания с библиотекой `cryptopp.dll`, а эта задача возлагается на систему CMake (которая связывает приложение с нужной версией библиотеки по сложным правилам).

Далее определена экспортируемая исполняемым модулем функция `SetNewAndDeleteFromCryptoPP()`, которая будет вызвана библиотекой `cryptopp.dll` при первой попытке обращения к динамической памяти. Функция сохраняет в глобальных переменных передаваемые ей указатели на `operator new ()` и `operator delete ()` из библиотеки. Переопределяемые дальше функции `operator new ()` и `operator delete ()` для приложения вызывают свои аналоги из библиотеки по ранее сохранённым указателям. Объявление `extern "C"` в заголовке функции предотвращает добавление информации о типах её параметров к её имени в таблице экспорта модуля (что делается для функций в языке C++, где разрешена перегрузка).

Файл `cryptopp.cmake`:

```
set(OUT_SUBDIR Output)
set(LIB_NAMES cryptopp)
set(ADD_DLL_IMPORT FALSE)
set(ADD_DLL_MINGW_IMPORT FALSE)

if(MSVC)
  set(LIB_NAMES cryptlib)
  #
  if(BUILD_SHARED_LIBS)
    set(OUT_SUBDIR DLL_Output)
    list(APPEND LIB_NAMES cryptopp)
    set(ADD_DLL_IMPORT TRUE)
    #
    add_library(CryptLib STATIC IMPORTED)
```

```
  endif()
elseif(MINGW OR CYGWIN)
  if(BUILD_SHARED_LIBS)
    set(LIB_NAMES libcryptopp.dll.a)
    set(ADD_DLL_MINGW_IMPORT TRUE)
  else()
    set(LIB_NAMES libcryptopp.a)
  endif()
elseif(UNIX)
  if(BUILD_SHARED_LIBS)
    set(LIB_NAMES libcryptopp.so)
  else()
    set(LIB_NAMES libcryptopp.a)
  endif()
endif()

add_library(CryptoPP STATIC IMPORTED)

if(ADD_DLL_IMPORT)
  get_filename_component(
    CRYPT_SRC cryptopp-dll.cpp ABSOLUTE)
  set_property(
    TARGET CryptoPP
    APPEND
    PROPERTY INTERFACE_SOURCES
    "${CRYPT_SRC}")
  #
```

```
  set_property(
    TARGET CryptoPP
    APPEND
    PROPERTY INTERFACE_COMPILE_DEFINITIONS
    CRYPTOPP_IMPORTS)
endif()

if(ADD_DLL_MINGW_IMPORT)
  set_property(
    TARGET CryptoPP
    APPEND
    PROPERTY INTERFACE_COMPILE_DEFINITIONS
    CRYPTOPP_MINGW_IMPORTS)
endif()

find_path(
  CRYPT_H_DIR aes.h
  PATH_SUFFIXES cryptopp)

if(NOT CRYPT_H_DIR)
  message(
    FATAL_ERROR "Could not find aes.h")
endif()

foreach(CONFIG Debug Release)
  set(NUM_LIB 0)
  foreach(LIB IN LISTS LIB_NAMES)
```

```
    math(EXPR NUM_LIB "${NUM_LIB} + 1")
    set(VAR_NAME LIB_${NUM_LIB}_${CONFIG})
    find_library(
      ${VAR_NAME}
      NAMES ${LIB}
      PATH_SUFFIXES "${OUT_SUBDIR}/${CONFIG}")
    #
    if(${VAR_NAME})
      message(
        STATUS
        "Using a library ${${VAR_NAME}} "
        "for config ${CONFIG}")
      set(LIB_${LIB}_${CONFIG} "${${VAR_NAME}}")
    else()
      message(
        WARNING
        "Could not find ${LIB} "
        "for config ${CONFIG}")
    endif()
  endforeach()
endforeach()

get_property(
  DEBUG_CONFIGS
  GLOBAL
  PROPERTY DEBUG_CONFIGURATIONS)
```

```
foreach(CONFIG IN LISTS CMAKE_CONFIGURATION_TYPES
  ITEMS "")
  list(FIND DEBUG_CONFIGS "${CONFIG}" DEBUG_CFG_NUM)
  if(CONFIG STREQUAL Debug OR
    NOT DEBUG_CFG_NUM EQUAL -1)
    set(CONFIG_NAME Debug)
  else()
    set(CONFIG_NAME Release)
  endif()
  #
  string(TOUPPER "${CONFIG}" CONFIG)
  if(CONFIG)
    set(PROPERTY_NAME IMPORTED_LOCATION_${CONFIG})
  else()
    set(PROPERTY_NAME IMPORTED_LOCATION)
  endif()
  #
  set(LIB_CRYPTOPP_NAME LIB_1_${CONFIG_NAME})
  set(LIB_CRYPTLIB_NAME LIB_2_${CONFIG_NAME})
  #
  if(NOT ${LIB_CRYPTOPP_NAME})
    set(${LIB_CRYPTOPP_NAME} ${${LIB_CRYPTLIB_NAME}})
  endif()
  #
  set_property(
    TARGET CryptoPP
    PROPERTY ${PROPERTY_NAME}
```

```
    ${${LIB_CRYPTOPP_NAME}})
  #
  if(TARGET CryptLib)
    set_property(
      TARGET CryptLib
      PROPERTY ${PROPERTY_NAME}
      ${${LIB_CRYPTLIB_NAME}})
  endif()
endforeach()

set_property(
  TARGET CryptoPP
  PROPERTY INTERFACE_INCLUDE_DIRECTORIES
  "${CRYPT_H_DIR}")

if(TARGET CryptLib)
  set_property(
    TARGET CryptoPP
    PROPERTY INTERFACE_LINK_LIBRARIES
    CryptLib)
endif()

include(correct_vc_static.cmake)
```

Здесь вначале происходит заполнение вспомогательных переменных в зависимости от используемой конечной системы построения и варианта сборки библиотеки Crypto++. Пе-

ременная OUT_SUBDIR содержит имя подкаталога для поиска библиотек компилятором Visual C++. В зависимости от использования статических или динамических версий библиотек (управляется переменной BUILD_SHARED_LIBS) этой переменной устанавливается значение «Output» или «DLL_Output» (см. также рис. 3.16). Переменной LIB_NAMES присваивается список имён библиотек, поиск которых в дальнейшем выполняется командой find_library() (п. 2.9.2). В общем случае это имя «cryptopp». В случае использования Visual C++ это «cryptlib», а также, если используется динамическая библиотека, «cryptopp». При использовании POSIX-совместимых систем базовые имена статической и разделяемой версии библиотеки одинаковые. Поэтому чтобы команда find_library() нашла требуемую версию в соответствии со значением переменной BUILD_SHARED_LIBS, ей необходимо передать имя библиотеки полностью. Для статической версии это «libcryptopp.a», для библиотеки импорта динамической библиотеки при использовании MinGW/cygwin — «libcryptopp.dll.a», для разделяемой библиотеки в остальных системах — «libcryptopp.so».

Также вначале заполняются вспомогательные логические переменные, которые используются дальше: ADD_DLL_IMPORT устанавливается в истину, если используется динамическая библиотека в Visual C++, и ADD_DLL_MINGW_IMPORT, если динамическая библиотека в MinGW. Кроме этого, создаются цели «CryptoPP» и при необходимости — «CryptLib» для подключаемых библиотек при помощи команды add_library(... IMPORTED) (п. 2.5.2).

В оставшейся части модуля `cryptopp.cmake` выполняется поиск необходимых файлов и установка свойств созданных целей библиотек. Если используется динамическая библиотека в Visual C++ (переменная `ADD_DLL_IMPORT` содержит значение истины), к использующему её проекту необходимо добавить исходный модуль `cryptopp-dll.cpp`, а также к настройкам компилятора — определение символа препроцессора `CRYPTOPP_IMPORTS`. Последнее необходимо для того, чтобы в заголовочных файлах библиотеки Crypto++ объявления были адаптированы для динамической библиотеки (например, к функциям и классам была добавлена спецификация `__declspec (dllimport)`). Символ `CRYPTOPP_IMPORTS` определяется в заголовочном файле `dll.h`, однако, как уже отмечалось, данный пример его не использует, поэтому этот символ необходимо определить в настройках компилятора. Обе эти задачи реализуются установкой свойств цели `CryptoPP`, соответственно `INTERFACE_SOURCES` (файлы исходных модулей для зависимых целей) и `INTERFACE_COMPILE_DEFINITIONS` (определения символов препроцессора для зависимых целей). Для обычных целей эти свойства можно настроить гораздо проще при помощи соответственно команд `target_sources()` и `target_compile_definitions()` (п. 2.6.4) с аргументом `INTERFACE`. Однако цель `CryptoPP` является импортированной, из-за чего эти команды к ней неприменимы (см. п. 2.5.2). Остаётся отметить, что исходный файл `cryptopp-dll.cpp` нужно добавлять к проекту по его абсолютному пути, иначе он не будет найден системой CMake. Для получения этого пути ис-

пользуется команда `get_filename_component(... ABSOLUTE)` (п. 2.9.1).

Аналогичным образом к настройкам проекта добавляется определение символа препроцессора `CRYPTOPP_MINGW_IMPORTS` при использовании динамической библиотеки Crypto++ в MinGW/cygwin. Это можно применить для обеспечения корректности работы данной библиотеки (см. замечание после примера).

Далее при помощи команды `find_path()` (п. 2.9.2) выполняется поиск заголовочного файла `aes.h` из состава библиотеки Crypto++ (остальные её заголовочные файлы будут расположены в том же каталоге). Каталог, в котором располагается файл, будет записан в переменную `CRYPT_H_DIR`. На работу команды (каталоги, в которых будет выполняться поиск) влияют значения различных специальных переменных CMake, таких как `CMAKE_INCLUDE_PATH`. После аргумента `PATH_SUFFIXES` (допнительные подкаталоги для поиска) указано имя подкаталога `cryptopp`, так как цель установки make-файла библиотеки Crypto++ устанавливает заголовочные файлы в подкаталог `include/cryptopp` (см. рис. 3.15).

Следующий цикл `foreach()` выполняет своё тело дважды, обходя имена конфигураций «`Debug`» и «`Release`». Вложенный в него цикл обходит все имена библиотек, сохранённые в списке `LIB_NAMES` (см. выше), которые необходимо найти. В теле цикла формируется имя очередной переменной для хранения пути к файлу библиотеки: `LIB_1_Debug`, `LIB_2_Debug` и т. д. Это имя передаётся команде `find_library()` (п. 2.9.2)

для записи в соответствующую переменную найденного пути. При помощи аргумента `PATH_SUFFIXES` команде передаётся дополнительный подкаталог для поиска файла библиотеки: [`DLL_`]`Output/`⟨*имя_конфигурации*⟩. В нём будет находиться библиотека в случае использования компилятора Visual C++. Путь к каталогу, содержащему этот каталог, с требуемой версией библиотеки (`Win32` или `x64`), нужно передать инструменту CMake, например, через переменную `CMAKE_LIBRARY_PATH` (см. далее). Для других компиляторов файл библиотеки будет находиться непосредственно в переданном инструменту каталоге, он будет сохранён в этом фрагменте кода CMake в переменные, соответствующие обеим конфигурациям.

Далее выполняется опрос значения глобального свойства `DEBUG_CONFIGURATIONS` (имена отладочных конфигураций помимо «`Debug`»). Оно используется в следующем цикле по всем конфигурациям, обрабатываемым CMake (список в переменной `CMAKE_CONFIGURATION_TYPES`), а также конфигурации с пустым именем (используется по умолчанию для генераторов с фиксированной конфигурацией, см. п. 2.12.1). В начале тела цикла проверяется, является ли конфигурация отладочной (имя совпадает с «`Debug`» или принадлежит списку из свойства `DEBUG_CONFIGURATIONS`). В зависимости от этого выбираются версии файлов библиотек «`Debug`» или «`Release`», найденные в предыдущем цикле (пути в переменных `LIB_1_Debug`/`LIB_1_Release` и т. д.). Пути к этим библиотекам устанавливаются в качестве значений свойств `IMPORTED_LOCATION` (для конфигурации с пустым именем) или `IMPORTED_LOCATION_`⟨*имя*⟩

(для всех остальных) целей «CryptoPP» и «CryptLib» (если она ранее была создана).

В конце устанавливаются свойства цели CryptoPP для облегчения её использования с целями пользовательских приложений. Свойство INTERFACE_INCLUDE_DIRECTORIES (используемые в зависимых целях пути поиска заголовочных файлов) устанавливается в путь к ранее найденному каталогу заголовочных файлов библиотеки Crypto++. Таким образом, его добавление к зависимым целям будет выполняться автоматически. Если ранее была определена цель CryptLib, для цели CryptoPP также устанавливается свойство INTERFACE_LINK_LIBRARIES, которое определяет дополнительные подключаемые библиотеки к зависимым целям, именем цели CryptLib. В результате зависимые цели достаточно будет связать только с целью CryptoPP, цель CryptLib при необходимости будет подключена автоматически.

Последней выполняется настройка параметров компиляции для инструментов Visual C++ при помощи подключения модуля correct_vc_static.cmake. Его содержимое было приведено и подробно рассмотрено в примере использования библиотек OpenCV на с. 258.

Файл CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8)

project(ex-cryptopp)

include(cryptopp.cmake)
```

```
add_executable(ex-cryptopp ex-cryptopp.cpp)
target_link_libraries(ex-cryptopp CryptoPP)
```

Здесь остаётся только подключить файл `cryptopp.cmake` и связать цель приложения с целью библиотеки `CryptoPP`.

Сценарий для запуска инструмента CMake в системе Windows при использовании компилятора Visual C++ может быть следующим:

```
set CRYPTO_H_DIR=D:\Tools\cryptopp562
set CRYPTO_LIB_DIR=%CRYPTO_H_DIR%\Win32

cmake^
  -G "Visual Studio 12 2013"^
  -D CMAKE_INCLUDE_PATH=%CRYPTO_H_DIR%^
  -D CMAKE_LIBRARY_PATH=%CRYPTO_LIB_DIR%^
  -D BUILD_SHARED_LIBS=ON^
  D:\Work\ex-cryptopp
```

Здесь переменной окружения `CRYPTO_H_DIR` присваивается путь к каталогу, в котором находятся исходные коды библиотеки, в частности её заголовочные файлы. В команде вызова CMake значение этой переменной присваивается переменной CMake `CMAKE_INCLUDE_PATH`. Эта настройка влияет на поиск файлов командой `find_file()`. Аналогичным образом переменной окружения `CRYPTO_LIB_DIR` присваивается путь к каталогу, в котором находятся собранные версии библиотеки для дан-

ной платформы (Win32/x64) и компилятора. Это значение влияет на поведение команды `find_library()`.

В случае использования компилятора gcc-MinGW совместно с библиотекой Crypto++, для которой выполнена процедура установки в отдельный каталог при помощи команды `mingw32-make install`, команда вызова инструмента CMake может устанавливать значение переменной `CMAKE_PREFIX_PATH` путём к каталогу установки библиотеки вместо переменных `CMAKE_INCLUDE_PATH` и `CMAKE_LIBRARY_PATH`.

Аналогичным образом выглядит команда запуска инструмента CMake в системах Linux:

```
cmake \
  -D CMAKE_PREFIX_PATH=$HOME/install/cryptopp562 \
  -D BUILD_SHARED_LIBS=ON \
  $HOME/work/ex-cryptopp
```

Если требуется подключить библиотеку Crypto++, установленную в систему из стандартных репозитариев, установку переменной `CMAKE_PREFIX_PATH` в командной строке инструмента CMake можно не выполнять. Как и в случае системы Windows, при необходимости переменной `BUILD_SHARED_LIBS` можно присвоить значение `OFF`. *

Замечание: попытка построения приведённого примера с использованием динамической библиотеки `cryptopp.dll` в системе gcc-MinGW приведёт к ошибкам времени сборки. Это вызвано ошибками в текущей версии библиотеки Crypto++: в объявлении класса `CryptoPP::StringStore` пропущено описание

CRYPTOPP_DLL, а также использованы объявления внешних конкретизаций шаблонов, например:

```
extern template class AllocatorWithCleanup <byte>;
```

без соответствующих им определений явных конкретизаций:

```
template class CryptoPP::AllocatorWithCleanup <byte>;
```

Чтобы обойти эти ошибки, нужно добавить в исходный код примера указанные определения, а также скопировать туда же определения методов класса StringStore из кода библиотеки.

▲

3.5. Инструменты разработки

В заключение главы рассмотрим несколько модулей CMake, которые, в отличие от предыдущих, предназначены для поиска не библиотек, а инструментов, часто используемых в процессе разработки.

3.5.1. Управление версиями

Одним из видов инструментов, которые используются коллективами разработчиков, являются клиенты систем управления версиями. В состав CMake входят модули поиска для наиболее распространённых из них: Concurrent Versions System (CVS), Git, Mercurial и Subversion (SVN). Рассмотрим для примера организацию поддержки работы с Git, с остальными системами можно работать аналогично.

Интерфейс модуля поиска Git является очень простым, достаточно вызвать команду:

```
find_package(Git)
```

После этого результаты работы модуля становятся доступными при помощи трёх переменных (табл. 3.7).

Таблица 3.7

Переменные, заполняемые модулем поиска Git

Переменная	Значение
GIT_FOUND	Истина, если исполняемый файл найден
GIT_EXECUTABLE	Путь к программе
GIT_VERSION_STRING	Версия найденной программы

Замечание: для поиска исполняемого файла модуль использует команду find_program() (п. 2.9.2), которая создаёт переменную GIT_EXECUTABLE в кэше. При последующих запусках CMake со старым файлом кэша результат поиска загружается из него. Таким образом, генерирование проекта ускоряется. Нет необходимости самостоятельно организовывать хранение переменной GIT_EXECUTABLE в кэше. ▲

При взаимодействии с удалённым хранилищем в системе Git разработчику приходится вручную набирать много команд в консоли, особенно при разрешении конфликтов версий. Трудно предложить какую-либо автоматизированную реализацию

этого процесса. Однако есть некоторые наиболее часто используемые последовательности одинаковых команд, для которых можно предусмотреть правила в системе построения. С этой целью и можно использовать CMake.

Пример

Пусть требуется реализовать команды загрузки и выгрузки проекта из репозитария, подготовки пакета изменений и опроса состояния, оформив их в виде фальшивых целей.

Для удобства повторного использования весь код, решающий эту задачу, вынесен в модуль `git.cmake`:

```
find_package(Git)

if(NOT GIT_FOUND)
  message(
    WARNING
    "Could not find Git on this system. You can "
    "manually assign a path to it to "
    "GIT_EXECUTABLE variable.")
  return()
endif()

add_custom_target(
  git_pull
  COMMAND "${GIT_EXECUTABLE}" pull
  WORKING_DIRECTORY "${CMAKE_SOURCE_DIR}")
```

```
add_custom_target(
  git_commit
  COMMAND "${GIT_EXECUTABLE}" add *
  COMMAND "${GIT_EXECUTABLE}" commit
  COMMAND "${GIT_EXECUTABLE}" push ${GIT_PUSH_ARGS}
  WORKING_DIRECTORY "${CMAKE_SOURCE_DIR}")

add_custom_target(
  git_push
  COMMAND "${GIT_EXECUTABLE}" push ${GIT_PUSH_ARGS}
  WORKING_DIRECTORY "${CMAKE_SOURCE_DIR}")

add_custom_target(
  git_status
  COMMAND "${GIT_EXECUTABLE}" status
  WORKING_DIRECTORY "${CMAKE_SOURCE_DIR}")
```

Здесь вначале выполняется проверка успешности поиска Git. Если это не так, цели не создаются. Иначе создаётся фальшивая цель `git_pull`, которая исполняет команду «`git pull`» в корневом каталоге проекта. По этой команде выполняется загрузка последних версий файлов из удалённого репозитария. Аналогично, цель `git_commit` сначала исполняет команду «`git add *`», которая добавляет новые и изменившиеся файлы в каталоге проекта к будущему набору изменений («commit»). Далее исполняется команда «`git commit`», которая формирует очередной набор изменений из добавленных файлов.

Здесь система Git запускает текстовый редактор, чтобы пользователь мог ввести комментарий к изменениям. Последняя команда «`git push`» фиксирует ранее сформированные наборы изменений в репозитарий в виде транзакции. Часто этой команде требуется передача дополнительных аргументов, которые можно присвоить переменной `GIT_PUSH_ARGS`. Далее, цель `git_push` исполняет только последнюю команду. Это может быть необходимо, если после всех изменений и исполнения цели `git_commit` соединение с удалённым хранилищем по каким-то причинам не удалось и необходимо повторно выполнить команду «`git push`». Наконец, цель `git_status` исполняет команду «`git status`», которая выводит состояние локальной копии хранилища (списки новых и изменившихся файлов, информация о том, есть ли локальные фиксации, которые не были отправлены в удалённый репозитарий и т. д.).

Пример файла `CMakeLists.txt` с подключением модуля:

```
cmake_minimum_required(VERSION 2.8)

project(ex-git)

add_executable(ex-git ex-git.cpp)

# При необходимости добавить параметры для git push,
# например:
# set(GIT_PUSH_ARGS origin HEAD:refs/for/master)

include(git.cmake)
```

Замечание: при использовании в приведённом примере среды разработки, например Microsoft Visual Studio, попытка со стороны клиента Git запроса у пользователя пароля для доступа к репозитарию по протоколу HTTPS может привести к «подвисанию» процесса построения целей. Это происходит из-за того, что у инструментов, запускаемых средой, нет доступа к консоли. Чтобы решить эту проблему, можно организовать доступ к репозитарию по протоколу SSH (сервер хранилища должен его поддерживать) с аутентификацией по открытому ключу. Для этого в системе Windows можно воспользоваться утилитами из набора PuTTY[14]:

1) Убедиться, что переменная окружения `GIT_SSH` содержит полный путь к команде `plink`, например:

```
c:\Program Files\Putty\PLINK.EXE
```

   При необходимости можно изменить значение этой переменной через Панель управления Windows.

2) Воспользоваться программой PuttyGen для генерации пары ключей шифрования по алгоритму SSH2-RSA.

3) Загрузить на сервер хранилища открытый ключ, сгенерированный на шаге 2 (скопировать текст из окна PuttyGen). Если сервер работает на системе Linux, это можно сделать, добавив текст ключа в конец файла `.ssh/authorized_keys` в домашнем каталоге пользователя, от имени которого нужно получить доступ к хранили-

[14] http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html (дата обращения: 17.03.2015).

щу. Для этого можно зайти на сервер по протоколу SSH при помощи программы PuTTY, если такая возможность есть, или попросить администратора самому отредактировать файл. Если хранилище интегрировано с системой управления проектами, можно воспользоваться её Web-интерфейсом. Эта функция должна быть доступна в настройках пользователя («SSH keys» в BitBucket, «SSH Public Keys» в Gerrit и т. д.).

4) Запустить программу `pageant`, передав ей в командной строке путь к созданному на шаге 2 закрытому ключу («`.ppk`»):

```
"%ProgramFiles%\PuTTY\pageant.exe" ⟨путь_к_.ppk⟩
```

Этот пункт нужно повторять каждый раз перед началом работы с проектом. Для удобства можно создать ярлык с этой командой.

5) При необходимости переключить доступ к репозитарию с протокола HTTPS на SSH, исполнив в каталоге проекта команду:

```
git remote set-url origin ⟨URL_для_репозитария_по_SSH⟩
```

6) При попытке доступа к хранилищу может выводиться сообщение об ошибке, связанное с тем, что удалённый сервер неизвестен локальной системе:

```
The server's host key is not cached in the registry. You
have no guarantee that the server is the computer you
```

```
think it is.
The server's rsa2 key fingerprint is:
...
```

В этом случае можно один раз зайти на сервер по протоколу SSH при помощи программы PuTTY, указав ей URL для доступа к репозитарию (см. шаг 5).

В Linux процедура создания ключей будет следующей:

1) Вызвать в командной строке утилиту `ssh-keygen`:

```
ssh-keygen -t rsa
```

Программа запросит путь к файлу закрытого ключа. Нужно ввести пустую строку, тогда ключ запишется в файл `.ssh/id_rsa` в домашнем каталоге. Затем программа запросит пароль и его подтверждение. Открытый ключ будет записан в файл `.ssh/id_rsa.pub`

2) Загрузить открытый ключ на сервер хранилища, как в случае Windows. ▲

3.5.2. Генерирование документации

Ещё одной задачей, которая часто возникает перед разработчиком, является поддержка документации к коду, генерируемой на основе специальных комментариев. В CMake есть модуль поиска для системы Doxygen[15]. В её состав входит консольный генератор `doxygen`, который получает на вход текстовый

[15] http://www.stack.nl/~dimitri/doxygen/ (дата обращения: 18.03.2015).

файл настроек (по умолчанию с именем `Doxyfile`) и создаёт документацию в нескольких форматах. Входной файл, как правило, находится в каталоге проекта и может редактироваться либо в текстовом редакторе, либо программой с графическим интерфейсом Doxywizard, также входящей в состав Doxygen. Выходные форматы включают HTML, LaTeX, RTF и т. д.

В процессе работы инструмент Doxygen может использовать следующие дополнительные инструменты:

— Программу dot, входящую в состав пакета GraphViz[16] (ПО и библиотеки для визуализации графов). Doxygen может быть настроен на добавление в документацию различных диаграмм (классов, взаимодействия, подключения заголовочных файлов и т. д.), которые генерируются вызываемым им dot в виде графических изображений.

— Программу HTML Help Compiler (`hhc`) из состава HTML Help Workshop[17], которая генерирует файл справки в формате Compressed HTML (`.chm`) из набора файлов HTML, изображений и т. д. Формат CHM используется в системе Windows. Doxygen вызывает эту программу после генерирования документации в HTML для создания её компактной версии.

— Аналогично, Doxygen может создавать компактную форму документации из HTML при помощи программы Qt Help Generator (`qhelpgenerator`) из состава Qt. Получаемые таким образом файлы в формате Qt Compressed Help (`.qch`)

[16] `http://www.graphviz.org/` (дата обращения: 24.03.2015).

[17] `http://www.microsoft.com/en-us/download/details.aspx?id=21138` (дата обращения: 24.03.2015).

можно далее помещать в наборы справки Qt Help Collection (`.qhc`) при помощи инструмента Qt Collection Generator (`qcollectiongenerator`).

— Компилятор документов LaTeX и генератор предметных указателей makeindex, входящие в состав системы вёрстки документов TeX[18]. Doxygen вызывает инструмент `latex` для генерирования изображений формул, которые могут отображаться в HTML-документации. Кроме этого, в выходном каталоге создаётся make-файл (и сценарий `make.bat` в Windows) с правилами вызова этих инструментов для построения документации из генерируемых файлов LaTeX. Этот сценарий может исполняться пользователем вручную, Doxygen непосредственно его не вызывает. Если в файле его настроек включена опция генерирования документации в формате PDF, вместо инструмента LaTeX в сценарий записываются правила вызова pdfLaTeX. Также в сценарии может быть использован вызов инструмента BibTeX для генерирования библиографических ссылок.

— Интерпретатор сценариев на языке Perl[19] для реализации связи с внешней документацией.

Основной проблемой при использовании системы Doxygen является то, что пути ко всем исполняемым файлам перечисленных инструментов (кроме pdfLaTeX и BibTeX) хранятся в файле настроек `Doxyfile`, это делает его привязанным к конкретной системе и затрудняет поддержку кроссплатформенности. В ка-

[18] `https://www.ctan.org/` (дата обращения: 24.03.2015).
[19] `https://www.perl.org/` (дата обращения: 24.03.2015).

честве выхода можно указывать в настройках только имена файлов, перечисляя пути к ним в переменной окружения `PATH` в системе, которая используется для разработки. Такой подход не работает в случае использования Perl, для которого нужно указывать полный путь. С другой стороны, использовать Perl совместно с Doxygen приходится нечасто.

Другая похожая проблема заключается в том, что относительные пути ко входным файлам и каталогам, как и выходной каталог для системы Doxygen, интерпретируются по отношению к рабочему каталогу, в котором запускается инструмент. Это затрудняет следование концепции построения вне каталога проекта (файлы документации генерируются внутри каталога исходных файлов). Можно указать путь к выходному каталогу в настройках, однако это опять-таки приведёт к проблеме непереносимости проекта.

Выход из этой ситуации, который обычно рекомендуется в случае использования инструмента CMake, заключается в том, что в файле настроек Doxygen вместо путей к инструментам и выходным файлам указываются конструкции вида «@⟨*имя_переменной*⟩@». Далее в описании проекта применяется команда `configure_file()` (п. 2.10.1), которая, используя файл настроек Doxygen в качестве шаблона, генерирует его окончательный вариант в каталоге построения. Для поиска инструментов используются модули CMake, описание которых приведено далее.

Хотя этот метод работает, его использование не совсем удобно: в каждом создаваемом файле настроек Doxygen нужно

найти все места, где указываются пути, и записать туда конструкции с переменными CMake. Ниже будет продемонстрирован альтернативный способ решения проблемы. Его смысл заключается в копировании файла настроек в выходной каталог и дозаписи в его конец нескольких строк с необходимыми путями. Файл настроек Doxygen (кроме комментариев) состоит из строк следующего формата:

```
⟨имя_настройки⟩ = ⟨значение⟩
```

Например, настройка с именем OUTPUT_DIRECTORY содержит путь к каталогу выходных файлов. В документации к системе Doxygen говорится о том, что из нескольких настроек с одинаковым именем в одном файле она учитывает только самую последнюю. На этом и основан представленный здесь метод. Сами настройки каталогов в файле Doxygen можно вообще не заполнять.

Для реализации взаимодействия с системой Doxygen будут необходимы следующие модули поиска CMake:

```
find_package(Doxygen)
```

Эта команда выполняет поиск пути к консольному генератору Doxygen, а также вспомогательной программе dot (если значение переменной DOXYGEN_SKIP_DOT не установлено в истину). Результаты записываются в следующие переменные (табл. 3.8).

Таблица 3.8

Переменные, заполняемые модулем поиска Doxygen

Переменная	Значение
DOXYGEN_FOUND	Истина, если исполняемый файл Doxygen найден
DOXYGEN_EXECUTABLE	Путь к программе Doxygen
DOXYGEN_VERSION	Версия Doxygen
DOXYGEN_DOT_FOUND	Истина, если исполняемый файл dot найден
DOXYGEN_DOT_EXECUTABLE	Путь к программе dot

---

find_package(HTMLHelp)

Этот модуль предназначен для поиска консольного компилятора `hhc.exe`, а также пути к заголовочному файлу и библиотеке HTMLHelp, которые используются в приложениях с поддержкой справки в этом формате. Результаты поиска записываются в следующие переменные (табл. 3.9).

Таблица 3.9

Переменные, заполняемые модулем поиска HTMLHelp

Переменная	Значение
HTML_HELP_COMPILER	Путь к программе `hhc.exe`
HTML_HELP_INCLUDE_PATH	Путь к каталогу с заголовочным файлом `htmlhelp.h`

Окончание табл. 3.9

Переменная	Значение
HTML_HELP_LIBRARY	Путь к библиотеке htmlhelp.lib

---

find_package(
  LATEX [COMPONENTS ⟨*компонент*$_1$⟩ ... ⟨*компонент*$_n$⟩])

Этот модуль предназначен для поиска инструментов, используемых для подготовки документов в системе LaTeX. Из комментариев в тексте модуля (FindLATEX.cmake) можно сделать вывод, что, если в команде не указывать дополнительных компонент, будет найден только путь к инструменту latex. На самом же деле текущая реализация этого модуля находит все инструменты независимо от дополнительных аргументов команды find_package(). Возможные значения этих компонент приведены в табл. 3.10.

Таблица 3.10

Дополнительные компоненты модуля поиска LaTeX

PDFLATEX	XELATEX	LUALATEX	BIBTEX	BIBER
MAKEINDEX	XINDY	DVIPS	DVIPDF	PS2PDF
PDFTOPS	LATEX2HTML	HTLATEX		

Результаты поиска записываются модулем в следующие переменные (табл. 3.11).

Таблица 3.11

Переменные, заполняемые модулем поиска LaTeX

Переменная	Значение
LATEX_FOUND	Истина, если исполняемый файл LaTeX и остальных инструментов найдены
LATEX_⟨*компонент*⟩_FOUND	Истина, если файл компонента найден
LATEX_COMPILER	Путь к компилятору LaTeX
⟨*компонент*⟩_COMPILER	Путь к файлу инструмента

---

find_package(Perl)

Этот модуль выполняет поиск интерпретатора Perl. Результаты записываются в следующие переменные (табл. 3.12).

Таблица 3.12

Переменные, заполняемые модулем поиска Perl

Переменная	Значение
PERL_FOUND	Истина, если исполняемый файл Perl найден
PERL_EXECUTABLE	Путь к файлу Perl
PERL_VERSION_STRING	Версия найденного интерпретатора (поддерживается в CMake начиная с версии 2.8.8)

Пример

Пусть требуется реализовать поддержку создания документации приведённым выше способом.

Файл `doxygen.cmake`:

```
if(NOT DOXYGEN_EXECUTABLE)
  find_package(Doxygen)
  if(NOT DOXYGEN_FOUND)
    message(
      WARNING
      "Could not find Doxygen on this system. You "
      "can manually assign a path to it to "
      "DOXYGEN_EXECUTABLE variable.")
    return()
  endif()
endif()

if(NOT HTML_HELP_COMPILER)
  find_package(HTMLHelp)
  if(NOT HTML_HELP_COMPILER)
    message(
      WARNING
      "Could not find HTMLHelp compiler on this "
      "system. You can manually assign a path to it "
      "to HTML_HELP_COMPILER variable.")
  endif()
endif()
```

```
if(NOT QHG_EXECUTABLE)
  find_package(Qt5Core QUIET)
  if(Qt5Core_FOUND)
    get_property(
      MOC_PATH
      TARGET Qt5::moc
      PROPERTY IMPORTED_LOCATION)
    get_filename_component(
      MOC_DIR "${MOC_PATH}" DIRECTORY)
    find_program(
      QHG_EXECUTABLE
      "qhelpgenerator"
      HINTS "${MOC_DIR}"
      DOC
      "Qt help generator (qhelpgenerator) exec. path")
    if(NOT QHG_EXECUTABLE)
      message(
        WARNING
        "Found Qt5, but failed to find "
        "qhelpgenerator program. You can manually "
        "assign a path to it to QHG_EXECUTABLE "
        "variable.")
    endif()
  else()
    message(
      WARNING
      "Could not find Qt help generator on this "
```

```
      "system. If you have a Qt installation you can "
      "add a path to it to CMAKE_PREFIX_PATH "
      "variable. You can also manually assign "
      "a path to qhelpgenerator program to "
      "QHG_EXECUTABLE variable.")
  endif()
endif()

if(NOT LATEX_COMPILER AND NOT MAKEINDEX_COMPILER)
  find_package(LATEX COMPONENTS MAKEINDEX)
  if(NOT LATEX_FOUND)
    message(
      WARNING
      "Could not find LATEX on this system. You can "
      "manually assign a path to latex executable to "
      "LATEX_COMPILER variable and a path to "
      "makeindex executable to MAKEINDEX_COMPILER "
      "variable.")
  endif()
endif()

if(NOT PERL_EXECUTABLE)
  find_package(Perl)
  if(NOT PERL_FOUND)
    message(
      WARNING
      "Could not find Perl interpreter on this "
```

```
      "system. You can manually assign a path to it "
      "to PERL_EXECUTABLE variable.")
  endif()
endif()

set(DOC_DIR "${CMAKE_BINARY_DIR}/doc")

set(
  VARS
  # Doxygen name      CMake name
  OUTPUT_DIRECTORY    DOC_DIR
  HHC_LOCATION        HTML_HELP_COMPILER
  DOT_PATH            DOXYGEN_DOT_EXECUTABLE
  QHG_LOCATION        QHG_EXECUTABLE
  LATEX_CMD_NAME      LATEX_COMPILER
  MAKEINDEX_CMD_NAME MAKEINDEX_COMPILER
  PERL_PATH           PERL_EXECUTABLE
  )

set(
  CMDS
  COMMAND
    "${CMAKE_COMMAND}" -E copy
      "${CMAKE_CURRENT_SOURCE_DIR}/Doxyfile"
      "${CMAKE_CURRENT_BINARY_DIR}/doc/Doxyfile")

list(LENGTH VARS N)
```

```
foreach(I2 RANGE 1 ${N} 2)
  math(EXPR I1 "${I2} - 1")
  list(GET VARS ${I1} VAR_DOXYGEN)
  list(GET VARS ${I2} VAR_CMAKE)
  set(VALUE_CMAKE "${${VAR_CMAKE}}")
  if(VALUE_CMAKE)
    list(
      APPEND CMDS
      COMMAND
        "${CMAKE_COMMAND}"
          -E echo
          "${VAR_DOXYGEN} = \"${VALUE_CMAKE}\"" >>
          "${CMAKE_CURRENT_BINARY_DIR}/doc/Doxyfile")
  endif()
endforeach()

add_custom_command(
  OUTPUT
    doc/Doxyfile
  ${CMDS}
  MAIN_DEPENDENCY
    Doxyfile
  VERBATIM)

add_custom_target(
  doc
```

```
  COMMAND
    "${DOXYGEN_EXECUTABLE}"
      "${CMAKE_CURRENT_BINARY_DIR}/doc/Doxyfile"
  WORKING_DIRECTORY
    "${CMAKE_CURRENT_SOURCE_DIR}"
  DEPENDS
    doc/Doxyfile)
```

Здесь сначала выполняется поиск требуемых инструментов при помощи соответствующих модулей. Исключение составляет инструмент `qhelpgenerator`, поиск которого не поддерживается конфигурационными файлами Qt и для которого не существует модуля поиска в составе CMake. Вместо этого используется тот факт, что все инструменты Qt при установке помещаются в один каталог. В конфигурационном файле библиотеки `Qt5Core` определена цель `Qt5::moc` для компилятора метаобъектов. Путь к инструменту moc содержится в её свойстве `IMPORTED_LOCATION` (путь к результирующему файлу импортируемой цели). Из этого пути выделяется путь к содержащему файл каталогу при помощи команды `get_filename_component()` (п. 2.9.1), который затем используется для поиска исполняемого файла Qt Help Generator при помощи команды `find_program()` (п. 2.9.2). Библиотека `Qt5Core` не зависит от других компонент, и на их поиск не тратится лишнее время.

После вычисления всех необходимых путей начинается формирование команд цели, которая генерирует файл настро-

ек Doxygen в каталоге построения. Переменной VARS присваивается список из имён настроек Doxygen и переменных CMake, содержащих соответствующие значения. Переменной CMDS присваивается список аргументов команды add_custom_command() (п. 2.10.5), определяющих запускаемые инструменты в процессе построения. Эта команда описывает правило создания файла doc/Doxyfile в каталоге построения из Doxyfile каталога проекта. Сначала переменной CMDS присваиваются аргументы, определяющие команду копирования входного файла в выходной («cmake -E copy ...», см. также пример на с. 223 и замечание к нему). Дальше элементы списка VARS обходятся в цикле парами (настройка Doxygen + переменная CMake). К переменной CMDS добавляется определение команды, которая дописывает в выходной файл строку вида «⟨*настройка*⟩ = "⟨*путь*⟩"». Это делает команда «cmake -E echo ...» с перенаправлением вывода в файл.

Последней вызывается команда add_custom_target() (п. 2.10.4), определяющая фальшивую цель для вызова инструмента Doxygen, которому передаётся сгенерированный файл настроек. В качестве рабочего каталога программы устанавливается каталог текущего подпроекта. Это позволяет не менять в настройках Doxygen путей к обрабатываемым файлам и каталогам исходных кодов (INPUT и т. д.). В противном случае пришлось бы реализовывать синтаксический разбор файла Doxyfile, что является нетривиальной задачей (в файле настроек можно подключать другие файлы командами @INCLUDE).

Каталог для генерируемых файлов ранее был записан в файл `Doxyfile` (настройка `OUTPUT_DIRECTORY`).

Пример файла `CMakeLists.txt`, использующего приведённый выше модуль `doxygen.cmake`:

```
cmake_minimum_required(VERSION 2.8)

project(ex-doxygen)

add_executable(
  ex-doxygen
  ex-doxygen.cpp ex-classes.cpp ex-classes.h)

include(doxygen.cmake)
```

*

Замечание: в качестве альтернативы вызовам команд вида:

```
cmake -E echo "⟨строка⟩" >> ⟨файл⟩
```

в цели `doc/Doxyfile` можно было бы использовать одну команду исполнения сценария («`cmake -P ...`», п. 2.1.2), который можно сгенерировать командой `configure_file()` (п. 2.10.1). ▲

3.6. Упражнения

3.6.1. Тест рубежного контроля

1. Команда без указания дополнительных компонент:

```
find_package(OpenCV REQUIRED)
```

приведёт к тому, что:

(a) Будут найдены файлы скомпилированных модулей всех библиотек OpenCV, имена соответствующих им целей будут записаны в переменную `OpenCV_LIBS`.

(b) Поиск файлов библиотек OpenCV выполняться не будет, конфигурационным файлом будет заполнена лишь переменная CMake `OpenCV_INCLUDE_DIRS` (каталоги подключаемых файлов библиотек) и остальные переменные конфигурационного файла.

(c) Исполнение команды приведёт к аварийному останову инструмента CMake с выводом сообщения об ошибке.

2. Команда без указания дополнительных компонент:

```
find_package(Boost REQUIRED)
```

приведёт к тому, что:

(a) Будут найдены файлы скомпилированных модулей всех библиотек Boost, пути к ним будут записаны в переменную CMake `Boost_LIBRARIES`, а также для каждого компонента по отдельности в переменную `Boost_`⟨*компонент*⟩`_LIBRARY`.

(b) Поиск файлов библиотек Boost выполняться не будет, модулем поиска будет заполнена лишь переменная CMake `Boost_INCLUDE_DIRS` (каталоги подключаемых файлов библиотек) и остальные переменные модуля поиска.

(c) Исполнение команды приведёт к аварийному останову инструмента CMake с выводом сообщения об ошибке.

3. Команда без указания дополнительных компонент:

```
find_package(Qt5 REQUIRED)
```

приведёт к тому, что:

(a) Будут найдены файлы скомпилированных модулей всех библиотек Qt, пути к ним будут записаны в свойство `IMPORTED_LOCATION` (а для конфигураций с непустыми именами — также в свойства `IMPORTED_LOCATION_`⟨*конфигурация*⟩) целей библиотек с именами `Qt5::`⟨*компонент*⟩.

(b) Поиск файлов библиотек Qt выполняться не будет, конфигурационным файлом будет заполнена лишь переменная CMake `Qt5_INCLUDE_DIRS` (каталоги подключаемых файлов библиотек) и остальные переменные конфигурационного файла.

(c) Исполнение команды приведёт к аварийному останову инструмента CMake с выводом сообщения об ошибке.

4. Команда без указания дополнительных компонент:

```
find_package(LATEX)
```

приведёт к тому, что:

(a) Будут найдены исполняемые файлы всех инструментов LaTeX (`latex`, `bibtex` и т. д.), пути к ним будут записаны в переменные CMake ⟨*компонент*⟩`_COMPILER`.

(b) Будет найден исполняемый файл только инструмента `latex`, путь к нему будет записан в переменную CMake `LATEX_COMPILER`.

(c) Исполнение команды приведёт к аварийному останову инструмента CMake с выводом сообщения об ошибке.

5. Если к проекту, использующему набор библиотек Qt 5, требуется добавить изображения для использования в исполняемой цели, имя файла описания ресурсов для изображений в описании проекта на языке CMake нужно передать команде:

(a) `qt5_wrap_cpp()`; (b) `qt5_add_resources()`;

(c) `qt5_wrap_ui()`.

6. Если в проекте, использующем набор библиотек Qt 5, требуется добавить слот к классу диалогового окна (производного от `QDialog`), обрабатывающий сигналы от дочерних элементов управления (кнопки, поля ввода и т. д.), имя заголовочного файла с определением класса диалога в описании проекта на языке CMake нужно передать команде:

(a) `qt5_wrap_cpp()`; (b) `qt5_add_resources()`;

(c) `qt5_wrap_ui()`.

7. Вызов команды `qt5_create_translation()` приведёт к добавлению в файлы конечной системы построения правил запуска:
   (a) только инструмента `lupdate`;
   (b) только инструмента `lrelease`;
   (c) как инструмента `lupdate`, так и `lrelease`.
8. Вызов команды `qt5_add_translation()` приведёт к добавлению в файлы конечной системы построения правил запуска:
   (a) только инструмента `lupdate`;
   (b) только инструмента `lrelease`;
   (c) как инструмента `lupdate`, так и `lrelease`.

3.6.2. Проектное задание

1. Изучите документацию к наборам библиотек Qt и OpenCV. Реализуйте приложение с графическим пользовательским интерфейсом, предназначенное для просмотра изображений в формате JPEG и других, поддерживаемых Qt, и применения к ним одного из фильтров, реализованных в модуле OpenCV imgproc[20] (фильтр Собеля, сглаживающий фильтр, преобразования поворота и перспективы, поиск элементов изображения и т. д.). Основные классы Qt, которые могут пригодиться в процессе разработки:

[20] http://docs.opencv.org/modules/imgproc/doc/filtering.html (дата обращения: 13.05.2015).

QMainWindow: реализует основное окно приложения с однооконным пользовательским интерфейсом.

QImage: организует загрузку и хранение в памяти изображения. Существуют способы преобразования изображения в формат, поддерживаемый OpenCV, и обратно[21].

QLabel: реализует элемент пользовательского интерфейса, способный выводить изображение из объекта класса `QPixmap` (который может создаваться из объекта `QImage`).

QScrollArea: реализует отображение другого элемента управления в области ограниченного размера, добавляя при необходимости полосы прокрутки. Необходим для вывода больших изображений в объектах `QLabel`. Сам объект можно расположить в главном окне приложения при помощи метода `QMainWindow::setCentralWidget()`.

QFileDialog: реализует вывод диалогового окна открытия файла.

QThread: реализует возможность запуска длительных по времени операций в отдельном потоке.

Требования к разрабатываемому приложению:

— Операции пользовательского интерфейса необходимо реализовать в виде меню или панели управления и т. п. с применением механизма сигналов и слотов.

[21] http://answers.opencv.org/question/7779/convert-cvmat-to-qimage/ (дата обращения: 13.05.2015).

— Пользовательский интерфейс необходимо разработать в приложении Qt Designer с компиляцией инструментом uic.

— Необходимо поместить пиктограмму основного окна либо изображения панели инструментов в ресурсы, компилируемые инструментом rcc.

— Необходимо реализовать поддержку локализации приложения при помощи инструментов Qt Linguist и т. д.

— Необходимо реализовать в проекте цель установки приложения при помощи стандартного модуля CMake `DeployQt4`.

2. Изучите документацию к библиотеке MPIR[22], реализующей арифметические операции с длинными целыми, рациональными и вещественными числами. Изучите возможные варианты расположения файлов библиотеки в каталогах сборки и установки при использовании компиляторов gcc и Visual C++, а также сборки в виде статических и динамических (разделяемых) версий. Реализуйте модуль CMake для подключения библиотеки, аналогичный приведённому в настоящем учебнике для библиотеки Crypto++ (п. 3.4). При помощи этого модуля реализуйте тестовое приложение, выполняющее шифрование и дешифрование длинного целого числа при помощи криптографической схемы

[22] `http://mpir.org/` (дата обращения: 24.09.2015).

RSA[23]. Протестируйте сборку и работу приложения на платформах Windows с различными компиляторами и Linux.

[23] http://people.csail.mit.edu/rivest/Rsapaper.pdf (дата обращения: 24.09.2015).

Заключение

На этом мы завершаем изучение системы CMake. Надеемся, что к этому моменту читатели должны понять причины столь высокой популярности этого инструмента в среде разработчиков программного обеспечения. Несмотря на то что представленный здесь материал больше подходит для начального освоения CMake, его вполне достаточно для решения многих типичных задач построения проектов, в том числе таких, решение которых с помощью других систем требует больших усилий. В настоящем руководстве были рассмотрены методы решения следующих задач:

— Описание простых проектов, состоящих из целей приложений и библиотек, настройка взаимосвязей между целями.

— Организация крупных проектов в виде нескольких более простых проектов нижнего уровня, расположенных в разных подкаталогах.

— Настройка построения целей: определение параметров компиляции и компоновки, определение подкаталогов поиска подключаемых файлов и подкаталогов для выходных файлов. Определение настроек для зависимых целей, что существенно упрощает подключение библиотек.

— Организация хранения данных и пользовательских настроек построения в кэше, что позволяет ускорять последующие запуски построения и организовать интерфейс его настроек.

— Реализация специальных целей, выполняющих генерирование файлов исходных текстов по шаблону (аналогично Autotools).

— Реализация целей, выполняющих тестирование проекта.

— Реализация целей установки результатов работы других целей проекта, файлов, необходимых разработчикам (заголовочные файлы, библиотеки импорта, документация и т. д.), а также сценариев CMake, облегчающих подключение библиотек сторонними проектами.

— Реализация аналогов фальшивых целей системы make, выполняющих заданные последовательности вызова инструментов, а также целей, генерирующих файлы при помощи произвольных наборов инструментов. Добавление вызовов собственных инструментов к другим целям. Вызываемые инструменты могут быть как внешними, так и результатами исполнения других целей.

— Организация передачи различных настроек целям в зависимости от используемого набора инструментов компилятора, платформы, конфигурации и т. д.

— Использование в проектах некоторых популярных внешних наборов библиотек и инструментов: Boost, Qt, OpenCV,

Crypto++, системы управления версиями и создания документации.

Вместе с тем в настоящий учебник ввиду ограниченности его объёма не вошло рассмотрение следующих важных вопросов:

— Реализация конфигурационных файлов CMake для разрабатываемых пакетов при помощи стандартного модуля `CMakePackageConfigHelpers`. Конфигурационные файлы необходимы для того, чтобы пакеты можно было использовать в сторонних проектах, подключая их командой `find_package()`. При этом пакеты могут быть установлены в произвольные каталоги, все вспомогательные пути в конфигурационных файлах будут вычисляться относительно их расположения.

— Аналогично, реализация модулей поиска для сторонних пакетов, в составе которых отсутствуют конфигурационные файлы CMake. Модули поиска позволяют подключать такие пакеты при помощи той же команды `find_package()`. В их реализации, как и в реализации конфигурационных файлов, активно используются низкоуровневые средства CMake для описания импортируемых целей.

— Создание пакетов установки в различных форматах (NSIS[24], ZIP, 7Z, RPM и т. д.) при помощи инструмента CPack и одноимённого пакета, входящих в состав CMake.

[24] http://nsis.sourceforge.net/ (дата обращения: 10.04.2015).

— Использование других стандартных модулей CMake (около 160) для решения различных вспомогательных задач. В настоящем учебнике были рассмотрены лишь некоторые из них.

— Создание сложных сценариев тестирования с возможностями запуска тестов покрытия, анализа использования памяти, выгрузки результатов на сервер сбора статистики тестирования CDash[25] и т. д. при помощи сценариев инструмента CTest.

— Организация кросс-компиляции.

— Определение пользовательских свойств и определение совместимости свойств целей библиотек и соответствующих свойств зависимых целей — аналогично некоторым стандартным свойствам. Например, стандартное логическое свойство `POSITION_INDEPENDENT_CODE` для цели определяет, будет ли она скомпилирована в код, не зависящий от адреса загрузки (используется для разделяемых библиотек в системе Linux и т. д.). Свойство `INTERFACE_POSITION_INDEPENDENT_CODE` определяет требование для зависимых целей устанавливать для себя свойство `POSITION_INDEPENDENT_CODE` таким же значением. В противном случае система CMake выведет диагностическое сообщение. Такое же поведение можно определять для пользовательских свойств (логические значения, номера версий и т. д.).

[25] `http://www.cdash.org/` (дата обращения: 10.04.2015).

— Работа с файлами на низком уровне (чтение, запись, копирование, поиск, загрузка и т. д.) при помощи команды `file()`, запуск процессов при помощи команды `execute_process()`. В отличие от рассмотренных в настоящем учебнике команд `configure_file()`, `add_custom_target()` и т. д., эти команды не определяют правил целей и выполняются только на этапе запуска инструмента CMake. Чтобы использовать их в определении целей (`add_custom_target()`, `add_custom_command()`), можно использовать вызов инструмента CMake (при помощи переменной `CMAKE_COMMAND`) с ключом «`-P`» (исполнение сценария, п. 2.1.2). Таким способом можно определять кроссплатформенные цели, запускающие процессы и выполняющие обработку файлов со сложной логикой.

— Определение экспериментов с компиляцией тестовых исходных файлов с возможными последующими попытками исполнения скомпилированных программ на этапе запуска инструмента CMake (аналогично системе Autotools) при помощи низкоуровневых команд. Были рассмотрены только некоторые стандартные модули, использующие эти команды, такие как `CheckCXXSourceCompiles`.

— Определение целей библиотек, состоящих из объектных модулей, и определение исходных файлов и объектных модулей для подключения к зависимым целям.

— Организация логически взаимосвязанных наборов файлов и целей в виде именованных групп в генерируемых описа-

ниях проектов для интегрированных сред разработки (Visual Studio и т. д.). Интегрированные среды отображают такие проекты в виде деревьев с именованными раскрывающимися группами целей и файлов, что существенно упрощает для разработчиков ориентирование в крупных проектах.

Как можно видеть, несмотря на актуальность всех этих задач, их изучение может быть отложено начинающими пользователями системы CMake на более поздние этапы её освоения.

Мы желаем всем разработчикам успехов в освоении любых технологий, которые помогут облегчить их труд, а всем пользователям — как можно больше качественных программных продуктов.

Библиография

1. *Бланшет Ж.*, *Саммерфилд М.* Qt 4. Программирование GUI на C++ : пер. с англ. — 2-е изд. — СПб. : КУДИЦ-Пресс, 2008. — 718 с. — ISBN 978-5-91136-059-7.
2. *Боровский А.* Интроспекция и логика // Linux Format. — 2008. — Окт. — 10 (110). — С. 92—95. — URL: http://www.linuxformat.ru/anons110.phtml (дата обращения: 18.12.2014).
3. *Боровский А.* Раздвигая горизонты // Linux Format. — 2008. — Нояб. — 11 (111). — С. 82—84. — URL: http://www.linuxformat.ru/anons111.phtml (дата обращения: 18.12.2014).
4. *Боровский А.* Собираясь в путь // Linux Format. — 2008. — Сент. — 9 (109). — С. 92—95. — URL: http://www.linuxformat.ru/anons109.phtml (дата обращения: 18.12.2014).
5. *Саммерфилд М.* Qt. Профессиональное программирование. Разработка кроссплатформенных приложений на C++ : пер. с англ. — СПб. : Символ-плюс, 2011. — 560 с. — ISBN 978-5-93286-207-0.
6. *Соммервилл И.* Инженерия программного обеспечения : пер. с англ. — 6-е изд. — М. : Вильямс, 2002. — 624 с. — ISBN 5-8459-0330-0.
7. *Хоффман Б.*, *Мартин К.* Разработка программного обеспечения в небольшой организации : пер. с англ. // Открытые системы. — 2007. — № 03. — URL: http://www.osp.ru/os/2007/03/4158385/ (дата обращения: 18.12.2014).

8. *Шлее М.* Qt 5.3. Профессиональное программирование на C++. — СПб. : БХВ-Петербург, 2015. — 928 с. — (В подлиннике). — ISBN 978-5-9775-3346-1.

9. CMake 3.2 Documentation / Kitware, Inc. — 04.2015. — URL: http://www.cmake.org/cmake/help/v3.2/ (дата обращения: 02.05.2015).

10. GNU Make Manual / Free Software Foundation. — 05.10.2014. — URL: http://www.gnu.org/software/make/manual/ (дата обращения: 23.12.2014).

11. *Martin K., Hoffman B.* Mastering CMake. — 6th ed. — Kitware, Inc., 09.2013. — 641 p. — ISBN 978-1-930934-26-9.

12. *Miller P. A.* Recursive Make Considered Harmful // AUUGN: The Journal of AUUG Inc. — 1998. — Sept. — Vol. 19, no. 1. — P. 14–25. — ISSN 1035-7521. — URL: http://miller.emu.id.au/pmiller/books/rmch/ (дата обращения: 26.12.2014).

13. The Open Group Base Specifications Issue 7: IEEE Std 1003.1™, 2013 Edition / The IEEE, The Open Group. — URL: http://pubs.opengroup.org/onlinepubs/9699919799/ (дата обращения: 24.12.2014).

14. *Thelin J.* Foundations of Qt Development. — Apress, 2007. — 528 p. — ISBN 978-1-59059-831-3. — DOI: 10.1007/978-1-4302-0251-6.

Ответы на тесты

С. 42:

1. (a), (c). 2. (a), (b), (c), (e).

С. 248:

1. (a). 2. (a). 3. (a). 4. (b). 5. (a). 6. (c). 7. (c). 8. (b). 9. (b). 10. (b). 11. (d). 12. (e), (h). 13. (c). 14. (b). 15. (g).

С. 394:

1. (a). 2. (b). 3. (c). 4. (a). 5. (b). 6. (a). 7. (c). 8. (b).

Предметный указатель

К

Л

М

О

П

Р

С

Т

Ф

Ц

B

C

L

O

P

Q

R