

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ**
**Федеральное государственное бюджетное
образовательное учреждение высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»**

М. Ю. ЛУПАНОВ, Н. А. СИДОРОВ

**РАЗРАБОТКА
КЛИЕНТ-СЕРВЕРНЫХ ПРИЛОЖЕНИЙ
С ПОДСИСТЕМОЙ АУТЕНТИФИКАЦИИ**

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

ПЕНЗА 2024

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное
образовательное учреждение высшего образования
«Пензенский государственный университет» (ПГУ)

М. Ю. Лупанов, Н. А. Сидоров

Разработка
клиент-серверных приложений
с подсистемой аутентификации

Учебно-методическое пособие

Пенза
Издательство ПГУ
2024

УДК 004.056
Л85

Рецензент

доктор технических наук, доцент,
заместитель начальника отделения
Научно-производственного предприятия «Рубин» (г. Пенза)
А. А. Ганин

Лупанов, Михаил Юрьевич.

Л85 Разработка клиент-серверных приложений с подсистемой аутентификации : учеб.-метод. пособие / М. Ю. Лупанов, Н. А. Сидоров. – Пенза : Изд-во ПГУ, 2024. – 76 с.

Рассмотрены вопросы разработки клиент-серверных многомодульных приложений с подсистемой аутентификации на языке Си++, функциональное и модульное тестирование, а также формирование документации с использованием генераторов документации. В качестве приложений в пособии приводятся примеры программ для разбора параметров командной строки, использования криптографических библиотек, модульного тестирования.

Издание подготовлено на кафедре «Информационная безопасность систем и технологий» Пензенского государственного университета и предназначено для обучающихся по специальностям 10.05.02 «Информационная безопасность телекоммуникационных систем» и 10.05.03 «Информационная безопасность автоматизированных систем» при выполнении курсовой работы.

УДК 004.056

© Пензенский государственный
университет, 2024

СОДЕРЖАНИЕ

Введение	4
1. Общие положения	5
1.1. Форма отчетности и оценка курсовой работы	5
1.2. Выбор темы	5
2. Этапы курсовой работы	6
2.1. Оформление задания	6
2.2. Проектирование программы	7
2.2.1. Построение UML-диаграмм вариантов использования	8
2.2.2. Построение UML-диаграмм классов	8
2.2.3. Построение UML-диаграмм последовательностей	8
2.2.4. Построение UML-диаграмм деятельности	9
2.2.5. Проектирование пользовательского интерфейса	9
2.3. Реализация программы	9
2.4. Тестирование программы	10
2.5. Документирование программы	10
2.6. Отчет по курсовой работе	11
2.7. Защита курсовой работы	11
3. Рекомендации по выполнению этапов курсовой работы	12
3.1. Построение UML-диаграмм	12
3.1.1. Диаграммы вариантов использования	12
3.1.2. Диаграмма классов	15
3.1.3. Диаграмма последовательностей	16
3.1.4. Диаграмма деятельности	17
3.2. Проектирование пользовательского интерфейса	18
3.3. Реализация программы	20
3.3.1. Функция main	20
3.3.2. Реализация пользовательского интерфейса командной строки	21
3.3.3. Реализация сетевого взаимодействия	24
3.3.4. Реализация подсистемы аутентификации с помощью библиотеки Crypto++	37
3.4. Тестирование программы	43
3.4.1. Модульное тестирование	43
3.4.2. Функциональное тестирование	51
3.5. Документирование программы	53
Приложение А. Программа для разбора параметров командной строки с использованием библиотеки boost	63
Приложение Б. Примеры программ для формирования «соли», формирования хеша, проверки хеша	66
Приложение В. Примеры программ для формирования и проверки подписи аутентификационного сообщения	69
Приложение Г. Программа тестирования интерфейсного модуля	72

Введение

Разработка программного обеспечения в современных реалиях – сложный процесс. С одной стороны, есть требования по сокращению сроков и снижению стоимости разработки, с другой – требования по функциональности и удобству использования. При этом к программам предъявляются требования по надежному и безопасному функционированию. Выполнение всех условий обеспечивается за счет использования различных технологий разработки программного обеспечения.

Также не стоит забывать, что программное обеспечение чаще всего разрабатывается как часть сложной информационной системы, где необходимо обеспечивать взаимодействие между различными компонентами, в том числе и с использованием телекоммуникационных сетей. Для построения таких систем используются различные архитектуры, но наиболее распространенной является модель архитектуры «клиент-сервер».

В модели «клиент-сервер» могут взаимодействовать как программные приложения, так и сетевые устройства. Серверы в этой модели предназначены для обработки больших объемов информации. Они находятся в режиме ожидания запроса от клиентов. Сетевые клиенты выполняют такие запросы, отправляя специальные сообщения серверам. Серверы обрабатывают запросы и возвращают результаты. При этом, чтобы избежать несанкционированного использования серверов, в процесс обмена информацией должна включаться процедура аутентификации.

Как правило, один сервер поддерживает множество клиентов и несколько серверов могут быть объединены в пул, чтобы справиться с возросшей нагрузкой на обработку по мере роста числа клиентов. Многие из популярных сетевых приложений работают по модели «клиент-сервер», включая электронную почту, файловые и веб-сервисы.

В учебно-методическом пособии рассматриваются вопросы разработки на языке Си++ клиент-серверных приложений с подсистемой аутентификации. Приведен порядок осуществления функционального и модульного тестирования разрабатываемых программ, а также формирование документации разработчика с использованием специализированного языка разметки и генератора документации. В приложениях приведены примеры программ для реализации пользовательского интерфейса командной строки, функций аутентификации, а также примеры модульного тестирования программы.

При выполнении курсовой работы студенты получают опыт проектирования и разработки клиент-серверных приложений с подсистемой аутентификации, овладеют навыками тестирования и документирования программного обеспечения.

1. Общие положения

1.1. Форма отчетности и оценка курсовой работы

Работа над курсовой начинается с выбора темы и оформления задания. Курсовая работа выполняется в течение семестра поэтапно, в соответствии с заданием. Результаты по каждому этапу загружаются в ЭИОС и оцениваются преподавателем. Сумма баллов за все этапы представляет собой оценку за семестр и должна быть в пределах от 36 до 60 баллов.

По итогам курсовой работы оформляется отчет и выполняется ее защита в виде доклада. При дистанционной форме защиты работы допускается как представление записи заранее подготовленного доклада, так и выполнение его в режиме онлайн. По результатам доклада студенту могут быть заданы вопросы по теме, на которые он обязан ответить. За отчет по курсовой работе, доклад и ответы на вопросы студент получает от 24 до 40 баллов.

1.2. Выбор темы

На первой неделе семестра студент должен выбрать тему курсовой работы и приступить к ее выполнению. Тема может быть выбрана из предложенных преподавателем или сформулирована студентом самостоятельно и согласована с преподавателем для оценки трудоемкости работы и соответствия темы программе дисциплины.

2. Этапы курсовой работы

2.1. Оформление задания

В ходе курсовой работы студент должен реализовать процессы следующих этапов жизненного цикла программного обеспечения (ПО):

- анализ требований к программе;
- оформление технического задания;
- проектирование программы;
- реализация программы;
- тестирование программы;
- документирование программы.

В начале работы студент должен проанализировать требования к программе, сформулировать их и оформить в виде документа «Задание на курсовую работу».

Задание на курсовую работу оформляется в соответствии с образцом, представленным в ЭИОС дисциплины. В п. 1 задания указывается дисциплина, в п. 2 – фамилия, имя, отчество студента, в п. 3 – группа студента.

В п. 4 задания студент указывает цель работы (п. 4.1) и требования к программе (п. 4.2). При формулировке требований к программе следует придерживаться следующих правил. Требования должны быть структурированы и ранжированы в соответствии с их назначением:

- 1) требования к функционалу программы (перечисление функций и действий, которые программа должна выполнять);
- 2) требования к данным, обрабатываемым программой (источники данных, формат, способ ввода и вывода и т.п.);
- 3) требование к пользовательскому интерфейсу (тип интерфейса, предоставляемые интерфейсом возможности);
- 4) требование к среде функционирования (требования к аппаратным ресурсам, к ОС и другим программам, с которыми выполняется взаимодействие);
- 5) требования к средствам разработки (требования к компиляторам, библиотекам, средствам тестирования и отладки и т.п.).

При формулировке требований студент должен придерживаться следующих принципов:

- 1) однозначность – требование не должно иметь различных вариантов толкований;

- 2) полнота – требование не должно оставлять неясных моментов;
- 3) атомарность – требование должно быть максимально простым, не содержать союзов «и», «или», перечислений;
- 4) отслеживаемость – должно быть понятно, какими внешними условиями (законы, приказы и т.п.) требование порождено и на какие алгоритмы, модули, функции, классы программы оно влияет;
- 5) проверяемость – требования должны подразумевать возможность проверки их выполнения, например тестированием;
- 6) выполнимость – требование должно подразумевать наличие возможности его реализации в программе;
- 7) непротиворечивость – требования не должны вступать в конфликт друг с другом, со стандартами, законами и другими нормативными документами.

В п. 5 задания указывается структура работы: в подп. 5.1 – состав отчета по курсовой работе; в подп. 5.2 – состав графической части (чертежи, плакаты, презентации и т.п.). Графическая часть может отсутствовать.

В п. 6 задания приводится календарный график выполнения курсовой работы. Календарный график привязан к неделям текущего семестра. В качестве даты выполнения данного этапа может указываться либо конец недели, либо начало следующей недели.

Задание подписывают исполнитель, нормоконтролер, руководитель и утверждает заведующий кафедрой.

По окончании работы оформляется отчет и выполняется защита работы.

2.2. Проектирование программы

На этапе «Проектирование программы» разрабатывается проект программы в виде совокупности диаграмм *UML*. Решение о том, какие диаграммы необходимы для реализации программы, студент принимает самостоятельно. Однако в ходе проектирования должны быть обязательно разработаны следующие диаграммы:

- диаграмма вариантов использования;
- диаграмма классов;
- диаграмма последовательностей;
- диаграмма деятельности.

Кроме диаграмм *UML*, на данном этапе могут быть разработаны алгоритмы реализации отдельных методов классов. Решение о разработке алгоритмов принимается в том случае, если реализация метода класса должна содержать циклы или ветвления.

2.2.1. Построение *UML*-диаграмм вариантов использования

Диаграмма вариантов использования должна показывать:

- все возможные классы пользователей программы;
- все основные варианты использования программы всеми классами пользователей;
- варианты использования, расширяющие основные варианты неким дополнительным функционалом;
- варианты использования, выполняющие некоторые специфичные функции и рассматривающиеся как обособленная часть основных вариантов использования.

2.2.2. Построение *UML*-диаграмм классов

Диаграммы классов являются центральной частью проекта объектно ориентированной программы и содержат описание классов программы, их атрибутов и операций (методов), а также отношения между классами программы.

Диаграмма классов может быть обобщенной и детальной. На обобщенной диаграмме допускается указывать только имена классов. Иногда на ней перечисляют наиболее значимые атрибуты и операции, но без детализации, только указывая имена. Цель обобщенной диаграммы – показать отношения между классами.

На детальной диаграмме может изображаться отдельная часть обобщенной диаграммы или даже отдельный класс. На таких диаграммах указываются все атрибуты класса с обозначением не только имен, но и их типов. Для операций указываются имена самих операций, имена и типы передаваемых операции параметров, а также тип возвращаемого результата. По такой диаграмме можно написать полную спецификацию класса.

2.2.3. Построение *UML*-диаграмм последовательностей

Диаграммы последовательностей также являются частью этапа проектирования и показывают взаимодействие пользователей и классов программ друг с другом на шкале времени. Таким образом, диаграмма последовательностей показывает порядок выполняемых операций и раскрывает причинно-следственные связи процессов в проектируемой программе.

2.2.4. Построение *UML*-диаграмм деятельности

Диаграмма деятельности раскрывает алгоритмическую составляющую этапа проектирования программы. Отличие от обычного алгоритма в том, что деятельность проходит через все классы программы, показывая переход одной деятельности между классами.

2.2.5. Проектирование пользовательского интерфейса

Пользовательский интерфейс является одной из самых значимых частей программы, так как именно он позволяет осуществлять взаимодействие пользователя с программой. Пользовательский интерфейс, независимо от своего типа, должен обеспечивать возможность выполнения всех функций программы, перечисленных в техническом задании на проектирование, а также ввод пользовательских данных и вывод результатов работы как отдельных функций, так и программы в целом.

2.3. Реализация программы

На этапе реализации записывается код программы и выполняется ее документирование. Классы реализуются в соответствии с построенными на этапе проектирования диаграммами классов. Наиболее сложные методы классов реализуются в соответствии с ранее разработанными алгоритмами. В случае обнаружения ошибок в проекте программы (диаграммах *UML* или алгоритмах) следует вернуться на предыдущий этап и внести соответствующие изменения в проект.

При написании кода студентам рекомендуется использовать свой репозиторий (например, на *github.com* или других аналогичных ресурсах) в целях совершенствования навыков работы с *git*.

Также необходимо показать свое умение грамотно и эффективно использовать стандартные библиотеки языка программирования для решения тех или иных задач.

Для реализации подсистемы аутентификации необходимо использовать дополнительные библиотеки, так как стандартные не содержат необходимых функций. Одной из предпочтительных библиотек является библиотека *Crypto++*, так как в ней реализованы как разнообразные хеш-функции, так и функция аутентификации *HMAC*.

Процесс документирования программы реализуется с использованием специальной разметки в комментариях программы. Для формирования самой документации необходимо использовать программное средство *Doxygen*.

2.4. Тестирование программы

В рамках курсовой работы выполняется два вида тестирования: модульное и функциональное.

Модульное тестирование проводится на этапе проектирования. Для тестирования каждого класса программы создается модульный тест. Создание модульного теста состоит из разработки тестовых сценариев, когда решается, что и как тестировать, разработки непосредственно кода тестов. Модульное тестирование позволяет еще на этапе проектирования избавиться от большинства дефектов в коде классов. Так как модульное тестирование является автоматизированным, то после написания кода тестов само тестирование не занимает много времени. Для написания тестов можно использовать специализированные библиотеки для модульного тестирования, например фреймворк `UnitTest++`.

Если специализированные библиотеки модульного тестирования не используются, то тогда модульное тестирование следует проводить с использованием специального макроса *assert*. Такой подход допустим, если это упрощает тестирование по сравнению с библиотеками модульного тестирования. Так как все макросы *assert* отключаются при компиляции программы с параметром «-DNDEBUG», то тестирование можно непосредственно встраивать как в основную программу, так и в модули.

Функциональное тестирование проводится на этапе тестирования и по своей сути является приемочным. Задача функционального тестирования – проверить программу на выполнение всех указанных в задании требований. Для функционального тестирования также разрабатываются тестовые сценарии. Однако реализация функциональных тестов может быть, например, ручной. Также для функционального тестирования могут быть использованы скрипты автоматизации на языке оболочки операционной системы.

При тестировании сетевых функций программы студенты могут воспользоваться специально разработанными для этих целей тестовыми программой-клиентом и программой-сервером. Сами программы и инструкции по их использованию размещены в ЭИОС дисциплины.

2.5. Документирование программы

При разработке программы параллельно создается рабочая документация с описанием всех переменных, функций и классов, их методов и атрибутов. Для переменных и атрибутов указывается их назначение. Для методов и функций определяется их назначение,

описываются входные параметры, возвращаемый результат, а также возбуждаемые исключения. Для классов указывается их краткое и полное описание. Также в документации описываются введенные или переопределенные типы данных, не являющиеся классами.

2.6. Отчет по курсовой работе

Отчет по курсовой работе оформляется в соответствии с требованиями к отчетам, предъявляемым на кафедре «Информационная безопасность систем и технологий». Отчет должен содержать титульный лист, задание, введение, несколько разделов с описанием всех этапов курсовой работы в соответствии с подп. 5.1 задания, заключение и приложения.

Код программы должен быть представлен в виде приложения. Также в виде приложения в отчет включается документация на программу, созданная программным средством Doxygen.

После защиты курсовой работы отчет должен быть загружен в соответствующий раздел ЭИОС. При этом титульный лист отчета и задание на курсовую работу должны быть сканированными копиями оригинальных подписанных и утвержденных документов.

2.7. Защита курсовой работы

Защита курсовой работы проходит в виде доклада, в котором четко и кратко изложены основные положения работы, и ответов на вопросы по теме. Продолжительность доклада – от 3 до 5 мин. Студент должен рассказать о задачах, поставленных перед ним в курсовой работе, и кратко доложить о методах и результатах их решения.

В качестве иллюстративной основы для доклада студент может использовать либо отчет о курсовой работе, либо специально подготовленную презентацию.

3. Рекомендации по выполнению этапов курсовой работы

3.1. Построение *UML*-диаграмм

3.1.1. Диаграммы вариантов использования

Исходными данными для построения всех диаграмм служит техническое задание на проектирование программы. Поэтому перед построением диаграмм необходимо тщательно проанализировать все требования к программе, приведенные в техническом задании на проектирование.

При построении диаграммы вариантов использования сначала необходимо определить всех «действующих лиц», взаимодействующих с программой. К ним могут относиться как пользователи-люди, так и программные системы. Пользователи делятся на различные классы, выполняющие разные действия с программой и имеющие различный уровень доступа. Каждый такой класс пользователей должен быть представлен своим «действующим лицом».

После определения «действующих лиц» необходимо определить основные варианты использования программы каждым «действующим лицом». Далее определяются варианты использования, которые либо расширяют основные варианты использования, либо являются их специфичной частью, т.е. включаются в варианты. Также могут быть определены варианты использования, являющиеся частными случаями (подклассами) основных вариантов использования.

Рассмотрим пример диаграммы для программы-клиента. Так как программа предназначена для передачи данных серверу для обработки и приема обработанных данных обратно, то можно сделать вывод, что у программы будет только один класс пользователей, который будет выполнять все возможные действия с программой. Назовем его «Пользователь». Кроме того, программа должна обращаться к некоторой программе-серверу для выполнения некоторых вычислений. Значит, у нас появляется еще одно «действующее лицо». Назовем его «Сервер».

Исходя из технического задания, все взаимодействие «Пользователя» с программой сводится к заданию параметров при старте. Причем пользователь может либо задать все параметры верно и выполнить необходимые расчеты, либо задать параметры неверно и по-

лучить сообщение об ошибке. Кроме того, возможен вариант запуска, когда программа выдает справку о своей работе, при этом при выполнении вычислений будет задействован «Сервер».

Таким образом, к основным вариантам использования программы можно отнести «Выполнить вычисления» и «Выдать справку». На рис. 1 приведена диаграмма вариантов использования в самом общем виде.



Рис. 1. Диаграмма вариантов использования программы-клиента в общем виде

Такая диаграмма является отправной точкой для дальнейшего анализа и уточнения вариантов использования. Вариант «Выдать справку» является элементарным и с ним больше ничего не надо делать, а вот вариант «Выполнить вычисления» подразумевает выполнение нескольких функций. Часть функций можно найти в требованиях к программе: чтение исходных данных из файла, обмена данных с сервером, сохранение результатов в файл. Все три функции можно рассматривать как варианты использования, связанные с вариантом «Выполнить вычисления» отношением включения. При этом очевидно, что «Сервер» будет связан только с одним из этих вариантов. На рис. 2 показан фрагмент диаграммы вариантов использования после добавления данных функций.

Как видим, теперь «Сервер» связан не с основным вариантом «Выполнить вычисления», а с включенным в него вариантом «Работать с сервером».

Дальнейший анализ требований к программе показывает, что мы не отразили на нашей диаграмме работу с пользовательским интерфейсом. Так как он у нас простой, то вся работа сводится к анализу и декодированию параметров командной строки. Поэтому к включаемым вариантам, показанным на рис. 2, следует добавить еще «Декодировать командную строку».

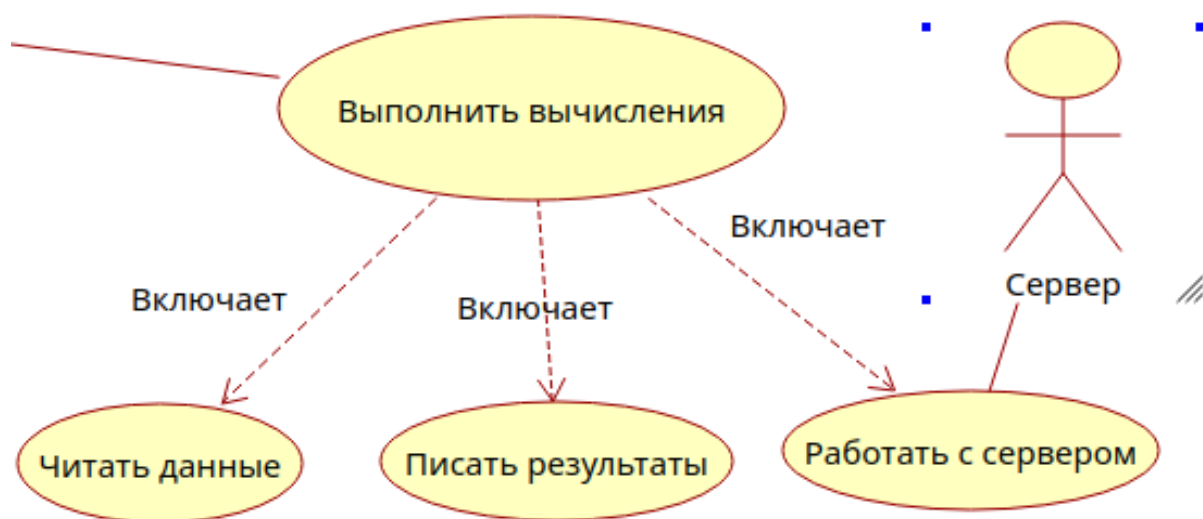


Рис. 2. Фрагмент диаграммы вариантов использования после добавления новых вариантов

Следующим этапом можно детализировать включаемые варианты использования. Так, вариант «Читать данные» может включать такие варианты, как «Открыть файл», «Читать файл», «Закрыть файл». Вариант «Работать с сервером» на основе анализа технического задания будет включать варианты «Установить соединение», «Выполнить аутентификацию», «Передать данные», «Принять данные», «Разорвать соединение».

Решение о целесообразности детализации того или иного варианта использования принимает разработчик на основании того, будут ли в дальнейшем эти новые варианты реализовываться отдельными функциями, классами или методами классов. Если такая реализация не имеет смысла, то и включать такие варианты в диаграмму не следует.

Наконец, переходим к требованию обработки ошибок. Вариант использования «Обработать ошибки» является расширяющим для большинства вариантов использования нашей диаграммы. Отношение «расширяет» показывает, что вариант использования реализуется только при определенных условиях. Все обработчики ошибок всегда являются расширяющими вариантами, так как реализуются только при возникновении ошибок. На рис. 3 приведен пример добавления обработчиков ошибок на диаграмму вариантов использования.

Как видим из рисунка, вариант «Обработать ошибки» расширяет четыре варианта использования. Кроме того, так как «Обработать

ошибки» является обобщенным вариантом использования, он может конкретизироваться частными вариантами, как и показано на рис. 3, с помощью отношений обобщения.



Рис. 3. Добавление обработчика ошибок на диаграмму вариантов использования

3.1.2. Диаграмма классов

Диаграмма классов строится исходя из необходимости реализации некоторых вариантов использования классами. Например, класс *UserInterface* может реализовывать вариант «Декодировать строку», класс *DataReader* – вариант «Читать данные», класс *Communicator* – вариант «Работать с сервером» и т.д. Для обработки ошибок можно создать иерархию классов, соответствующую иерархии вариантов использования обработчиков ошибок.

Так как класс *UserInterface* занимается разбором параметров командной строки и в итоге получает всю информацию по тому, где находятся исходные данные, как связаться с сервером и куда сохранить результаты, то логично будет выполнять работу с остальными классами из него, не забывая при этом, что при возникновении ошибок должны возбуждаться исключения. Пример диаграммы классов приведен на рис. 4.

В этой диаграмме указаны только классы и отношения между ними. Это первый этап построения диаграммы. Далее необходимо детализировать каждый класс, добавив в него атрибуты и операции.

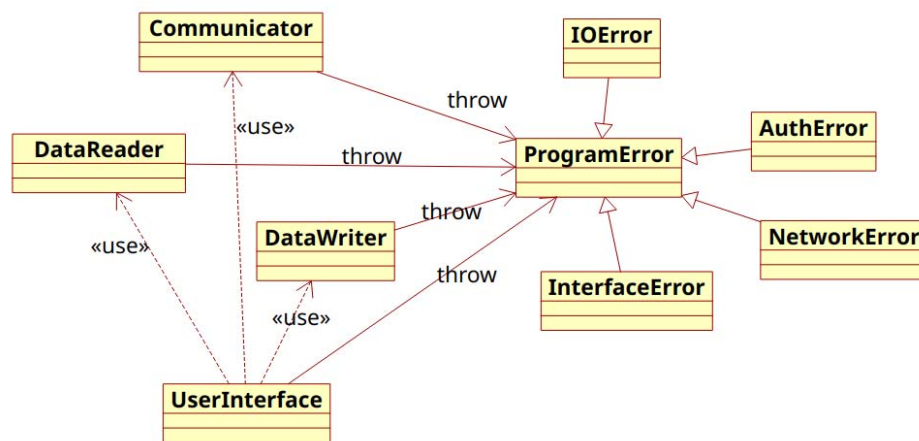


Рис. 4. Пример диаграммы классов

Кроме того, в ходе детализации может понадобиться ввести в диаграмму новые классы, например, для работы с исходными данными и результатами вычислений. Такие классы, используемые для хранения и передачи данных, могут вообще не иметь методов, а только атрибуты.

Если для данных нет необходимости создавать новые классы, а можно обойтись типами данных стандартной библиотеки, то такие типы также указываются на диаграмме классов. Например, для данных, состоящих из набора векторов чисел с плавающей точкой, может быть использован тип, показанный на рис. 5. Такие типы данных должны быть связаны с классами, в которых используются, отношением агрегации.

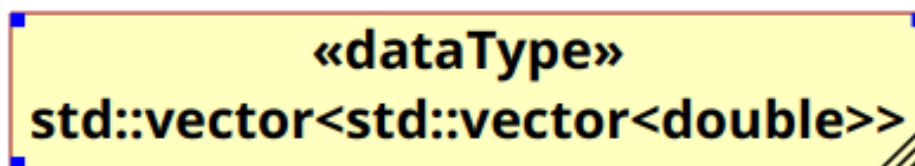


Рис. 5. Пример типа данных для диаграммы классов

3.1.3. Диаграмма последовательностей

Диаграмма последовательностей отражает последовательность взаимодействия объектов программы друг с другом. На этой диаграмме должны быть представлены как классы программы, так и «действующие лица». Особенно важна диаграмма последовательностей для анализа сетевого взаимодействия.

При построении этой диаграммы «Действующие лица» берутся из диаграммы вариантов использования, а классы – из диаграммы

классов. Пример диаграммы последовательностей для рассмотренных ранее диаграмм вариантов использования и классов представлен на рис. 6.

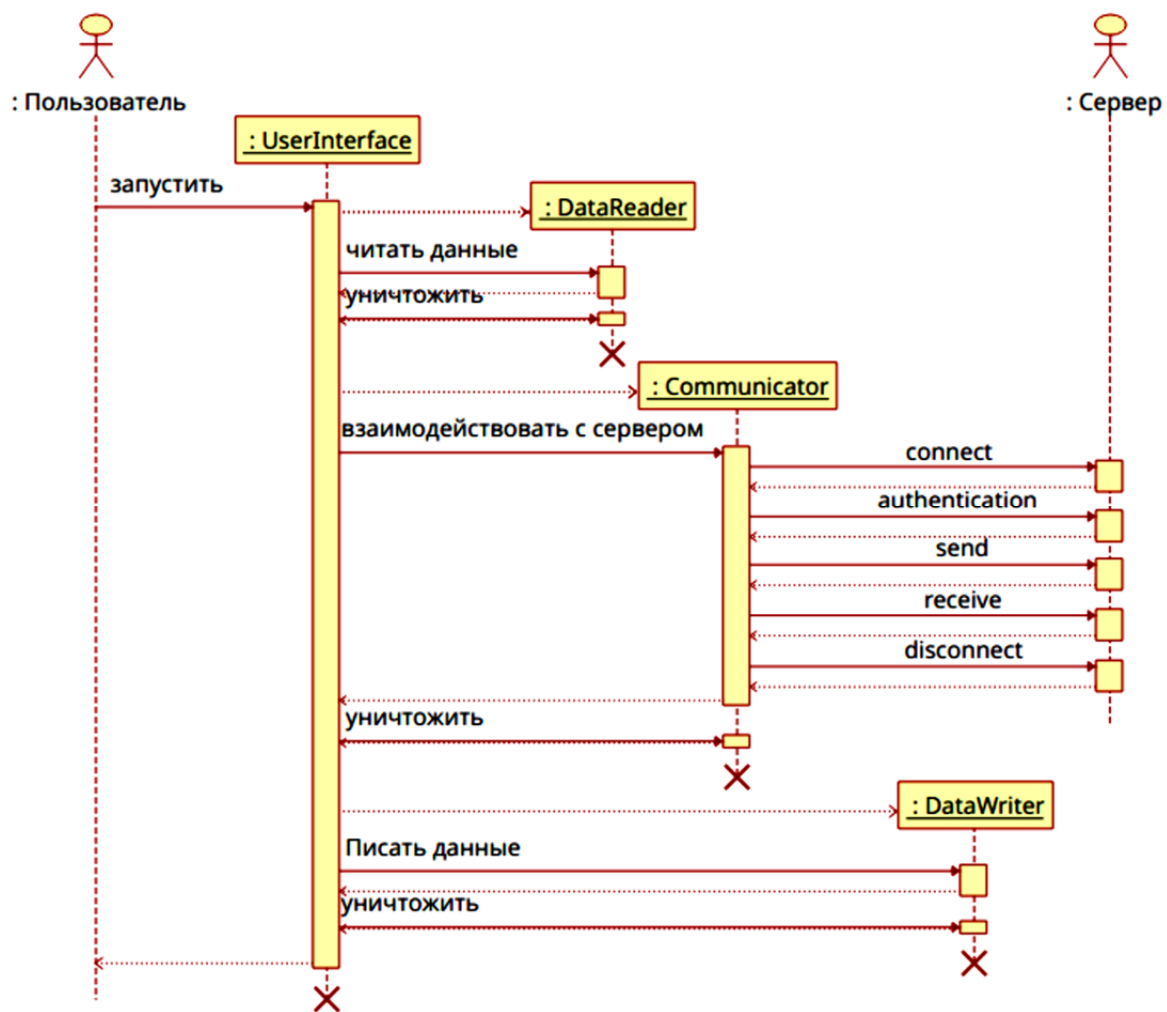


Рис. 6. Пример диаграммы последовательностей

Как видно из рисунка, создание и уничтожение классов для выполнения основных действий происходит из класса *UserInterface*. В этом же классе хранятся все необходимые данные для работы программы.

3.1.4. Диаграмма деятельности

Начать создание диаграммы деятельности можно с представления сообщений диаграммы последовательностей в виде деятельности. На следующем этапе можно детализировать некоторые элементы данной диаграммы в виде отдельных диаграмм деятельности.

Пример общей диаграммы деятельности приведен на рис. 7. Если проанализировать элементы диаграммы, то можно прийти к вы-

воду, что деятельность «Читать данные» простая и не требует детализации, а деятельность «Выполнить аутентификацию» сложная и требует детализации. Исходя из требований к программе, диаграмма деятельности для процесса аутентификации будет выглядеть так, как показано на рис. 8.

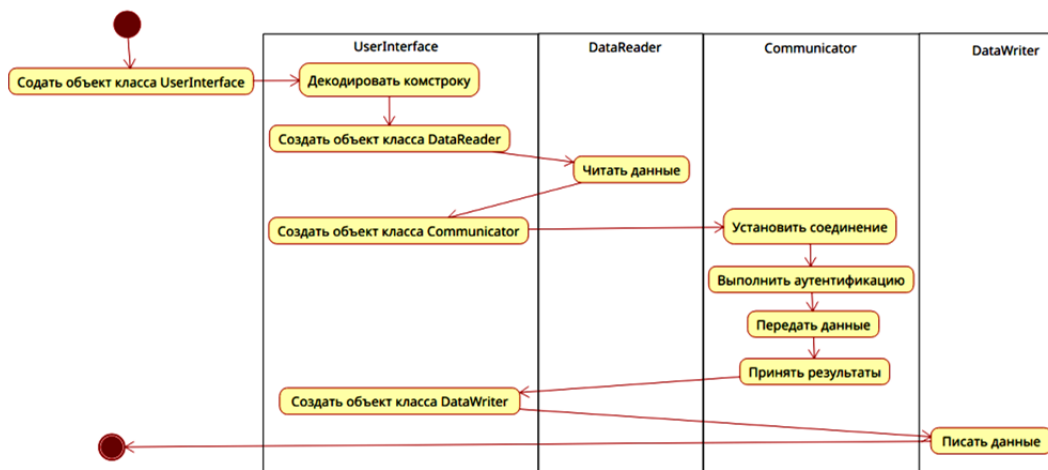


Рис. 7. Пример диаграммы деятельности

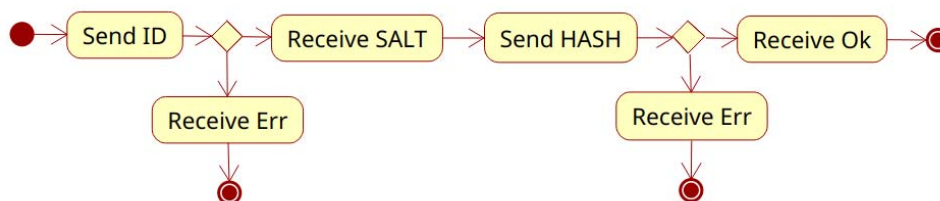


Рис. 8. Диаграмма деятельности для «Выполнить аутентификацию»

3.2. Проектирование пользовательского интерфейса

В соответствии с заданием на проектирование пользовательский интерфейс должен быть реализован в виде командной строки, передаваемой программе при запуске. В этом случае все необходимые данные (или часть их) указываются непосредственно в командной строке.

Если данных много, то возможна передача их в программу в виде файлов. Тогда в командной строке должны быть указаны не сами данные, а их источники.

Результаты работы программы при использовании такого интерфейса могут отображаться на экране в текстовом виде. В случае большого объема данные могут сохраняться в файлах. В этом случае имена файлов для сохранения результатов должны быть указаны в командной строке при запуске программы.

Если программа является серверной и должна работать в непрерывном режиме, то в этом случае информация, выдаваемая программой на экран, может дублироваться в специальный файл, называемый журналом работы программы. Тип и объем информации, записываемой в журнал, обычно задается при запуске программы. Однако для простых программ он может быть фиксированным и не управляться.

Рассмотрим интерфейс для программы-клиента. Исходя из диаграммы вариантов использования, представленной на рис. 1, у программы всего два режима работы: выдача справки и выполнение вычислений. Выдача справки не требует передачи программе каких-либо дополнительных данных, а вот выполнение вычислений требует передачи следующих параметров:

- источник входных данных – для операции чтения;
- адрес и порт сервера – для установки соединения;
- имя пользователя и пароль – для аутентификации на сервере;
- файл для сохранения результатов – для операции записи.

Таким образом, для выполнения вычислений надо передать шесть параметров. Для некоторых параметров можно использовать умолчания, т.е. если параметр не указан, то тогда используется значение по умолчанию. Такие параметры являются необязательными.

Тем не менее некоторые параметры не могут использовать умолчания, например сетевой адрес сервера или имя пользователя. Особый случай с паролем. Указывать пароль в командной строке небезопасно. Поэтому в некоторых случаях для пароля в интерфейсе не задают параметра, а программа запрашивает его при старте интерактивно. Если же пароль задается параметром в командной строке, то пользователь должен предпринять меры предосторожности при запуске программы.

Результат работы по проектированию командной строки лучше всего свести в таблицу, аналогичную табл. 1.

Таблица 1

Проект интерфейса командной строки

Параметр	Значение	Функция	Умолчание	Вариант использования
-h, --help	Нет	Справка	Да	Выдать справку
-i, --input	File name	Источник данных	Input.txt	Выполнить вычисления
-u, --user	User name	Имя пользователя	Нет	
...	...	...	...	

Параметры, относящиеся к разным вариантам использования, должны быть взаимоисключающими, т.е. не должны комбинировать-

ся, и, наоборот, параметры, относящиеся к одному варианту использования, должны быть указаны совместно.

Обратите внимание на первую строку табл. 1. Здесь в столбце «Значение» указано «Нет», т.е. у параметра нет значения, для получения справки достаточно просто указать этот параметр. В столбце «Умолчание» указано «Да». Это означает, что если не указано никаких параметров из других вариантов использования, то подразумевается вариант использования «Выдать справку» с параметром «-h, --help». Аналогично, если для какого-то варианта использования не указаны все обязательные параметры, то этот вариант использования выполнить нельзя и выполняется вариант использования по умолчанию.

3.3. Реализация программы

3.3.1. Функция `main`

В объектно ориентированной программе весь функционал должен реализовываться классами. При этом надо стараться соблюдать для классов правило единственной ответственности, т.е. каждый класс отвечает за что-то одно: один вариант использования, одна операция и т.п.

Исходя из этого, главная функция `main` не должна выполнять каких-либо действий, кроме как создание основных объектов, передача им управления и обработка исключений.

Допустим, что мы реализуем проект, представленный на рис. 4. Тогда наша функция `main` должна создать экземпляр класса *UserInterface*, передать ему управление и обработать возникшие исключения:

```
int main((int argc, char** argv)
{
    try{
        UserInterface interface;
        interface.run(argc, argv);
        return 0;
    } catch(std::exception& e) {
        std::cerr << e.what() << std::endl;
        return 1;
    }
}
```

Следовательно, при таком подходе все остальные классы создаются и стартуют уже из метода *run* класса *UserInterface*.

Рассмотрим другой вариант, когда все экземпляры классов создаются и выполняются из *main*:

```
int main((int argc, char** argv)
{
    try{
        UserInterface interface;
        auto params = interface.getParams(argc, argv);
        DataReader reader(params.inputFileName);
        auto data = reader.getData();
        Communicator comm(params.Address, params.Credentials);
        comm.connect();
        comm.auth();
        auto answer = comm.request(data);
        DataWriter writer(params.outputFileName);
        writer.save(answer);
        return 0;
    } catch(std::exception& e) {
        std::cerr << e.what() << std::endl;
        return 1;
    }
}
```

Здесь на функцию *main* уже возлагается задача по созданию экземпляров всех классов, а также вызов их методов.

3.3.2. Реализация пользовательского интерфейса командной строки

Для реализации пользовательского интерфейса командной строки можно использовать различные библиотеки. Например, можно использовать функции *getopt* и *getopt_long* из библиотеки POSIX языка Си. Она является стандартной для всех POSIX-совместимых операционных систем. Эти функции можно использовать как в программах на языке Си, так и на языке Си++.

В программах на языке Си++ можно также воспользоваться возможностями библиотеки *boost*, в которой есть специальный пакет *program_options*. Этот пакет существенно облегчает работу с интерфейсом командной строки. Однако так как *boost* – это библиотека, не входящая в стандартную библиотеку Си++, то она требует установки перед использованием.

Для обработки параметров командной строки можно вообще не использовать какие-либо библиотеки, а всю обработку сделать самостоятельно, применяя *argc* и *argv* функции *main*. Однако этот путь является сложным и весьма трудозатратным.

Рассмотрим пример использования библиотеки *boost* для параметров, приведенных в табл. 1 (подраздел 3.2). Для хранения значений параметров варианта использования «Выполнить вычисления» создадим специальную структуру:

```
struct Params {  
    std::string inFileName;  
    std::string userName;  
};
```

Так как в табл. 1 перечислены не все параметры, а только часть, то и в программе тоже использована только часть параметров.

Для того чтобы сократить имена в программе, введем сокращенное имя пространства имен:

```
namespace po = boost::program_options;
```

Далее создадим парсер и добавим в него параметры:

```
po::options_description desc("Allowed options");  
desc.add_options()  
    ("help,h", "Show help")  
    ("input,i",  
     po::value<std::string>(&params.inFileName)-  
>default_value("input.txt"),  
     "Set input file name")  
    ("user,u",  
     po::value<std::string>(&params.userName)->required(),  
     "Set user name");
```

Обратите внимание, что при добавлении параметров используется метод *add_options* со специфичным синтаксисом: описание каждого параметра заключено в круглые скобки и содержит два или три

элемента. Первый элемент – это имена параметров для длинной и короткой форм. Так, запись «user,u» означает, что в командной строке можно использовать как *-user*, так и *-u*. Второй элемент указывает на тип значения параметра. Если у параметра нет значения, то второй элемент в записи отсутствует. Третий элемент задает строку описания параметра в справке о программе. Теперь давайте рассмотрим подробнее второй элемент в описании параметра. Запись вида:

```
po::value<std::string>()
```

указывает на то, что значение параметра будет строкой типа *std::string*, которое сохраняется при парсинге в специальную переменную, являющуюся ассоциативным массивом, хранящим значения всех параметров после парсинга.

Другой возможный вариант:

```
po::value<std::string>(&params.inFileName)
```

позволяет указать любую переменную в программе, куда будет сохранено значение параметра. При этом значение параметра сохраняется также и в специальный ассоциативный массив.

Третий вариант позволяет не только сохранить значение в указанную переменную, но и указать значение по умолчанию, если данный параметр будет отсутствовать в командной строке:

```
po::value<std::string>(&params.inFileName)->default_value("input.txt")
```

Кроме того, можно указать, что параметр является обязательным:

```
po::value<std::string>(&params.userName)->required()
```

Далее надо объявить переменную для ассоциативного массива, куда будут сохраняться все результаты парсинга:

```
po::variables_map vm;
```

выполнить парсинг и сохранить результаты в ассоциативном массиве (и других переменных, если они были указаны):

```
po::store(po::parse_command_line(argc, argv, desc), vm);
```

здесь *argc* и *argv* – параметры функции *main*, через которые в Си++ стандартно осуществляется доступ к командной строке.

Если были использованы значения по умолчанию или задана нотификация пользовательских функций, то после сохранения результатов парсинга необходимо выполнить функцию нотификации, передав ей результаты парсинга:

```
po::notify(vm);
```

Эта функция возбуждает исключения при возникновении различных ошибок: неизвестный параметр, отсутствует обязательный параметр, у параметра нет значения т.п. Поэтому обработку параметра *--help* необходимо выполнять до вызова этой функции, иначе при наличии только *--help* в командной строке будем получать исключения по ошибкам.

Полный пример программы приведен в прил. А. Он состоит из модуля *interface*, в котором реализованы классы *Interface* и *Params*, и модуля *main*, в котором реализована функция *main*.

Конструктор класса *Interface* создает парсер. Метод *Parser* выполняет разбор параметров и возвращает *true*, когда разбор завершен успешно. Если в командной строке был указан параметр *--help*, то метод возвращает *false*. Метод *getParams* возвращает параметры, метод *getDescription* – строку с описанием параметров.

Функция *main* создает экземпляр класса *Interface*, вызывает метод *Parser*. Если он возвращает *true*, то печатает параметры, если *false* – печатает описание параметров. Также в функции *main* реализована обработка ошибок интерфейса.

3.3.3. Реализация сетевого взаимодействия

Алгоритм

В курсовой работе предлагается реализовать сетевое взаимодействие с использованием в качестве сетевого протокола *IP* версии 4, а в качестве транспортного – *TCP*.

Рассмотрим, как будет выглядеть сетевое взаимодействие для клиента и сервера, из каких фаз оно будет состоять и какие действия должны выполняться на каждой фазе.

Для клиента процедура сетевого взаимодействия линейна и состоит из следующих фаз:

- подготовка соединения;
- установка соединения;
- обмен данными в рамках установленного соединения;
- завершение соединения.

Более подробно сетевое взаимодействие со стороны клиента можно представить в виде алгоритмов, представленных на рис. 9. Слева на рисунке представлен общий алгоритм сетевого взаимодействия, а справа – алгоритм его основной части – обмена данными. Рассмотрим подробнее каждый блок этих алгоритмов.

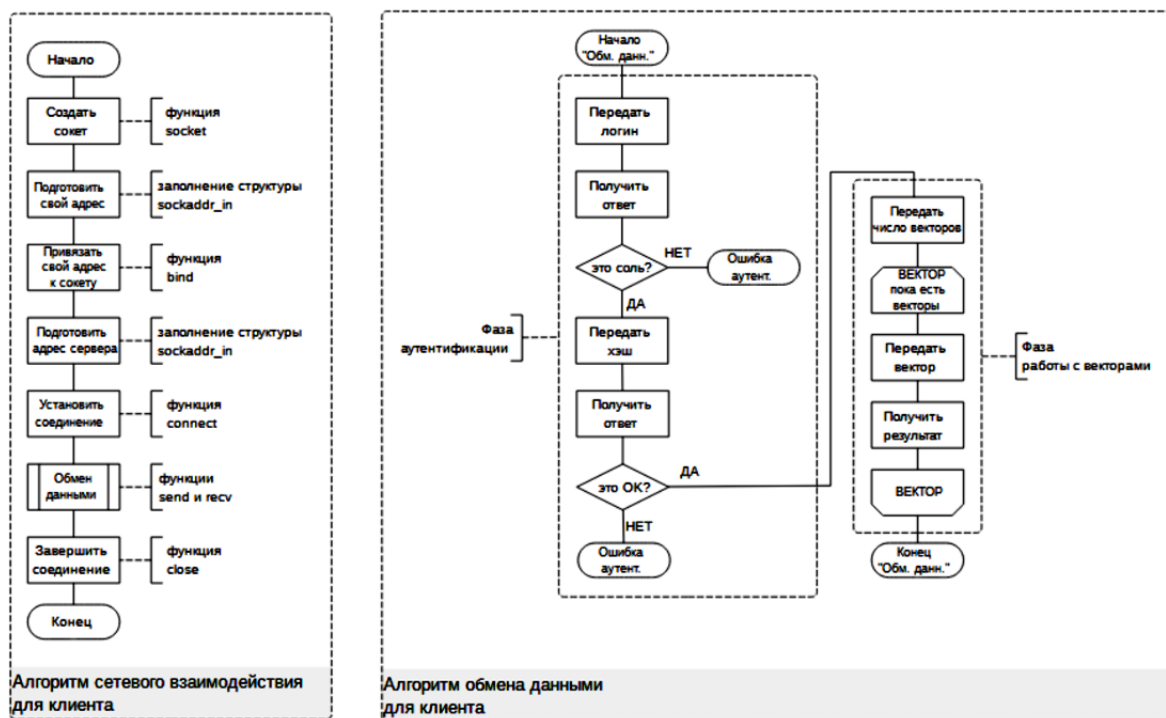


Рис. 9. Алгоритмы сетевого взаимодействия со стороны клиента

Создание сокета

В блоке «Создать сокет» создается сетевой сокет, который впоследствии будет использоваться для обмена данными. Это выполняется с помощью функции *socket*:

```
int s = socket(AF_INET, SOCK_STREAM, 0);
```

Первый параметр функции задает систему адресации, которую будет использовать сокет. В данном случае это константа *AF_INET*, которая означает использование адресов протокола *IP* версии 4 сети Интернет. Таким образом, адрес будет состоять из четырех байт (или 32 бит) и представляется значением типа *uint32_t*.

Второй параметр задает тип сокета. Так как у нас сокет будет использовать протокол *IP* версии 4, то в этом случае доступны всего два типа сокетов, задаваемых константами *SOCK_STREAM* и *SOCK_DGRAM*. Константа *SOCK_STREAM* соответствует транспорт-

ному протоколу *TCP*. Данный протокол предоставляет поток данных с предварительной установкой соединения, осуществляет повторный запрос данных в случае их потери и устраняет дублирование при получении двух копий одного пакета, гарантируя тем самым целостность передаваемых данных и уведомление отправителя о результатах передачи. Константа *SOCK_DGRAM* соответствует транспортному протоколу *UDP*. Данный протокол – это возможность передачи пакетов данных конечной длины (дейтаграмм) без установки соединения. Контроля потери пакетов или их дублирования не выполняется. Также не гарантируется получение пакетов в той последовательности, в которой они были переданы.

Функция возвращает целочисленное значение типа *int*. Если создание сокета прошло успешно, то возвращенное функцией значение является идентификатором сокета, который в дальнейшем будет использоваться для выполнения над ним различных операций. Если функция возвращает значение -1 , то это означает ошибку. Код ошибки находится в глобальной переменной *errno* стандартной библиотеки языка Си (заголовочный файл *errno.h*). В программах на языке Си++ эта переменная может быть использована для возбуждения системных исключений *std::system_error*:

```
int s = socket(AF_INET, SOCK_STREAM, 0);
if ( s == -1 )
    throw std::system_error(errno, std::generic_category());
```

Так как все сетевые функции являются функциями языка Си, то аналогичную проверку с возбуждением исключений в случае ошибки рекомендуется делать после каждого вызова сетевых функций, иначе код вашей программы будет небезопасен.

Для использования данной функции необходимо подключить заголовочный файл *sys/socket.h*.

Подготовка сетевых адресов

В блоке «Подготовить свой адрес» заполняется структура с сетевым адресом. Аналогичные действия выполняются и в блоке «Подготовить адрес сервера». Так как действия очень похожи, рассмотрим реализацию этих блоков одновременно.

В сетевой библиотеке есть структура *sockaddr*, которую можно назвать структурой для абстрактного адреса. Она объявлена так:

```
struct sockaddr {
    uint32_t    sa_family;    /* Address family */
    char        sa_data[];    /* Socket address */
};
```

Первое поле – это идентификатор используемой системы адресации, а второе поле – это байтовый массив неопределенной длины с адресной информацией. Дело в том, что адресная информация для различных систем адресации имеет разный размер. Поэтому в общем случае размер адреса неизвестен. Кроме того, адрес может состоять из набора полей различного типа, так что и типы данных адреса тоже неизвестны. Поэтому для каждой системы адресации имеется своя адресная структура, описывающая адрес конкретной адресной системы. Структура *sockaddr* является универсальным абстрактным типом, используемым в различных сетевых функциях. Такие сложности необходимы из-за того, что сетевая библиотека написана на языке Си, а не Си++.

В нашем случае для сетевого протокола *IP* используется конкретная адресная структура *sockaddr_in*, подключаемая через заголовочный файл *netinet/in.h*. Эта структура объявлена следующим образом:

```
struct sockaddr_in {
    uint32_t    sin_family;    /* всегда AF_INET */
    uint16_t    sin_port;      /* Номер порта */
    struct in_addr sin_addr;    /* IPv4 адрес */
};
```

Первое поле этой структуры всегда должно содержать константу *AF_INET*, второе поле – номер порта для протоколов *TCP* или *UDP*, и третье – сетевой адрес протокола *IP* версии 4. Как видим, третье поле в свою очередь тоже является структурой. Эта структура состоит всего из одного элемента, что выглядит немного странным, но так сложилось исторически:

```
struct in_addr {
    uint32_t s_addr;
};
```

Так как сетевой адрес задается структурой, то рекомендуется использовать для адресов динамические переменные, а не автоматические. Это связано с тем, что автоматические переменные создаются на стеке, и использование на стеке не скалярных переменных может

привести к его переполнению. Поэтому в курсовой работе следует использовать для адресной информации динамические переменные.

Рассмотрим пример создания адресной переменной и ее заполнение для случая собственного адреса:

```
sockaddr_in *self_addr = new sockaddr_in;  
self_addr->sin_family = AF_INET;    // всегда так  
self_addr->sin_port = 0;             // автовыбор  
self_addr->sin_addr.s_addr = 0;     // автовыбор
```

В данном примере создается динамическая переменная и ее адрес присваивается переменной-указателю *self_addr*. Первое поле всегда заполняется значением *AF_INET*.

Второе поле должно содержать номер порта. Когда заполняем структуру для собственного адреса, можно указать здесь нулевое значение. В этом случае порт будет выбран автоматически из имеющихся в системе свободных портов. Такая техника считается предпочтительной.

Третье поле тоже может содержать ноль. В случае заполнения собственного адреса клиента это тоже означает автоматический выбор наиболее удобного адреса из имеющихся в системе.

У динамических переменных есть один существенный недостаток – их обязательно надо освобождать оператором *delete* после окончания использования. В сложной программе с большим количеством ветвлений и возбуждаемых исключений можно запутаться и пропустить *delete* в одной из ветвей. Для исключения таких ситуаций рекомендуется использовать «умные указатели» библиотеки языка Си++, выполняющие освобождение динамических переменных при выходе из блока. Приведем для примера создание динамической структуры адреса с использованием обычного и умного указателей:

```
sockaddr_in *self_addr = new sockaddr_in;           //обычный указатель  
std::unique_ptr <sockaddr_in> self_addr(new sockaddr_in); //умный указатель
```

«Умные указатели» подключаются заголовочным файлом *memory* и являются шаблонными классами с перегруженными операторами, используемыми с обычными указателями. Поэтому в большинстве случаев при переходе от обычных к «умным» указателям переписывать весь код не придется. Исключение составляет только передача указателя в качестве параметра функции. Для этого в «ум-

ных указателях» есть метод *get*, возвращающий обычный указатель. Его использование рассмотрим позднее, когда перейдем к следующим блокам алгоритма.

Для адреса сервера адресная структура заполняется с указанием конкретных адреса и порта, на которых работает сервер. Предположим, что наш клиент должен подключиться к серверу, доступному по адресу 198.51.100.12 и порту 46913. Тогда заполнение адресной структуры будет выглядеть так:

```
std::unique_ptr <sockaddr_in> serv_addr(new sockaddr_in); // умный указатель
serv_addr->sin_family = AF_INET;           // всегда так
serv_addr->sin_port = htons(46913);        // конкретное значение
serv_addr->sin_addr.s_addr = inet_addr("198.51.100.12");//конкретное значение
```

Обратите внимание, что номер порта перед присваиванием преобразуется с помощью функции *htons* из компьютерного формата в сетевой формат. Для обратного преобразования номера порта из сетевого формата в компьютерный необходимо использовать функцию *ntohs*.

Для сетевого адреса используется функция *inet_addr*, преобразующая строку с IP-адресом в сетевой адрес. Если сетевой адрес задан числом, например, в нашем случае это будет 0xc633640c (198 → 0xc6, 51 → 0x33, 100 → 0x64, 12 → 0x0c), то тогда для его преобразования из компьютерного формата в сетевой понадобится функция *htonl*:

```
serv_addr->sin_addr.s_addr = htonl(0xc633640c);
```

Для обратного преобразования используется функция *ntohl*. Имена данных функций являются аббревиатурами, поэтому их легко запомнить, если знать полное название:

htons – *host to network short*;

ntohs – *network to host short*;

htonl – *host to network long*;

ntohl – *network to host long*,

где *short* означает 16-разрядные целые (номер порта), а *long* – 32-разрядные данные (IP-адрес).

Для использования функций преобразования адресов и портов необходимо подключить заголовочный файл *arpa/inet.h*.

Привязка адреса к сокету

В блоке «Привязать свой адрес к сокету» выполняется связывание сокета с адресной структурой с собственным адресом клиента. Это выполняется с помощью функции *bind*:

```
int rc = bind(s, (sockaddr*) self_addr.get(), sizeof(sockaddr_in));
```

Первым параметром этой функции является идентификатор сокета, который получен нами при его создании.

Второй параметр должен быть указателем на абстрактную адресную структуру *sockaddr*. Однако выражение *self_addr.get()* возвращает нам указатель на адресную структуру для протокола IP *sockaddr_in*. Для преобразования указателя в данном примере использовано преобразование указателя в стиле Си: *(sockaddr*)*. Если использовать подход Си++, то преобразование указателя необходимо выполнять с использованием специального шаблона и вместо

```
(sockaddr*) self_addr.get()
```

надо писать

```
reinterpret_cast<const sockaddr*>(self_addr.get())
```

Следует также обратить внимание на то, что у нас использован «умный указатель» и указатель на структуру *sockaddr_in*, из него необходимо получать с помощью метода *get*:

```
self_addr.get()
```

Третий параметр должен содержать значение размера адресной структуры. В данном примере для этих целей используется оператор *sizeof*:

```
sizeof(sockaddr_in)
```

В случае успешной привязки адреса к сокету функция возвращает нулевое значение. При ошибке функция возвращает -1 , а в переменную *errno* записывается код ошибки. В программах на Си++ рекомендуется возбуждать соответствующее исключение в случае ошибки, как было показано при описании работы функции *socket*.

Для использования данной функции необходимо подключить заголовочный файл *sys/socket.h*.

Установка соединения

Установка соединения выполняется функцией *connect*:

```
int rc = connect(s, (sockaddr*) serv_addr.get(), sizeof(sockaddr_in));
```

По набору параметров эта функция аналогична функции *bind*, за исключением того, что вторым параметром должен быть указатель на адресную структуру, содержащую адрес сервера.

Возвращаемое значение тоже аналогично возвращаемому значению функции *bind*. После успешного выполнения функции устанавливается соединение с сервером, в рамках которого клиент и сервер могут обмениваться данными произвольной длины.

Для использования данной функции необходимо подключить заголовочный файл *sys/socket.h*.

Обмен данными. Передача

Для обмена данными в рамках установленного соединения используются функции *send* и *recv*. Первая позволяет отправить данные, а вторая – принять.

Рассмотрим сначала передачу данных. Пример передачи текстовых данных, находящихся в строке:

```
std::string message("Lorem Ipsum");  
ssize_t rc = send(s, message.c_str(), message.length(), 0);
```

Первым параметром функции является идентификатор сокета. Для этого сокета предварительно должно быть установлено соединение.

Вторым параметром является указатель на передаваемые данные. Для строки можно его получить, используя методы *c_str* или *data*.

Третий параметр должен содержать размер передаваемых данных в байтах. Для строки его можно получить одним из методов *length* или *size*.

Четвертый параметр позволяет задать флаги, изменяющие логику работы функции. В нашем случае можно оставлять этот параметр нулевым.

Функция возвращает количество реально переданных байтов в случае успеха и -1 при ошибках. Код ошибки записывается в глобальную переменную *errno*. Следует отметить, что функция возвращает -1 только при локальных ошибках, т.е. ошибках, произошедших на стороне отправляющего. Если в функции указано слишком большое количество данных, которые не помещаются в буфер отправки, то возврат из функции блокируется до тех пор, пока все данные не будут отправлены и буфер не освободится.

Теперь рассмотрим случай с передачей двоичных данных из скалярной переменной, например типа *double*:

```
double data = 1.234567;  
ssize_t rc = send(s, &data, sizeof data, 0);
```

Здесь второй параметр функции получает значения адреса переменной, а третий – размера. При этом размер вычисляется оператором *sizeof* применительно к самой переменной, а не к ее типу. Такой подход гарантирует отсутствие ошибок при смене типа переменной.

Рассмотрим ситуацию передачи двоичных данных из обычного массива:

```
float data[] = {1.25, 2.36, 3.47, 4.58, 5.69};  
ssize_t rc = send(s, data, sizeof data, 0);
```

Здесь переменная массива *data* уже является указателем, а размер обычного массива может быть вычислен оператором *sizeof* аналогично размеру скалярной переменной.

Теперь рассмотрим передачу данных из динамического массива размером *LEN*. Сразу приведем пример с умным указателем:

```
std::unique_ptr<float[]> data(new float[LEN]{1.25, 2.36, 3.47, 4.58, 5.69});  
ssize_t rc = send(s, data.get(), LEN * sizeof data[0], 0);
```

Так как использован «умный указатель», то адрес передаваемых данных здесь возвращает метод *get* «умного указателя», а размер данных можно вычислить умножением размера массива *LEN* на размер элемента массива *sizeof data[0]*.

Если же надо передать содержимое вектора, то можно использовать такой код:

```
std::vector<uint32_t> data {13, 25435, 48, 32543654, 7};  
ssize_t rc = send(s, data.data(), data.size() * sizeof data[0], 0);
```

Здесь указатель на внутренний буфер вектора возвращает метод *data* вектора, а размер передаваемых данных получается произведением размера вектора на размер его элементов.

Кроме содержимого контейнеров *std::vector*, таким образом можно передавать и другие контейнеры, имеющие метод *data* для доступа к внутренним данным, например *std::array*. Контейнеры, не имеющие такого метода, передавать функцией *send* нельзя.

Обмен данными. Прием

Теперь рассмотрим прием данных, выполняемых функцией *recv*. Прием данных бывает блокирующий и неблокирующий.

При неблокирующем приеме функция *recv* запрашивает данные из буфера операционной системы и если они есть, то получает их сразу и возвращает число полученных байтов, а если нет – завершается с ошибкой и в переменную *errno* записывается код *EAGAIN* или *EWOULDBLOCK*. Таким образом, при неблокирующем приеме функция сразу возвращает управление, независимо от того, есть принятые данные или нет.

При блокирующем приеме если в буфере есть данные, то функция ведет себя так же, как и при неблокирующем. Если данных нет, то выполнение функции блокируется до тех пор, пока операционная система не получит данные через открытое соединение или пока соединение не будет закрыто. При использовании блокирующего приема следует предусмотреть возможность снятия блокировки, если данных с противоположной стороны нет длительное время. Установка тайм-аута в 3 мин для блокирующего приема выполняется следующим образом:

```
struct timeval timeout {180, 0};  
setsockopt(s, SOL_SOCKET, SO_RCVTIMEO, &timeout,  
sizeof(timeout));
```

где 180 – это количество секунд в таймауте; 0 – количество миллисекунд в таймауте. Таким образом, через 3 мин после отсутствия данных функция *recv* будет завершаться с ошибкой. В данной работе будем рассматривать только блокирующий прием, поэтому рекомендуется сразу после создания сокета установить для него таймаут снятия блокировки в несколько минут.

Подход к приему текста и двоичных данных обычно немного различается. При приеме текста, как правило, неизвестен размер принимаемых данных, а прием двоичных данных всегда соответствует некоторому оговоренному формату.

Рассмотрим прием двоичного значения скалярной переменной типа *double*:

```
double data; // скалярная переменная  
ssize_t rc = recv(s, &data, sizeof data, 0);
```

Первым параметром является идентификатор сокета. Вторым параметром указывает на область памяти, куда будут приняты данные.

Здесь это указатель на скалярную переменную. Третий параметр – это количество байтов, которые надо принять и записать по адресу, на который указывает второй параметр. Здесь это размер скалярной переменной.

Функция возвращает количество принятых и записанных байтов при успешном завершении и -1 при ошибке. Код ошибки записывается в глобальную переменную *errno*. Следует обратить внимание на то, что получение меньшего количества байтов, чем указано в третьем параметре, в данном случае свидетельствует о логической ошибке в алгоритме взаимодействия отправителя и получателя.

Теперь рассмотрим прием двоичного вектора. Для приема вектора необходимо использовать массив данных достаточного размера. Так как необходимый размер, как правило, становится известен только уже в ходе обмена данными между отправителем и получателем, то массив должен быть динамическим. Во избежание проблем с освобождением памяти для динамических массивов необходимо использовать «умные указатели». Рассмотрим пример, в котором сначала принимается скалярное значение с размером вектора, а затем и сам вектор данных типа *float*:

```
uint32_t vector_len; // здесь будет размер вектора
ssize_t rc = recv(s, &vector_len, sizeof vector_len, 0); // прием размера
// здесь должна быть обработка ошибок, но мы ее пропустим
std::unique_ptr<float[]> data(new float[vector_len]); // массив для вектора
rc = recv(s, data.get(), sizeof data[0] * vector_len, 0); // прием вектора
// здесь опять должна быть обработка ошибок, но мы ее снова пропустим
std::vector<float> v(data.get(), data.get() + vector_len); // массив в вектор
```

Как видим, во втором параметре функции использован метод *get* «умного указателя» для получения адреса буфера для приема данных. Размер буфера в третьем параметре получаем умножением размера одного элемента на их количество. В этом случае расхождение ожидаемого (третий параметр) и полученного (возвратила функция) количества байтов тоже сигнализирует о логической ошибке в алгоритме взаимодействия отправителя и получателя. Последняя строка примера показывает, что полученные данные из динамического массива можно использовать для создания вектора или любого другого контейнера Си++.

При приеме текстовых данных их размер, как правило, неизвестен, поэтому выделенной памяти для приема всех переданных дан-

ных может не хватить. Рассмотрим несколько вариантов решения задачи получения текстовых данных неизвестного размера.

Первый вариант самый простой, но ненадежный. Он заключается в выборе размера массива для приема данных достаточно большим, чтобы попытаться принять все данные, находящиеся в буфере приема. Однако не исключена вероятность, что такого размера все же окажется недостаточно. Это допустимо в студенческих работах, но недопустимо в реальных проектах.

Второй вариант заключается в двукратном увеличении массива для приема данных, пока все данные в него не поместятся. Здесь в функции *recv* используются различные флаги. Флаг *MSG_PEEK* позволяет получить данные не удаляя их из буфера приема, а флаг *MSG_TRUNC* позволяет удалить данные из буфера приема не получая их.

```
int buflen = 1024;                // начальный размер массива
std::unique_ptr<char[]> buf(new char[buflen]); // начальный массив
// цикл увеличения размера массива
while (true) {
    int rc = recv(s, buf.get(), buflen, MSG_PEEK); // принять данные
    if (rc < buflen) // данные уместились?
        break;      // да, выходим из цикла
    buflen *= 2;     // нет, увеличиваем размер массива вдвое
    buf = std::unique_ptr<char[]>(new char[buflen]); // и пересоздаем массив
}
std::string msg(buf.get(), rc);    // сохраняем массив в строку
recv(s, std::null_ptr, rc, MSG_TRUNC); // очищаем буфер приема
```

В приведенном фрагменте сперва создается массив размером 1024 байт. Далее в цикле выполняется прием данных в этот массив из буфера приема функцией *recv* с флагом *MSG_PEEK*. При этом буфер приема не очищается, что позволяет при необходимости повторно получить те же данные. Если количество принятых байтов равно размеру массива, то, значит, не все данные уместились в массив. Поэтому размер массива увеличивается вдвое и массив пересоздается. Если же количество принятых байтов меньше размера массива, то все данные приняты и цикл завершается. После этого данные из массива, если необходимо, копируются в строку. Затем буфер приема очищается функцией *recv* с флагом *MSG_TRUNC*, так как данные уже приняты и больше не нужны в буфере. В приведенном фрагменте отсутствует

обработка ошибок. При использовании его в программе обработку ошибок обязательно надо добавить.

Третий вариант заключается в дочитывании данных из буфера приема, не считанных с первого раза:

```
int buflen = 1024;                // начальный размер массива
std::unique_ptr<char[]> buf(new char[buflen]); // начальный массив
rc = recv(work_sock, buf.get(), buflen, 0); // принять данные
std::string res(buf.get(), rc);      // сохраняем массив в строку
if (rc == buflen) {                 // массив полон?
    int tail_size;                   // да
    ioctl(work_sock, FIONREAD, &tail_size); // узнаем остаток в буфере
    приема
    if (tail_size > 0) {              // остаток есть?
        if (tail_size > buflen) // да, остаток больше размера массива?
            // да, пересоздаем массив в размер остатка
            buf = std::unique_ptr<char[]>(new char[tail_size]);
        // нет, используем старый массив
        rc = recv(work_sock, buf.get(), tail_size, 0); // принять остаток
        res.append(buf.get(), rc); // добавляем остаток в строку
    }
}
```

Приведенным фрагментом создается массив размером 1024 байт и в него принимаются данные. Затем создается строка с принятыми данными. Если принятые данные заполнили весь массив, то определяется размер данных, оставшихся в буфере приема. Это делается с помощью вызова функции *ioctl* с параметром *FIONREAD*. Данная функция доступна после подключения *sys/ioctl.h*. Если в буфере приема еще остались данные, то они тоже принимаются в массив, а затем добавляются в строку. При этом если размер остатка больше массива, то массив пересоздается в размер остатка; если же остаток меньше, то данные принимаются в старый массив. В приведенном фрагменте отсутствует обработка ошибок. При использовании его в программе обработку ошибок обязательно надо добавить.

Для использования функций *send* и *recv* необходимо подключить заголовочный файл *sys/socket.h*.

Завершение соединения

Завершение соединения выполняется функцией *close*:

```
int rc = close(s);
```

Единственным параметром данной функции является идентификатор сокета завершаемого соединения. При успешном выполнении функция возвращает ноль, а при ошибке –1. Код ошибки сохраняется в переменной *errno*.

Для использования данной функции необходимо подключить заголовочный файл *unistd.h*.

Упрощенный пример реализации сетевого взаимодействия для клиента и сервера можно посмотреть в коде тестовых клиентов и сервера, приведенных в репозитории: <https://gitlab.com/winwood/timp.git>

3.3.4. Реализация подсистемы аутентификации с помощью библиотеки Crypto++

Схема аутентификации с использованием хеш-функции

Схема на основе хеш-функции может использоваться для аутентификации по открытому каналу, т.е. в ситуации, когда злоумышленник может перехватить содержимое сеанса аутентификации.

Рассмотрим ситуацию, когда необходимо, чтобы сервер аутентифицировал своих клиентов. Для этого необходимо, чтобы у клиента и сервера был общий секрет, который в дальнейшем будем называть *паролем*. Так как клиентов может быть много, то для их идентификации необходимо использовать уникальные идентификаторы. В дальнейшем идентификатор клиента будем называть *логином*. Таким образом, каждый клиент обладает парой значений «логин:пароль», в которой логин обладает свойством уникальности, а пароль – секретности. Сервер же должен хранить все пары «логин:пароль» своих клиентов. Набор этих пар будем называть *базой аутентификации*.

Так как логин не является секретом, то его можно передавать в открытом виде. Пароль же в открытом виде передавать нельзя, вместо него необходимо передавать значение криптографической хеш-функции от пароля. Свойства этой функции таковы, что по ее значению пароль можно найти только методом перебора всех возможных значений. При достаточной длине пароля и большом алфавите это может занять огромное время, несоизмеримое с интервалом смены паролей.

Однако простое использование хеш-функции от пароля имеет один существенный недостаток – значение хеш-функции будет всегда одинаковым. Таким образом, злоумышленник, перехватив это значение один раз, может использовать его в дальнейшем, даже не зная пароля. Поэтому необходимо обеспечить изменение значения хеш-функции для каждого сеанса аутентификации. Этого можно добиться,

используя каждый раз уникальное значение, которое далее будем называть *солью*. «Соль» добавляется к паролю, и уже от этой «смеси» вычисляется значение хеш-функции. Если для каждого сеанса аутентификации использовать новую «соль», то значение хеш-функции будет меняться от сеанса к сеансу при неизменном пароле.

Очевидно, что значение «соли» должны знать и клиент, и сервер. Так как значение «соли» не является секретом, то ее можно передавать в открытом виде. Формировать «соль» можно различным образом, например использовать в качестве «соли» значение текущего времени или получать ее от датчика случайных чисел. Перед использованием «соль» обычно переводят в текстовый вид либо представляют ее в виде шестнадцатеричного числа, либо используют специальное кодирование, например *base64*.

«Соль» может формироваться как клиентом, так и сервером. В зависимости от того, кто формирует «соль», схема аутентификации будет отличаться.

Рассмотрим сперва схему аутентификации, когда «соль» формирует клиент. В этом случае клиент формирует сообщение аутентификации следующим образом:

$$MSG_{AUTH} = LOGIN \parallel SALT_{16} \parallel HASH_{16} \quad (1)$$

$$HASH_{16} = \text{hash}(SALT_{16} \parallel PASSWORD) \quad (2)$$

где *LOGIN* – логин клиента; *PASSWORD* – пароль клиента; *SALT₁₆* – случайная «соль» в виде строки шестнадцатеричных цифр; *HASH₁₆* – значение хеш-функции от «соли» и пароля в виде строки шестнадцатеричных цифр; *MSG_{AUTH}* – сообщение аутентификации. Знак $\parallel$ обозначает конкатенацию строк.

Следует отметить, что такое сообщение вызывает трудности в выделении его частей на стороне сервера. Так как логин может иметь различную длину, то для однозначного выделения каждой части сообщения сервером необходимо, чтобы «соль» и хеш имели фиксированную длину. Тогда длина пароля может быть получена следующим образом:

$$\text{Length}_{LOGIN} = \text{Length}_{MSG} - \text{Length}_{SALT} - \text{Length}_{HASH} \quad (3)$$

Аутентификация в этом случае состоит из двух фаз. На первой фазе клиент формирует сообщение аутентификации по формуле (1) и отправляет его серверу. На второй фазе сервер посылает клиенту результат аутентификации. Для этого сервер из полученного сообще-

ния выделяет логин, «соль» и хеш, используя (3). Далее в базе аутентификации ищется логин клиента. Если логина в базе нет, то сервер посылает клиенту сообщение *ERR* и разрывает соединение.

Если логин есть, то из базы аутентификации извлекается пароль и вычисляется хеш, используя (2). Затем сравнивается вычисленный хеш с хешем, выделенным из сообщения аутентификации. В случае их совпадения аутентификация считается успешной, сервер посылает клиенту сообщение *OK*, и они продолжают взаимодействие. Если хеши не совпали, то аутентификация считается неуспешной, сервер посылает сообщение *ERR* и разрывает соединение. На рис. 10 приведена рассмотренная схема аутентификации.

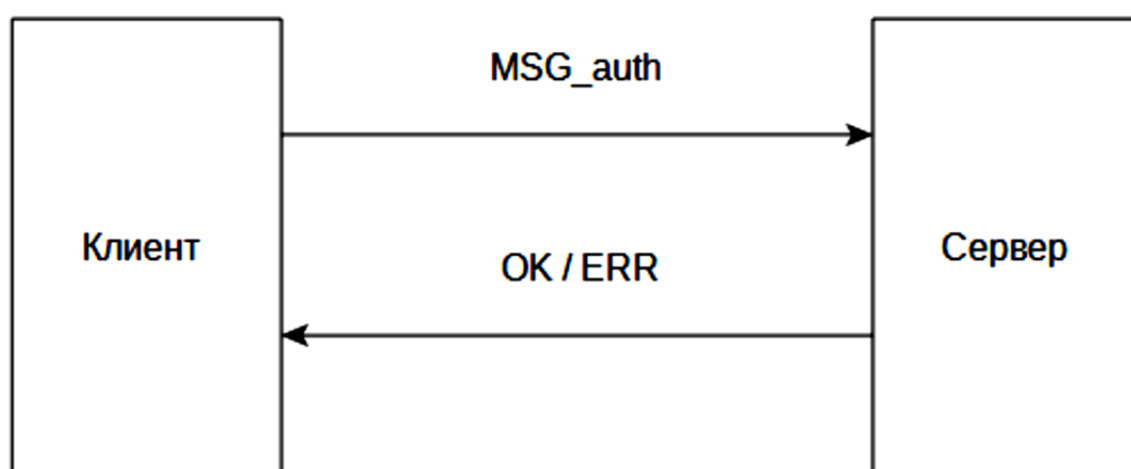


Рис. 10. Схема аутентификации при формировании «соли» клиентом

Если формирование «соли» будет выполнять сервер, то тогда аутентификация будет состоять из четырех фаз. На первой фазе клиент отправляет серверу свой логин. На второй фазе сервер ищет в базе аутентификации пароль клиента. Если логина в базе нет, то сервер посылает клиенту сообщение *ERR* и разрывает соединение. Если логин есть, то сервер формирует «соль» и отправляет ее клиенту. На третьей фазе клиент вычисляет хеш от «соли» и пароля, используя (2), и отправляет серверу. На последней фазе сервер извлекает из базы аутентификации пароль клиента и вычисляет хеш, используя (2). Затем вычисленный хеш сравнивается с полученным от клиента. В случае их совпадения аутентификация считается успешной, сервер посылает клиенту сообщение *OK*, и они продолжают взаимодействие. Если хеши не совпали, то аутентификация считается неуспешной, сервер посылает сообщение *ERR* и разрывает соединение. На рис. 11 приведена схема аутентификации при формировании «соли» сервером.

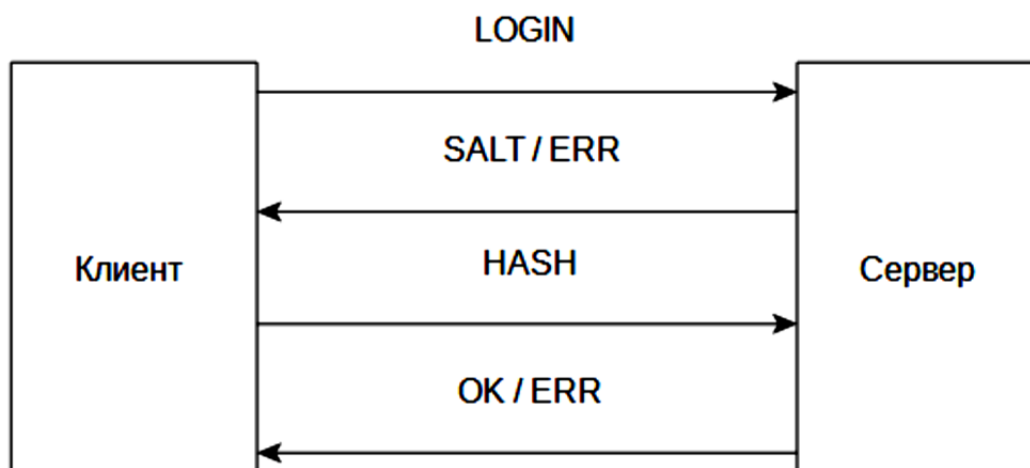


Рис. 11. Схема аутентификации при формировании «соли» сервером

В прил. Б приведены примеры программ формирования «соли», хеша и проверки хеша. Все программы используют библиотеку Crypto++.

Первая программа формирует «соль» в виде массива байтов заданного размера с помощью генератора псевдослучайной последовательности и преобразует ее в строку шестнадцатеричных цифр. Для генерации «соли» используется метод *GenerateBlock* класса *AutoSeededRandomPool*. Для преобразования «соли» в строку шестнадцатеричных цифр используется специальный класс-фильтр *HexEncoder*. За передачу данных в этот фильтр отвечает класс *ArraySource*, параметрами которого являются указатель на массив байт с данными, размер данных и адрес объекта-трансформатора, куда надо передать данные (в нашем случае это *HexEncoder*). Из фильтра *HexEncoder* данные передаются в специальный объект *StringSink*, который сохраняет результат в строке.

Вторая программа для заданных «соли» и пароля вычисляет хеш-функцию *sha256* и преобразует ее в строку шестнадцатеричных цифр. Для формирования хеша используется класс *SHA256*, реализующий хеш-функцию *sha256*. Объект *StringSource* принимает на вход строку со сцепленными «солью» и паролем. Затем передает эту строку классу-фильтру *HashFilter*, который выполняет хеширование. Первым параметром *HashFilter* является объект класса *SHA256*, который и задает алгоритм хеширования. Для смены хеш-функции достаточно в данный фильтр передать объект другого класса. Результат хеширования передается в класс-фильтр *HexEncoder*, который преобразует двоичное значение в строку шестнадцатеричных цифр. Наконец, результат кодирования сохраняется в строке с помощью класса *StringSink*.

Третья программа выполняет проверку хеша. Несмотря на то, что проверку можно было сделать простым сравнением полученного и вычисленного хеша (пример вычисления показан в предыдущей программе), для проверки используется специальный класс-фильтр *HashVerificationFilter*. Это связано с тем, что прямая проверка двух хешей на равенство выполняется быстрее при несовпадении хешей и медленнее при совпадении. Это позволяет провести на систему аутентификации так называемую «атаку по времени». Поэтому проверка хеша выполняется всегда специальной функцией, время работы которой не зависит от результата проверки.

Так как хеш задан у нас в виде строки шестнадцатеричных цифр, то перед проверкой его надо преобразовать в двоичный вид. Это делается с помощью класса-фильтра *HexDecoder*. Затем хеш сцепляется с «солью» и паролем и передается классу-фильтру *HashVerificationFilter*. В качестве первого параметра данный фильтр получает объект класса *SHA256*, определяющий алгоритм хеширования. Результат работы этого фильтра является булевским значением *true* или *false* в зависимости от того, совпали хеши или нет. Для сохранения результата используется класс *ArraySink*, для которого булевская переменная представляется в виде байтового массива.

Схема аутентификации с использованием HMAC

Код аутентификации сообщений *HMAC* (*hash-based message authentication code*) был разработан как средство обеспечения целостности сообщений. Его также можно использовать для аутентификации, если в качестве ключа использовать пароль, а в качестве сообщения – конкатенацию логина и «соли». «Соль» здесь нужна, чтобы сообщение всегда было уникальным. Таким образом, клиент при использовании *HMAC* будет формировать сообщение аутентификации следующим образом:

$$\text{MSG}_{\text{AUTH}} = \text{MSG} \parallel \text{HMAC}_{16} \quad (4)$$

$$\text{MSG} = \text{LOGIN} \parallel \text{SALT}_{16} \quad (5)$$

$$\text{HMAC}_{16} = \text{hmac}(\text{PASSWORD}, \text{MSG}) \quad (6)$$

где $\text{LOGIN} \parallel \text{SALT}_{16}$ – сообщение (5); HMAC_{16} – подпись этого сообщения, которая формируется с помощью функции *hmac*, как показано в (6). Функция на вход принимает два параметра: пароль и сообщение. Для формирования подписи используется один из алгоритмов хеширования. Сообщение аутентификации состоит из сообщения и его подписи (4).

Сообщение аутентификации формируется клиентом и отправляется серверу. Сервер, получив сообщение аутентификации, разделяет его на сообщение и подпись, а затем из сообщения выделяет логин. Далее в базе аутентификации определяется логин клиента. Если логина в базе нет, то сервер посылает клиенту сообщение *ERR* и разрывает соединение.

Если логин есть, то из базы аутентификации извлекается пароль и по формулам (5) и (6) осуществляется вычисление подписи сообщения. Далее вычисленная подпись сравнивается с принятой подписью сообщения. В случае их совпадения аутентификация считается успешной, сервер посылает клиенту сообщение *OK*, и они продолжают взаимодействие. Если хеши не совпали, то аутентификация считается неуспешной, сервер посылает сообщение *ERR* и разрывает соединение. В целом аутентификация проходит в две фазы аналогично аутентификации, приведенной на рис. 10, только на первой фазе передается сообщение, определяемое формулой (4), а не формулой (1).

В прил. В приведены примеры программ для формирования и проверки подписи. Они имеют незначительные отличия от аналогичных программ по формированию и проверке хеша из прил. Б, которые подробно рассмотрены ранее. Формирование «соли» для всех случаев идентично, и пример ее определения приведен в прил. Б.

База аутентификации

Метод хранения и формат базы аутентификации могут быть любыми. Рассмотрим простейший пример, когда база аутентификации хранится в тестовом файле, в котором каждая пара логина и пароля записаны в отдельной строке и разделены двоеточием. Содержимое файла с базой аутентификации, например, может быть таким:

```
ibst128:SupErP@$w0rd  
valera_ivanov:ow8ahra8Uv  
masha_medvedeva:EekohH3she
```

Для чтения данной базы в программе можно использовать как стандартную библиотеку Си++. Прочитать базу и выделить из нее логины и пароли с использованием стандартной библиотеки можно так:

```
std::string line;  
std::ifstream auth_file("auth_base.txt");  
while(auth_file >> line) {
```

```

size_t pos = line.find(':');
std::string login = line.substr(0, pos);
std::string pass = line.substr(pos + 1);
std::cout << login << " has password " << pass << std::endl;
}

```

3.4. Тестирование программы

3.4.1. Модульное тестирование

Рассмотрим проектирование и реализацию модульных тестов с помощью библиотеки `UnitTest++`.

При проектировании классов сразу необходимо учитывать их тестируемость. Например, для тестирования конструктора класса может возникнуть необходимость проверки устанавливаемых им приватных атрибутов. Тогда следует предусмотреть набор публичных методов, возвращающих значения таких атрибутов. Пусть в классе *Foo* есть приватный атрибут *bar*, определение значения которому надо протестировать. Тогда необходимо сделать метод, возвращающий данный атрибут:

```

class Foo {
private:
    std::string bar;
    // тут что-то еще приватное
public:
    // тут что-то еще публичное
    std::string getBar() { return bar;}
};

```

Теперь рассмотрим, как можно выполнить модульное тестирование в тех или иных ситуациях.

Тестирование пользовательского интерфейса

Рассмотрим тестирование примера, описанного в 3.3.2 и приведенного в прил. А. Сначала составим тестовые сценарии. В соответствии с проектом интерфейса, приведенным в подразделе 3.2, нам необходимо реализовать два позитивных сценария: тестирование параметра *help* и тестирование рабочих параметров.

Первый сценарий будет состоять из четыре тестов: тестирование *-h*, тестирование *--help* и тестирование одного из этих параметров в комбинации с любым другим, причем тестов с другим параметром должно быть два, с перестановкой параметров местами.

Второй сценарий должен проверять рабочие параметры *input* и *user*, причем в коротком и длинном вариантах, с умолчанием и без него, с перестановкой местами. Количество вариантов можно сократить, тестируя на разных перестановках либо длинные, либо короткие варианты.

Третий сценарий должен быть негативным. В нем должны быть тесты на возникновение ошибок: на полное отсутствие параметров, на отсутствие обязательных параметров, на присутствие лишних параметров. Тестовые сценарии для модуля *interface* приведены в табл. 2.

Таблица 2

**Тестовые сценарии для модульного тестирования
пользовательского интерфейса**

Сценарий	Тип	Тест	Параметры	Ожидаемый результат
Тестирование справки	Позитивный	1.1	-h	Вывод справки
		1.2	--help	Вывод справки
		1.3	-h -u vasya	Вывод справки
		1.4	-u vasya -h	Вывод справки
Тестирование рабочих параметров	Позитивный	2.1	-u vasya	InFileName = input.txt userName = vasya
		2.2	--user vasya	InFileName = input.txt userName = vasya
		2.3	-u vasya -i a.txt	InFileName = a.txt userName = vasya
		2.4	--input a.txt -u x	InFileName = a.txt userName = x
Тестирование исключений	Негативный	3.1	Нет параметров	po::required_option
		3.2	-u	po::invalid_command_line_syntax
		3.3	-i a.txt	po::required_option
		3.4	-i a.txt -v	po::unknown_option

Программа, реализующая данные сценарии, приведена в прил. Г, а код тестируемого модуля – в прил. А. Рассмотрим подробнее по одному тесту из каждого сценария. В тестировании справки рассмотрим тест на дополнительный параметр вместе с параметром справки:

```
TEST(HelpAndUser) {
    Interface iface;
```

```

const char* argv[] = {"test", "-h", "-u", "vasya", nullptr};
int argc = sizeof argv / sizeof nullptr - 1;
REQUIRE CHECK(!iface.Parser(argc, argv));
CHECK(!iface.getDescription().empty());
}

```

Здесь параметры командной строки формируются в виде массива указателей на строки в переменной *argv*, а количество параметров записывается в *argc*. Обратите внимание, что первая строка в массиве не имеет отношения к параметрам и может иметь любое значение, а последний указатель в массиве нулевой. В переменную *argc* записывается количество элементов в массиве, при этом последний нулевой указатель не учитывается. Сначала выполняется проверка, что *Parser* вернул *false*. Если это так, то выполняется следующая проверка; если нет – то тест провален. Если первая проверка прошла, то дальше проверяется, что строка описания непустая. На содержимое строки описания проверка не делается, так как в процессе разработки описание может меняться. Окончательную проверку строки описания выполняют вручную при функциональном тестировании.

Теперь рассмотрим один тест рабочих параметров:

```

TEST(UserAndFile) {
    Interface iface;
    std::string user("vasya");
    std::string file("a.txt");
    const char* argv[] = {"test",
                          "-u", user.c_str(),
                          "-i", file.c_str(),
                          nullptr};
    int argc = sizeof argv / sizeof nullptr - 1;
    REQUIRE CHECK(iface.Parser(argc, argv));
    CHECK_EQUAL(iface.getParams().userName, user);
    CHECK_EQUAL(iface.getParams().inFileName, file);
}

```

Здесь появились две строки *user* и *file* с именами пользователя и файла соответственно. Эти переменные используются и как значения при формировании параметров для теста, и как образцы при проверке возвращенных значений. Сначала проверяется, что *Parser* вернул *true*, и лишь затем проверяются сами параметры.

В сценарии проверки исключений проверяется, что при вызове *Parser* возбуждается исключение нужного типа. Обратите внимание,

что у исключения вместо пространства имен *boost::program_options* используется пространство имен *po*. Это обеспечивает строка *namespace po = boost::program_options;* в заголовочном файле модуля *interface*.

Тестирование модулей работы с файлами

При тестировании модулей, которые читают файлы, необходимо в тесте сначала эти файлы сформировать и затем тестировать. Рассмотрим пример тестирования класса *Reader*, предназначенного для чтения данных из двоичного файла. Объявление класса имеет следующий вид:

```
class Reader {
private:
    std::ifstream f;
public:
    Reader() = delete;
    Reader(const char * name);
    Reader(const std::string& name);
    BinaryData read();
};
```

Здесь тип данных *BinaryData* представляет собой вектор, состоящий из векторов чисел типа *double* и объявленный так:

```
typedef std::vector<std::vector<double>> BinaryData;
```

В данном классе конструктор открывает указанный в параметрах файл, а метод *read* читает данные и возвращает их в виде вектора векторов. Кроме того, в конструкторе для открытого файла включается режим возбуждения исключений, поэтому при любых ошибках в работе с файлом будут возбуждаться исключения типа *std::ios_base::failure*.

Для проверки класса достаточно двух тестовых сценариев: позитивного — для чтения корректного файла и негативного — для проверки ситуаций, когда файл отсутствует и в нем нарушена структура данных. Самым простым будет тест на отсутствие файла:

```
TEST(NoFileTest) {
    CHECK_THROW(Reader r("notfound.bin"), std::ios_base::failure);
}
```

Здесь нет необходимости в подготовке данных, достаточно вызвать конструктор с именем несуществующего файла и проверить возбуждение исключения.

Для других тестов потребуется предварительно создавать файлы с заданным содержимым, а затем их читать и сравнивать записанное с прочитанным. Создание файла можно выполнить с помощью специального класса, который потом можно указать как тестовую фикстуру.

Рассмотрим еще один тест из негативного сценария – чтение пустого файла. Для создания пустого файла используем такой класс-фикстуру:

```
struct EmptyFile {  
    EmptyFile() {  
        std::ofstream f(FILE_NAME);  
    }  
};
```

Здесь конструктор данного класса создает файл и ничего в него не пишет. После завершения конструктора получим пустой файл с заданным именем. Имя файла задается с помощью директивы препроцессора `#define FILE_NAME "data.bin"`.

Для проверки используем следующий тест:

```
TEST_FIXTURE(EmptyFile, EmptyFileTest) {  
    Reader r(FILE_NAME);  
    CHECK_THROW(r.read(), std::ios_base::failure);  
}
```

Перед выполнением теста будет вызван конструктор класса *EmptyFile*, который создаст пустой файл. В тесте для проверки возбуждения исключения выполняется попытка чтения из этого файла.

Теперь рассмотрим тест из позитивного сценария, проверяющего прочитанные данные. Для этого используем класс-фикстуру, который не только создает файл, но и записывает в него данные. Вот пример такого класса:

```
struct DataVectors {  
    BinaryData writed_data{{1.0, 2.0, 3.0}, {-1.2, -3.3}, {2.234E12}};  
    DataVectors() {  
        uint32_t size = writed_data.size();  
        char* size_ptr = reinterpret_cast<char*>(&size);  
    }  
};
```

```

std::ofstream f(FILE_NAME);
f.write(size_ptr, sizeof size);
for(auto vect : writed_data) {
    size = vect.size();
    f.write(size_ptr, sizeof size);
    for(auto data : vect)
        f.write(reinterpret_cast<char*>(&data), sizeof data);
}
}
};

```

Атрибут *writed_data* содержит значения векторов, которые надо записать. Всего три вектора с тремя, двумя и одним значением соответственно. В конструкторе создается файл с указанным именем, а затем в него записывается содержимое вектора. Тест в этом случае будет выглядеть так:

```

TEST_FIXTURE(DataVectors, DataVectorsTest) {
    Reader r(FILE_NAME);
    auto readed_data = r.read();
    CHECK(writed_data == readed_data);
}

```

Перед выполнением теста, будет вызван конструктор класса *DataVectors*, а его атрибут *writed_data* будет доступен в тесте как простая переменная. Обратите внимание, что для сравнения используется не *CHECK_EQUAL*, а *CHECK*. Дело в том, что в случае ошибки *CHECK_EQUAL* выводит значения несовпавших элементов, а в данном случае это невозможно, так как для векторов не перегружен оператор вывода.

Кроме рассмотренного теста, в позитивном сценарии желательно рассмотреть случаи, когда в файле ноль векторов (не путать с пустым файлом) и когда в файле несколько векторов, но все они пустые. С технической точки зрения такие файлы корректны и не должны приводить к ошибкам чтения. Классы-фикстуры для таких тестов легко получить из рассмотренного выше класса *DataVectors* путем изменения значений атрибута *writed_data*, а сам тест будет отличаться только заголовком, в котором указаны фикстура и имя теста.

Из класса *DataVectors* также можно легко сделать класс для негативного сценария для проверки на чтение файла с неверной структурой. Для этого можно увеличить записываемое в файл значение количества векторов, добавив к ним единицу:

```
uint32_t size = writed_data.size() + 1;
```

Это приведет при чтении к попытке читать данные после достижения конца файла и, как следствие, к возбуждению исключения.

При тестировании классов, предназначенных для записи данных, ситуация обратная – в тесте необходимо читать данные и сравнивать их с записанными. Однако воспользоваться фикстурой не получится, так как она выполняется перед тестом, а читать данные надо уже в процессе теста, непосредственно перед сравнением. Таким образом, код для чтения файла необходимо помещать непосредственно в тест.

Тестирование коммуникационного модуля и функций аутентификации

Коммуникационный модуль выполняет сетевое взаимодействие с удаленной стороной, поэтому полноценное модульное тестирование для него невыполнимо. Тестирование сетевых функций, как правило, выполняется во время интеграционного тестирования. В данной курсовой работе тестирование коммуникационного модуля предполагается при функциональном тестировании.

Однако часть функционала коммуникационного модуля, отвечающего за аутентификацию, необходимо охватить модульными тестами. Поэтому функции аутентификации следует либо вынести в отдельный класс, либо реализовать в виде статических методов класса, отвечающего за сетевое взаимодействие. При этом модульные тесты будут сильно зависеть от выбранной схемы аутентификации. При этом значения хеш-функции или подписи следует сформировать заранее с использованием внешних программ. Наиболее удобной программой для формирования тестовых значений является openssl.

Функцию формирования «соли» необходимо тестировать на различие двух (или нескольких) последовательных значений.

Функцию формирования хеша из пароля и «соли» следует тестировать для фиксированных значений «соли» и пароля. Например, если «соль» будет `4F9C429F5C6884DB726445BE`, а пароль `SupErP@$$w0rd`, то тогда строка для хеширования согласно формуле (2) будет `4F9C429F5C6884DB726445BESupErP@$$w0rd`. Для формирования хеша можно воспользоваться следующей командой:

```
echo -n '4F9C429F5C6884DB726445BESupErP@$$w0rd' |openssl  
sha256 |tr a-z A-Z
```

Результатом работы будет вычисленный хеш:

```
9C67ABCFAD8E46E78F89173497800E5EE143D67C28765A60C94722  
1E7114FF68
```

Именно это значение можно использовать в модульном тесте в качестве ожидаемого значения. Следует отметить, что строки в командах следует обязательно заключать в одиночные кавычки, чтобы избежать подстановок командного интерпретатора.

Функцию формирования подписи тестируют аналогичным образом. Сообщение формируется из пароля по формуле (5), а подпись – по формуле (6). Для логина *ibst128*, «соли» *4F9C429F5C6884DB726445BE* и пароля *SupErP@\$\$w0rd* команда формирования подписи будет следующей:

```
echo -n 'ibst1284F9C429F5C6884DB726445BE' |\nopenssl mac -digest sha256 -macopt key:'SupErP@$$w0rd' HMAC
```

Результат ее выполнения будет таким:

```
265FC6E8247256158D9FBD6CE1A805BF4CDB7CF2230268591F69C  
D9A03E4A3D8
```

В примерах использована хеш-функция *sha256*. Если необходимо использовать другую функцию, то ее имя нужно указать в командах вместо *sha256*.

Тестирование функций проверки хеша и подписи сводится к проверке возвращаемого значения на *true* при успешной проверке и *false* – при неуспешной.

Тестирование серверных модулей

У сервера в отличие от клиента должны быть модуль вычисления и модуль работы с базой аутентификации.

Тестирование модуля работы с базой аутентификации должны включать как тесты на наличие файла базы и чтение данных из него, так и тесты на поиск аутентификационных данных по предъявленному логину. Построение таких тестов принципиально не отличается от тестов на чтение данных, рассмотренных ранее. Здесь в тестах также надо сперва формировать файл с базой аутентификационных данных, а потом читать его.

Тестирование модуля вычислителя заключается в предварительном формировании различных наборов данных и вычислении образцов

вых значений. Эти вычисления можно выполнить с использованием, например, интерпретатора языка Питон. Особое внимание следует обратить на ситуации целочисленного переполнения при вычислениях.

При тестировании значений с плавающей точкой необходимо использовать не проверку на равенство, а проверку на допустимое значение разницы между образцом и полученным значением.

3.4.2. Функциональное тестирование

Функциональное тестирование должно проверять соответствие функций программы требованиям технического задания. Кроме того, на этапе функционального тестирования должна быть проверена работа классов и функций, не затронутых модульным тестированием, а также те результаты работы программы, которые не поддаются автоматизации, например, выводимые текстовые сообщения.

Для ручных тестов в тестовых сценариях указываются:

- название теста;
- тип теста (позитивный/негативный);
- краткое описание;
- предусловия выполнения теста (что должно быть выполнено после старта программы для возможности выполнения теста);
- входные данные;
- ожидаемый результат;
- постусловие выполнения теста (что должно быть сделано после завершения теста для возможности проведения дальнейшего тестирования).

Для программы-клиента функциональное тестирование может содержать такие сценарии, как:

- проверка текстовых сообщений справки программы;
- проверка текстовых сообщений при передаче программе неверных параметров;
- проверка текстовых сообщений при ошибках чтения и записи данных;
- проверка текстовых сообщений при ошибках установки соединения с сервером, ошибках аутентификации, ошибках при приеме/передаче данных;
- проверка работы программы при всех корректных входных значениях.

Так как тестовые сценарии при функциональном тестировании достаточно объемные, то каждый сценарий можно представить в виде отдельной таблицы. В табл. 3 приведен пример одного такого сценария.

**Пример тестового сценария проверки сообщения
об ошибке при чтении файла**

Название	Ошибка чтения данных
Тип	Негативный
Описание	Проверка сообщения об ошибке при попытке чтения файла нулевого размера
Предусловие	Подготовить файл с именем zero.bin нулевого размера в папке программы
Входные данные	Запустить программу командой ./client -i zero.bin -u ibst128 -o result.txt -a localhost -p 43184
Ожидаемый результат	Сообщение: File : zero.bin Read error: basic_ios::clear: iostream error
Постусловие	Удалить файл zero.bin в папке программы
Полученный результат	Сообщение: File : zero.bin Read error: basic_ios::clear: iostream error
Итог	Пройден

Последние две строки таблицы заполняются тестирующим после выполнения теста. Если использовать для хранения информации по тестовым сценариям в электронной таблице, например, *LibreOffice Calc*, то тогда можно автоматически получать общий результат тестирования за счет обработки строки «Итог» по всем тестам. Кроме того, можно автоматизировать выполнение самих тестов, используя язык сценариев *bash* в *linux*. Так, для теста, приведенного в табл. 3, можно предложить такой сценарий:

```
FILE='zero.bin'
LOG='test.log'
echo '=====' >> $LOG
echo 'Тест: Ошибка чтения данных' >> $LOG
touch $FILE
./client -i $FILE -u ibst128 -o result.txt -a localhost -p 43184 2>> $LOG
rm $FILE
echo '=====' >> $LOG
```

В приведенном сценарии вся информация, выводимая тестируемой программой в поток *std::cerr*, перенаправляется в файл *test.log*. Туда же записывается и название теста. Если для каждого теста написать такой сценарий, а затем все их выполнить, то получим файл с отчетом по каждому тесту. Тогда тестировщику необходимо будет только просмотреть отчет и записать результаты теста в таблицу. Использование сценариев позволяет исключить ошибки при выполнении предусловий, постусловий и запуске программы при многократном прогоне тестов.

Тестирование программы-сервера будет иметь несколько особенностей.

Во-первых, сообщения об ошибках сервер должен выводить в специальный файл журнала, а не в поток *std::cerr*.

Во-вторых, при некритических ошибках сервер должен восстанавливать работоспособное состояние и продолжать работу. Например, при ошибках в сетевом соединении с клиентом сервер должен закрыть это соединение и быть готовым к приему новых соединений от других клиентов.

В-третьих, функциональные тесты сервера должны включать тестирование устойчивости, т.е. сервер должен продолжать работать после последовательного обслуживания большого количества клиентов, в том числе и при ошибках соединения с некоторыми из них.

В-четвертых, функциональные тесты должны включать нагрузочное тестирование, когда к серверу подключается (или пытается подключиться) одновременно большое количество клиентов.

В-пятых, функциональные тесты должны включать объемное тестирование, когда клиенты передают серверу огромные массивы данных.

Тестирование устойчивости должно обязательно проводиться в курсовой работе. Нагрузочное тестирование не обязательно, но желательно. Объемное тестирование может проводиться при наличии данной функции в тестовом клиенте.

3.5. Документирование программы

Генератор документации Doxygen. Общие сведения

Doxygen является стандартным инструментом для создания документации из аннотированных исходных текстов C++, но он также поддерживает другие популярные языки программирования, такие как C, Objective-C, C#, PHP, Java, Python, IDL (варианты Corba, Microsoft и UNO/ OpenOffice), Fortran и в некоторой степени D. Doxy-

gen также поддерживает язык описания аппаратного обеспечения VHDL.

Doxygen способен осуществлять генерацию онлайн-браузера документации в формате HTML и/или автономное справочное руководство в формате Latex из набора документированных файлов исходного кода.

Doxygen поддерживает генерацию выходных данных в форматах:

- RTF (MS-Word);
- PostScript;
- XML;
- PDF с гиперссылками;
- HTML;
- сжатый HTML;
- справочная страница Unix.

Документация извлекается непосредственно из исходного кода, что значительно облегчает приведение ее в соответствие с исходным кодом.

Также Doxygen возможно использовать для извлечения структуры кода из недокументированных исходных файлов. Это очень полезно, чтобы быстро ориентироваться в больших дистрибутивах с исходным кодом. Doxygen также может визуализировать отношения между различными элементами с помощью графиков зависимостей, диаграмм наследования и диаграмм совместной работы, которые генерируются автоматически.

Исполняемый файл Doxygen является основной программой, которая непосредственно анализирует исходный код и генерирует документацию. Опционально возможно использовать исполняемый файл doxuwizard, который представляет собой графический интерфейс для редактирования файла конфигурации, используемого doxygen, и для запуска doxygen в графической среде.

На рис. 12 показана взаимосвязь между инструментами и потоками информации между ними.

Файл конфигурации Doxygen

Doxygen использует конфигурационный файл для определения всех своих параметров. Каждый проект должен иметь свой собственный конфигурационный файл. Проект может состоять из одного исходного файла, но также может представлять собой целое дерево исходных файлов, которое рекурсивно сканируется.

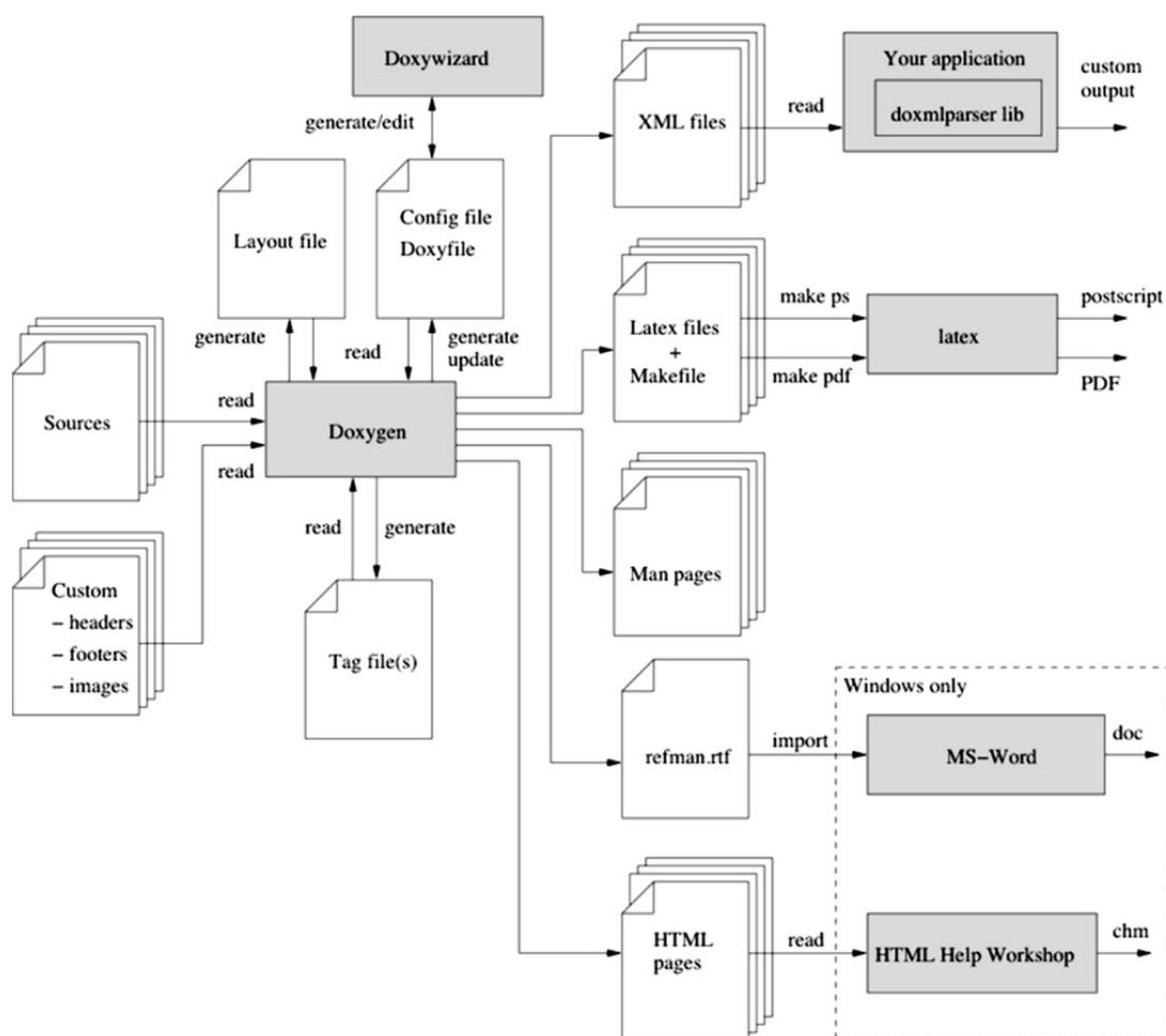


Рис. 12. Архитектура Doxygen

Чтобы упростить создание файла конфигурации, doxygen может создавать шаблон файла конфигурации. Для создания файла конфигурации необходимо вызвать doxygen из командной строки с параметром `-g`:

```
doxygen -g <config-file>
```

где `<config-file>` – это имя файла конфигурации.

Если наименование файла не указывать, будет создан файл с именем `Doxyfile`.

Если файл с именем `<config-file>` уже существует, doxygen переименует его в `<config-file>.bak` перед созданием шаблона конфигурации.

Если передать знак «-» (знак минус) в качестве имени файла, то doxygen попытается прочитать файл конфигурации из стандартного ввода `stdin`, что может быть полезно для написания сценариев.

Ниже представлен файл конфигурации doxygen с основными параметрами:

```
PROJECT_NAME      = "NAME"
PROJECT_NUMBER    = 1.0
PROJECT_BRIEF     = "Краткое описание проекта"
OUTPUT_DIRECTORY  = build
INPUT             = main.cpp
OUTPUT_LANGUAGE   = Russian
GENERATE_LATEX    = No
EXTRACT_PRIVATE   = Yes
HTML_OUTPUT       = HTML
```

PROJECT_NAME – одно слово или последовательность слов, заключенная в двойные кавычки, которое должно идентифицировать проект, для которого создается документация. Это имя используется в заголовке большинства сгенерированных страниц. Значение по умолчанию: My project.

PROJECT_NUMBER – номер версии проекта.

PROJECT_BRIEF – однострочное описание проекта, отображающееся вверху каждой страницы, краткое описание цели проекта.

OUTPUT_DIRECTORY используется для указания относительного или абсолютного пути, в который будет записана сгенерированная документация.

INPUT используется для указания файлов и/или каталогов, содержащих документированные исходные файлы. Если параметр не задан, будет выполняться поиск по текущему каталогу.

OUTPUT_LANGUAGE используется для указания языка, на котором написана вся документация, сгенерированная doxygen. По умолчанию документация генерируется на английском языке.

GENERATE_LATEX используется для генерации выходных данных в формате *latex*. По умолчанию генерация включена. Результат может быть преобразован в формат PDF.

GENERATE_HTML формирует документацию в формате HTML. По умолчанию включен. Такая документация удобна для размещения онлайн. Данный параметр можно выключить, если такая документация не нужна.

HTML_OUTPUT задает каталог, в который помещается документация в формате *html*. Значение по умолчанию *html*.

LATEX_OUTPUT задает каталог, в который помещается документация в формате *TeX*. Значение по умолчанию *latex*.

`EXTRACT_PRIVATE` документирует закрытые атрибуты и методы класса. По умолчанию они не документируются. Для включения их в документацию необходимо установить значение *YES*.

Для генерации документации в формате *PDF* необходимы дополнительные пакеты. Установка пакетов в *OS linux*:

```
sudo apt-get install texlive-latex-base texlive-latex-recommended texlive-latex-extra
```

Установка поддержки кириллицы:

```
sudo apt-get install texlive-lang-cyrillic
```

Для генерации документации необходимо передать `doxygen` файл конфигурации в качестве параметра. Исполнение в командной строке:

```
doxygen <config-file>
```

, где `<config-file>` – это имя файла конфигурации.

Документирование программного кода

Документирование кода осуществляется с помощью специального документирующего блока. Специальный блок комментариев – это блок комментариев в стиле C или C++ с некоторыми дополнительными пометками, обрабатывая которые `doxygen` знает, что это фрагмент структурированного текста, который должен попасть в сгенерированную документацию.

Если в файле конфигурации для параметра *EXTRACT_ALL* установлено значение *NO* (по умолчанию), тогда `doxygen` будет генерировать документацию только для документированных объектов.

Для членов, классов и пространств имен существует два основных варианта размещения блока комментариев:

- размещать специальный блок документации перед объявлением или определением члена, класса или пространства имен. Для членов файла, класса и пространства имен также разрешено размещать документацию непосредственно после члена;

- размещать специальный блок документации в другое место (другой файл или другое место) и помещать в блок документации структурную команду. Структурная команда связывает блок документации с определенным объектом, который может быть задокументирован (например, членом, классом, пространством имен или файлом).

Второй подход не рекомендован к использованию для документирования исходного кода, так как документирующий блок оторван от документируемого элемента.

Документировать файлы можно только с помощью второго варианта, так как перед файлом невозможно поставить блок документации. Конечно, члены файла (функции, переменные, определения типов, определения) не нуждаются в явной структурной команде; просто поместив специальный блок документации перед или позади них, все будет работать нормально.

Блок документирования в стиле Javadoc, который состоит из блока комментариев в стиле C, начинающегося с двух *:

```
/**
 * ... text ...
 */
```

Блок документирования в стиле Qt:

```
/*!
 * ... text ...
 */
```

В обоих случаях промежуточные * являются необязательными:

```
/*!
 ... text ...
 */
```

Также можно использовать блок, по крайней мере, из двух строк комментариев на C++, где каждая строка начинается с дополнительной косой черты или восклицательного знака. Примеры этих двух случаев:

```
///
/// ... text ...
///
```

или

```
/*!
/*! ... text ...
//!
```

**в этом случае блок документации заканчивается пустой строкой.*

Размещение блоков документации в одной строке

Если вы хотите задокументировать элементы файла, структуры, объединения, класса или перечисления, иногда желательно поместить блок документации после элемента, а не перед ним. Для этого вы должны поместить дополнительный маркер < в блок комментариев. Обратите внимание, что это также работает для параметров функции.

Примеры:

```
int var; /*!< Подробное описание после переменной */
```

QT пример:

```
int var; /**< Подробное описание */
```

или

```
int var; /*!< Подробное описание  
        /*!<
```

или

```
int var; ///< Подробное описание  
        ///<
```

Чаще всего требуется только краткое описание, для его определения в основном используют следующий синтаксис:

```
int var; /*!< Краткое описание
```

или

```
int var; ///< Краткое описание
```

Эти блоки могут использоваться только для документирования элементов и параметров. Они не могут применяться для документирования файлов, классов, объединений, структур, групп, пространств имен, макросов и самих перечислений. Кроме того, структурные команды, упомянутые в следующем разделе (например `\class`), не разрешены внутри этих блоков комментариев.

Пример блоков документирования класса:

```
/*! Класс Afterdoc тест */  
class Afterdoc_Test  
{  
public:  
    /** Тип Enum.  
    *  
    */  
    enum EnumType  
    {  
        int EVal1, /**< enum значение 1 */
```

```
int EVal2 /**< enum значение 2 */
};
void member(); /**< метод класса.
protected:
int value; /**< целочисленное значение */
```

Размещение блоков документации вне кода класса

В примерах из предыдущего раздела блоки комментариев всегда располагались перед объявлением или определением файла, класса или пространства имен, а также перед или после одного из его элементов. Несмотря на то, что это удобно, иногда могут быть причины поместить документацию в другое место. Например, для документирования файла это необходимо, поскольку нет такого понятия, как «перед файлом».

Доxygen позволяет размещать блоки документации практически в любом месте (исключение – внутри тела функции или обычного блока комментариев в стиле C).

Цена, которую вы платите за то, что не помещаете блок документации непосредственно перед или после элемента, заключается в необходимости поместить структурную команду внутри блока документации, что приводит к некоторому дублированию информации. Таким образом, на практике вам следует избегать использования структурных команд, если только другие требования не вынуждают вас делать это.

Структурные команды, как и все другие команды, начинаются с обратной косой черты (\) или знака (@), если используется стиль Javadoc, за которыми следуют имя команды и один или несколько параметров. Пример:

```
/*! \class Test
    \brief Короткое описание.

    Описание класса.
*/
```

Здесь специальная команда \class используется для указания на то, что блок комментариев содержит документацию для класса Test. Другими структурными командами являются:

```
\struct для документирования C-struct;
\union для документирования union;
\enum для документирования enumeration type;
```

`\fn` для документирования `function`;
`\var` для документирования `variable or typedef` или `enum value`;
`\def` для документирования `#define`;
`\typedef` для документирования `type definition`;
`\file` для документирования `file`;
`\namespace` для документирования `namespace`;
`\package` для документирования `Java package`;
`\interface` для документирования `IDL interface`.

Полный перечень команд приведен в документации `doxygen`.

Для построения документации часто также используются следующие команды:

`@brief` краткое описание (класса, метода и т.п.);
`@details` полное описание (класса, метода и т.п.);
`@param foo` описание параметра `foo` функции или метода;
`@return` описание возвращаемого функцией (методом) значения;
`@throw Foo` метод возбуждает исключение `Foo`;
`@warning` предупреждение;
`@authors` указывает автора или авторов;
`@version` используется для указания версии;
`@date` предназначена для указания даты разработки;
`@bug` перечисление известных ошибок;
`@copyright` используемая лицензия.

Пример использования команд в начале файла проекта:

```
/** @file
 * @author Фамилия И.О.
 * @version 1.0
 * @date 01.01.2020
 * @copyright ИБСТ ПГУ
 * @warning Это учебный пример
 * @brief Заголовочный файл Модуля EDU
 */
```

Пример документирующего блока класса с использованием команд:

```
/** @brief Шифрование методом Гронсфельда
 * @details Ключ устанавливается в конструкторе.
 * Метод encrypt осуществляет зашифрование
 * Метод decrypt осуществляет расшифрование
```

```
* @warning Реализация только для английского языка
*/ class modAlphaCipher
{
    // описание класса }
```

Пример документирования метода класса с использованием команд:

```
/** * @brief Зашифровывание
* @param [in] open_text Открытый текст. Не должен быть пустой строкой.
* Строчные символы автоматически преобразуются к прописным.
* Все не-буквы удаляются
* @return Зашифрованная строка
* @throw cipher_error, если текст пустой
*/ std::string encrypt(const std::string& open_text);
```

Программа для разбора параметров командной строки с использованием библиотеки boost

Модуль interface Заголовок interace.h

```
#pragma once
#include <boost/program_options.hpp>
#include <string>
namespace po = boost::program_options;

// структура для хранения полученных параметров
struct Params {
    std::string inFileName;
    std::string userName;
};
// Интерфейс
class Interface {
private:
    // пустой парсер
    po::options_description desc;
    // переменная для результатов парсинга
    po::variables_map vm;
    // результаты парсинга параметров
    Params params;
public:
    Interface();
    bool Parser(int argc, const char** argv);
    std::string getDescription();
    Params getParams() {
        return params;
    };
};
```

Реализация interface.cpp

```
#include "interface.h"
#include <sstream>
Interface::Interface() : desc("Allowed options")
```

```

{
    // добавление параметров в парсер командной строки
    desc.add_options()
        ("help,h", "Show help")
        ("input,i",
         po::value<std::string>(&params.inFileName)-
         >default_value("input.txt"),
         "Set input file name")
        ("user,u",
         po::value<std::string>(&params.userName)->required(),
         "Set user name");
}

bool Interface::Parser(int argc, const char** argv)
{
    // парсинг
    po::store(po::parse_command_line(argc, argv, desc), vm);
    // обрабатываем --help до вызова notify
    if (vm.count("help"))
        return false;
    // присвоение значений по умолчанию и возбуждение исключений
    po::notify(vm);
    return true;
}

std::string Interface::getDescription()
{
    std::ostringstream ss;
    ss << desc;
    return ss.str();
}

```

Модуль main

Реализация main.cpp

```

#include <iostream>
#include "interface.h"

int main(int argc, const char** argv)
{
    try {
        Interface iface;
        if (iface.Parser(argc, argv)) {
            // параметры получены

```

```

        Params params = iface.getParams();
        std::cout << "File: " << params.inFileName << "\n";
        std::cout << "User: " << params.userName << "\n";
    } else {
        // получен --help

        std::cout << iface.getDescription();
    }
} catch(po::error& e) {
    // ошибки парсинга
    std::cerr << "error: " << e.what() << "\n";
    std::cerr << "Use -h for help\n";
    return 1;
} catch(std::exception& e) {
    // прочие ошибки библиотеки C++
    std::cerr << "error: " << e.what() << "\n";
    return 2;
} catch(...) {
    // ошибки из неизвестных источников
    std::cerr << "Exception of unknown type!\n";
    std::terminate();
}
}

```

Примеры программ для формирования «соли», формирования хеша, проверки хеша

```
// ===== 1 =====  
/*
```

Пример формирования «соли» заданного рамера
с помощью генератора ПСП библиотеки Crypto++
и преобразование ее в шестнадцатеричную строку

```
*/  
#include <iostream>  
#include <string>  
#include <cryptopp/hex.h>  
#include <cryptopp/osrng.h>  
namespace CPP = CryptoPP;  
constexpr int salt_size = 12;           // размер «соли» в байтах  
int main(int argc, char *argv[]) {  
    CPP::byte salt[salt_size];          // массив для «соли»  
    CPP::AutoSeededRandomPool prng;  
    prng.GenerateBlock(salt, salt_size); // генерация «соли»  
    std::string salt_16;                 // строка для «соли»  
    // преобразование «соли» в шестнадцатеричную строку  
    CPP::ArraySource(salt, salt_size, true,  
                     new CPP::HexEncoder(  
                         new CPP::StringSink(salt_16)));  
    std::cout << salt_16 << std::endl;  // вывод «соли»  
    return 0;  
}
```

```
// ===== 2 =====  
/*
```

Пример формирования хеша от «соли» и пароля
с помощью библиотеки Crypto++
и преобразование его в шестнадцатеричную строку

```
*/  
#include <iostream>  
#include <string>  
#include <cryptopp/hex.h>  
#include <cryptopp/sha.h>
```

```

namespace CPP = CryptoPP;
constexpr std::string_view salt {"4F9C429F5C6884DB726445BE"};
constexpr std::string_view pass {"SupErP@$w0rd"};
int main(int argc, char *argv[]) {
    CPP::SHA256 sha256;           // хеш-функция
    std::string hash_16;          // строка для хеша
    // формирование хеша и преобразование в шестнадцатеричную
    // строку
    CPP::StringSource(
        std::string(salt).append(pass),
        true,
        new CPP::HashFilter(
            sha256,
            new CPP::HexEncoder(
                new CPP::StringSink(hash_16))));
    std::cout << hash_16 << std::endl; // вывод хеша
    return 0;
}

// ===== 3 =====
/*
    Пример проверки хеша с помощью библиотеки Crypto++
    и сохранение результата в булевскую переменную
*/
#include <iostream>
#include <string>
#include <cryptopp/hex.h>
#include <cryptopp/sha.h>
namespace CPP = CryptoPP;
// хеш в виде шестнадцатеричной строки
constexpr std::string_view hash_16 =

"9C67ABCFAD8E46E78F89173497800E5EE143D67C28765A60C9472
21E7114FF68";
// «соль» + пароль
constexpr std::string_view salt_pass =
    "4F9C429F5C6884DB726445BESupErP@$w0rd";

int main(int argc, char *argv[]) {
    CPP::SHA256 sha256;           // хеш-функция
    bool isVerified;              // для результата
    std::string hash;             // для двоичного хеша
    // преобразование хеша в двоичный вид из шестнадцатеричного

```

```

        CPP::StringSource (std::string(hash_16), true,
                           new CPP::HexDecoder(
                               new CPP::StringSink(hash)));
// проверка хеша
    CPP::StringSource (
        hash.append(salt_pass), true,
        new CPP::HashVerificationFilter(
            sha256,
            new CPP::ArraySink((CPP::byte*) &isVerified, sizeof(bool))));
// вывод результата
    std::cout << std::boolalpha << "Verified: " << isVerified << std::endl;
    return 0;
}

```

Примеры программ для формирования и проверки подписи аутентификационного сообщения

```
// ===== 1 =====  
/*  
    Пример формирования подписи сообщения аутентификации  
    с помощью библиотеки Crypto++  
    и преобразование его в шестнадцатеричную строку  
*/  
  
#include <iostream>  
#include <string>  
#include <cryptopp/hex.h>  
#include <cryptopp/hmac.h>  
#include <cryptopp/sha.h>  
namespace CPP = CryptoPP;  
  
// соль  
constexpr std::string_view salt {"4F9C429F5C6884DB726445BE"};  
  
// логин  
constexpr std::string_view login {"ibst128"};  
  
// пароль  
constexpr std::string_view pass {"SupErP@$w0rd"};  
  
int main(int argc, char *argv[]) {  
  
    // объект для формирования подписи с установленным паролем  
    CPP::HMAC<CPP::SHA256> hmac_sha256((const  
    CPP::byte*)pass.data(),  
                                         pass.size());  
  
    // строка для сохранения подписи  
    std::string hmac_16;
```

```

// формирование подписи и преобразование в шестнадцатеричную
строку
    CPP::StringSource(
        std::string(login).append(salt),
        true,
        new CPP::HashFilter(
            hmac_sha256,
            new CPP::HexEncoder(
                new CPP::StringSink(hmac_16))));

// вывод подписи
    std::cout << hmac_16 << std::endl;
    return 0;
}

// ===== 2 =====
/*
    Пример проверки подписи с помощью библиотеки Crypto++
    и сохранение результата в булевскую переменную
*/
#include <iostream>
#include <string>
#include <cryptopp/hex.h>
#include <cryptopp/sha.h>
#include <cryptopp/hmac.h>
namespace CPP = CryptoPP;

// подпись в виде шестнадцатеричной строки
constexpr std::string_view hmac_16 =

"265FC6E8247256158D9FBD6CE1A805BF4CDB7CF2230268591F69C
D9A03E4A3D8";

// сообщение логин + «соль»
constexpr std::string_view msg =
    "ibst1284F9C429F5C6884DB726445BE";

// пароль
constexpr std::string_view pass = "SupErP@$w0rd";

int main(int argc, char *argv[]) {

// объект для формирования подписи с установленным паролем

```

```

    CPP::HMAC<CPP::SHA256> hmac_sha256((const
CPP::byte*)pass.data(),
                                     pass.size());

// переменная для сохранения результата проверки
    bool isVerified;

// переменная для сохранения двоичной подписи
    std::string hmac;

// преобразование подписи в двоичный вид из шестнадцатеричного
    CPP::StringSource (std::string(hmac_16), true,
                       new CPP::HexDecoder(
                           new CPP::StringSink(hmac)));
// проверка подписи
    CPP::StringSource (
        hmac.append(msg), true,
        new CPP::HashVerificationFilter(
            hmac_sha256,
            new CPP::ArraySink((CPP::byte*) &isVerified, sizeof(bool))));
// вывод результата проверки
    std::cout << std::boolalpha << "Verified: " << isVerified << std::endl;
    return 0;
}

```

Программа тестирования интерфейсного модуля

```
#include <UnitTest++/UnitTest++.h>
#include "interface.h"
// сценарий тестирования справки
SUITE(HelpTest)
{
    TEST(ShortHelp) {
        Interface iface;
        const char* argv[] = {"test", "-h", nullptr};
        int argc = sizeof argv / sizeof nullptr - 1;
        REQUIRE CHECK(!iface.Parser(argc, argv));
        CHECK(!iface.getDescription().empty());
    }
    TEST(LongHelp) {
        Interface iface;
        const char* argv[] = {"test", "--help", nullptr};
        int argc = sizeof argv / sizeof nullptr - 1;
        REQUIRE CHECK(!iface.Parser(argc, argv));
        CHECK(!iface.getDescription().empty());
    }
    TEST(HelpAndUser) {
        Interface iface;
        const char* argv[] = {"test", "-h", "-u", "vasya", nullptr};
        int argc = sizeof argv / sizeof nullptr - 1;
        REQUIRE CHECK(!iface.Parser(argc, argv));
        CHECK(!iface.getDescription().empty());
    }
    TEST(UserAndHelp) {
        Interface iface;
        const char* argv[] = {"test", "-u", "vasya", "-h", nullptr};
        int argc = sizeof argv / sizeof nullptr - 1;
        REQUIRE CHECK(!iface.Parser(argc, argv));
        CHECK(!iface.getDescription().empty());
    }
}
// сценарий тестирования рабочих параметров
```

```

SUITE(WorkParamsTest) {
    TEST(ShortUserOnly) {
        Interface iface;
        std::string user("vasya");
        std::string file("input.txt");
        const char* argv[] = {"test", "-u", user.c_str(), nullptr};
        int argc = sizeof argv / sizeof nullptr - 1;
        REQUIRE CHECK(iface.Parser(argc, argv));
        CHECK_EQUAL(iface.getParams().userName, user);
        CHECK_EQUAL(iface.getParams().inFileName, file);
    }
    TEST(LongUserOnly) {
        Interface iface;
        std::string user("vasya");
        std::string file("input.txt");
        const char* argv[] = {"test", "--user", user.c_str(), nullptr};
        int argc = sizeof argv / sizeof nullptr - 1;
        REQUIRE CHECK(iface.Parser(argc, argv));
        CHECK_EQUAL(iface.getParams().userName, user);
        CHECK_EQUAL(iface.getParams().inFileName, file);
    }
    TEST(UserAndFile) {
        Interface iface;
        std::string user("vasya");
        std::string file("a.txt");
        const char* argv[] = {"test",
                               "-u", user.c_str(),
                               "-i", file.c_str(),
                               nullptr};
        int argc = sizeof argv / sizeof nullptr - 1;
        REQUIRE CHECK(iface.Parser(argc, argv));
        CHECK_EQUAL(iface.getParams().userName, user);
        CHECK_EQUAL(iface.getParams().inFileName, file);
    }
    TEST(FileAndUser) {
        Interface iface;
        std::string user("x");
        std::string file("a.txt");
        const char* argv[] = {"test",
                               "--input", file.c_str(),
                               "-u", user.c_str(),
                               nullptr};
        int argc = sizeof argv / sizeof nullptr - 1;
    }
}

```

```

        REQUIRE CHECK(iface.Parser(argc, argv));
        CHECK_EQUAL(iface.getParams().userName, user);
        CHECK_EQUAL(iface.getParams().inFileName, file);
    }
}
// сценарий тестирования исключений
SUITE(ExceptionTest) {
    TEST(NoParams) {
        Interface iface;
        const char* argv[] = {"test", nullptr};
        int argc = sizeof argv / sizeof nullptr - 1;
        CHECK_THROW(iface.Parser(argc, argv), po::required_option);
    }
    TEST(NoUserValue) {
        Interface iface;
        const char* argv[] = {"test", "-u", nullptr};
        int argc = sizeof argv / sizeof nullptr - 1;
        CHECK_THROW(iface.Parser(argc, argv),
po::invalid_command_line_syntax);
    }
    TEST(NoUserParam) {
        Interface iface;
        const char* argv[] = {"test", "-i", "a.txt", nullptr};
        int argc = sizeof argv / sizeof nullptr - 1;
        CHECK_THROW(iface.Parser(argc, argv), po::required_option);
    }
    TEST(UnknownParam) {
        Interface iface;
        const char* argv[] = {"test", "-i", "a.txt", "-v", nullptr};
        int argc = sizeof argv / sizeof nullptr - 1;
        CHECK_THROW(iface.Parser(argc, argv), po::unknown_option);
    }
}
int main()
{
    return UnitTest::RunAllTests();
}

```

Учебное издание

**Лупанов Михаил Юрьевич,
Сидоров Никита Алексеевич**

**Разработка клиент-серверных приложений
с подсистемой аутентификации**

Редактор *Н. А. Сидельникова*
Технический редактор *С. В. Денисова*
Компьютерная верстка *С. В. Денисовой*

Подписано в печать 08.11.2024.
Формат 60×84¹/₁₆. Усл. печ. л. 4,42.
Тираж 10. Заказ № 611.

Издательство ПГУ
440026, г. Пенза, ул. Красная, 40
Тел.: (8412) 66-60-49, 66-67-77; e-mail: iic@pnzgu.ru

Вниманию авторов!

Издательство ПГУ выпускает учебную, научную и художественную литературу, презентационную и акцидентную продукцию, а также полноцветные юбилейные и мемориальные издания в соответствии с ГОСТ 7.60–2003.

Издательство ПГУ принимает к изданию рукописи, подготовленные с использованием текстового редактора Microsoft Word for Windows версий **2003 и выше**. Формат – А4, основной шрифт – Times New Roman, 14–16 pt через одинарный интервал (минимальный размер шрифта в таблицах и сносках – 12,5 pt). Тип файла в электронном виде – doc, docx.

Работа должна содержать индекс УДК, аннотацию.

Аннотация (ГОСТ 7.86–2003, ГОСТ 7.9–1995) включает характеристику основной темы, проблемы объекта, цели работы и ее результаты. В аннотации указывают, что нового несет в себе данный документ в сравнении с другими, родственными по тематике и целевому назначению. Аннотация может включать сведения о достоинствах произведения. Текст аннотации начинают фразой, в которой сформулирована главная тема документа. Заканчивается аннотация читательским адресом.

Рисунки и таблицы должны быть размещены в тексте после ссылки на них (растровые рисунки предоставляются в виде отдельных файлов в формате jpg, BMP с разрешением 300 dpi, векторные рисунки в формате Corel Draw с минимальной толщиной линии 0,75 pt. Рисунки должны быть доступны для правки!). Рисунки должны сопровождаться подрисуночными подписями, на все рисунки и таблицы в тексте должны быть ссылки.

Формулы в тексте выполняются только в редакторе формул **MathType версия 5.0** и выше. Символы греческого и русского алфавита должны быть набраны прямо, нежирно; латинского – курсивом, нежирно; обозначения векторов и матриц – прямо, жирно; цифры – прямо, нежирно. Наименования химических элементов набираются прямо, нежирно. Эти же требования необходимо соблюдать и в рисунках.

В списке литературы **нумерация источников** должна соответствовать очередности ссылок на них в тексте ([1], [2], ...). Номер источника указывается в квадратных скобках. Требования к оформлению списка литературы на русские и иностранные источники (ГОСТ Р 7.0.5–2008): для книг – фамилия и инициалы автора, название, город, издательство, год издания, том, количество страниц; для журнальных статей, сборников трудов – фамилия и инициалы автора, название статьи, полное название журнала или сборника, серия, год, том, номер, страницы; для материалов конференций – фамилия и инициалы автора, название статьи, название конференции, город, издательство, год, страницы.

К материалам **должна** прилагаться следующая информация: фамилия, имя, отчество, контактные телефоны.

Контакты Издательства ПГУ: (8412) 66-60-49, 66-67-77. E-mail: iic@pnzgu.ru



