МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный электротехнический
университет «ЛЭТИ» им. В. И. Ульянова (Ленина)

БАЗОВЫЕ СВЕДЕНИЯ К ВЫПОЛНЕНИЮ ЛАБОРАТОРНЫХ РАБОТ ПО ДИСЦИПЛИНЕ «ПРОГРАММИРОВАНИЕ». ВТОРОЙ СЕМЕСТР

Учебно-методическое пособие

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2024

УДК 004.42(07)
ББК З 973.2-018я7
Б17

Авторы: **М. М. Заславский, А. А. Лисс, А. В. Гаврилов, С. А. Глазунов, Я. С. Государкин, С. А. Тиняков, В. П. Голубева, К. В. Чайка, В. Е. Допира.**

Б17 Базовые сведения к выполнению лабораторных работ по дисциплине «Программирование». Второй семестр: учеб.-метод. пособие. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2024. 84 с.

ISBN 978-5-7629-3306-3

Представлены материалы для выполнения лабораторных работ по дисциплине «Программирование». Предназначено для использования в весеннем семестре 1-го курса бакалавриата.

Содержит необходимые теоретические сведения и поясняющие примеры. Приведенные листинги программ позволяют не только глубже изучить темы дисциплины, но и дают отправную точку для выполнения лабораторных работ курса. Описание каждой лабораторной работы завершается вопросами для самоконтроля.

Предназначено для студентов направлений «Программная инженерия» и «Прикладная математика и информатика».

УДК 004.42(07)
ББК З 973.2-018я7

Рецензент кандидат технических наук Ю. Б. Остапченко (зам. ген. директора по специальной тематике ООО «Альвекс»).

Утверждено
редакционно-издательским советом университета
в качестве учебно-методического пособия

ISBN 978-5-7629-3306-3

ВВЕДЕНИЕ

В данном пособии приводятся методические указания для выполнения лабораторных работ по дисциплине «Программирование» на языке Си и C++ для 1-го курса весеннего семестра обучения. Перечень лабораторных работ соответствует рабочей программе дисциплины и включает следующее:

1. Регулярные выражения, их структура и применение для работы с текстовыми данными.
2. Записи (структуры) и их реализация в языке программирования.
3. Разновидности линейных списков.
4. Указатели, структуры, рекурсивные типы данных.
5. Абстрактные типы данных (АТД).
6. Организация ввода/вывода и работа с файлами.
7. Жизненный цикл разработки программ.
8. Идея абстрактных структур данных. Реализации АТД при помощи классов C++.

Материал для удобства изучения разделен на две большие части: общие базовые сведения и подробный разбор каждой лабораторной работы. Более полное описание возможностей стандартной библиотеки языка (libc), а также дополнительные материалы, помогающие в понимании предметной области, представлены в приложениях.

Для обеспечения образовательного процесса по дисциплине используются следующие информационные технологии:

1. Операционная система Ubuntu Desktop 20.04 x64.
2. Программное обеспечение общего и специализированного назначения: Git version 2.17, LibreOffice 6 и LibreOffice 6 Help Pack (Russian), GCC 7.5.0. Стандарт языка Си: C99.
3. Информационные справочные системы: международная ассоциация сетей «Интернет», электронные библиотечные системы и ресурсы удаленного доступа библиотеки СПбГЭТУ «ЛЭТИ».

Процедура защиты лабораторной работы. Процедура защиты лабораторной работы состоит из двух основных этапов: реализация программы в соответствии с заданием лабораторной работы с подготовкой пояснительной записки и защита лабораторной работы.

Реализация программы включает в себя написание файла с исходным кодом программы на языке Си/C++, которая удовлетворяет целям и задачам

лабораторной работы. Исходный код программы должен быть компилируемым без ошибок и предупреждений компилятора GCC 7.5.0. Кроме того, программа считается корректной только при прохождении тестирования. По реализованной программе должна быть написана пояснительная записка, которая включает постановку задачи с указанием варианта задания, описание программы и алгоритма решения поставленной задачи, тестирование программы на произвольных входных данных и выводы.

Для подтверждения навыков, которые студент приобрел в процессе выполнения работы, студент проходит процедуру защиты лабораторной работы. Защита включает в себя решение трех практических или теоретических задания, которые показывают уровень владения материалом лабораторной работы (примеры заданий на защиту приложены к лабораторным работам).

Лабораторная работа № 1
РЕГУЛЯРНЫЕ ВЫРАЖЕНИЯ

1.1 Цель и задачи

Функции форматного ввода-вывода позволяют динамически читать или выводить различные стандартные типы данных в соответствии с форматной строкой. В форматной строке можно указать последовательность символов, строк, чисел, которые будут введены или выведены. Однако контроль вводимых пользователем символов является достаточно сложной задачей, для решения которой недостаточно только форматных строк. Пользователи могут и будут ошибаться при вводе данных в программы – например, совершать грамматические ошибки, использовать лишние символы или пропускать необходимые знаки. Как быть, если пользователю необходимо ввести в программу корректный IP-адрес [1]? Такую задачу можно решить проверками строки после считывания, но при увеличении количества требований к считанной строке увеличивается и количество условий, которые нужно проверить. Такие проверки с помощью условных операторов требуют большого количества кода. Именно поэтому в данной работе будут рассмотрены расширенные возможности ввода в языке Си и регулярные выражения, которые позволят обрабатывать строки любого вида.

Целью работы является освоение работы с регулярными выражениями.

Для достижения поставленной цели требуется решить следующие задачи:

– изучить расширенные возможности форматного ввода/вывода в языке Си;

– ознакомиться с синтаксисом регулярных выражений;

– изучить способы применения POSIX регулярных выражения в языке Си;

– написать программу реализующую обработку и поиск подстрок по шаблону в тексте с помощью регулярных выражений.

1.2. Основные теоретические сведения

1.2.1. Расширенные возможности форматного ввода/вывода

В языке Си предусмотрен ряд функций для форматного ввода и вывода *printf* и *scanf* [2]; [3]:

```
int printf ( const char * format, arg1, arg2, ...argN);
int scanf ( const char * format, &arg1, &arg2, …&argN);
```

Данные функции получают на вход строку format с заданным форматом считывания или вывода. Затем идут аргументы, откуда *printf* берет значения для вывода или куда *scanf* записывает считанные данные в соответствии с форматом. Формат считывания или вывода переменных определен в табл. 1.1.

Таблица 1.1

Код	Формат
%c	Символ типа *char*
%d	Десятичное число целого типа со знаком
%i	Десятичное число целого типа со знаком для *printf*. Для *scanf*: по умолчанию считывается целое, если число начинается с 0, то считывает восьмеричное, а если с *0x*, то шестнадцатеричное
%e, %E	Экспоненциальный формат (научная нотация) с выводом символа ‘*e*’ в верхнем и нижнем регистре соответственно
%f	Десятичное число с плавающей точкой
%g	Использует наиболее короткий вывод среди кодов *%e* и *%f*
%G	Использует наиболее короткий вывод среди кодов *%E* и *%f*
%o	Восьмеричное беззнаковое целое число
%s	Строка символов
%u	Десятичное беззнаковое число целого типа
%x	Шестнадцатеричное беззнаковое целое число с буквами нижнего регистра
%X	Шестнадцатеричное беззнаковое целое число с буквами верхнего регистра
%p	Выводит на экран значение указателя (адрес в шестнадцатеричной системе счисления)
%%	Выводит символ *%*

Помимо спецификаторов для указания типа выводимого значения, точности, ширины, которые уже рассматривались в 1-м семестре курса, функции форматного ввода позволяют ограничивать перечень символов, который могут содержаться в строке. Эта возможность есть только для спецификатора *%s* в функциях форматного ввода, чтобы указать, какие символы в строке могут содержаться. Набор символов, который должен содержаться в искомой строке, указывается с помощью *%[]s*. Например, для сканирования строки, которая содержит только буквы *abc* и символ нижнего регистра, нужно указать спецификатор *%[a-c_]s*.

Рассмотрим следующий код:

```
#include <stdio.h>

int main()
{
   char str[1000] = "Old string";
   scanf("%[a-c_]s", str);
   puts(str);
   return 0;
}
```

Данная программа корректно считывает все первые символы *‘a’*, *‘b’*, *‘c’*, *‘_’* из входной строки пока не встретит символ, не входящий в этот список. То есть, если в начале входной строки идут заданные символы, то они будут считаны до первого символа не из списка, иначе ни один символ считан не будет. Если в начале строки окажется символ, не входящий в данный набор, то строка не будет перезаписана и выведется строка: *“Old string”*.

То есть, если на вход программе подать строку *“bc_ac_bkl”*, то будет выведено: *“bc_ac_b”*.

Если на вход программе подать строку *“Abc_ac_b”*, то будет выведено: *“Old string”*.

В квадратных скобках могут указываться как последовательность подряд идущих символов *%[abc]*, так и диапазон символов в соответствии с таблицей ASCII *%[0-z]* (соответствует всем символам в указанном диапазоне согласно таблице ASCII).

Для того чтобы считывать по принципу “всё кроме”, можно указать набор символов, которые не могут содержаться в считанной строке. Например, с помощью *%[^0-9]s* можно считать строку, которая не содержит цифр.

Использование наборов позволяет ограничить вводимые данные и считывать ровно то, что необходимо для обработки, а строки, которые не соответствуют указанным наборам символов, не обрабатывать.

1.2.2. Регулярные выражения

Базовые сведения. *Регулярные выражения* – это формальный язык для работы с текстом. Основными задачами, которые решают регулярные выражения, являются: поиск подстроки в строке, проверка строки на соответствие шаблону. Регулярные выражения позволяют описать множество строк одним выражением. По аналогии с форматным вводом регулярные выражения состоят из спецификаторов, которые определяют содержание, порядок и количество символов в строке. Регулярные выражения позволяют работать, как с отдельными символами, так и с числами или группами символов.

Определим что такое шаблон строки. Рассмотрим задачу, в которой необходимо искать любую из строк:

- *“abc”*;
- *“ab”*;
- *“ccab”*.

Для решения этой задачи необходимо описать все 3 строки некоторым набором правил. Выделим общие признаки каждой из строк: есть 2 общие буквы *“ab”*, минимальная длина – 2 буквы. Для поиска подобных строк можно написать программу, которая будет проверять соответствие строки написанным правилам. Такая программа будет состоять из множества проверок и строк. Но в общем виде все 3 строки можно описать так:

<0 или 2 символа “c” в начале>ab<1 и 0 символов “c” в конце>

В регулярных выражениях ситуацию *<0 и более символов>* можно описать в шаблоне спецификатором **“*”** после указания самого символа, а *<1 или 0 символов>* спецификатором **“?”** после символа. Таким образом, регулярное выражение будет выглядеть как: *“c*abc?”*, но такое выражение описывает и другие строки, например, *“ccccab”*, которой нет в нашем списке строк.

После конкретного символа, набора или группы символов можно в фигурных скобках указать количество повторений данного символа или группы символов в строке, а также можно указать условный выбор символов и наборов символов с помощью знака ИЛИ. Тогда выражение будет выглядеть следующим образом:

“(c{0}|c{2})abc?”.

Кроме того, это же выражение можно написать с помощью знаков ИЛИ: *“abc|ab|ccab”*. Подобные строки и являются шаблонами, которые описывают множество других строк. С помощью регулярных выражений можно описать любой набор строк.

Описанные регулярные выражения будут соответствовать только строкам:

• “abc”;

• “ab”;

• “ccab”.

Но регулярные выражения позволяют не только проверить строку на соответствие шаблону, но и находить подстроки. Таким образом по данным регулярным выражениям можно найти подстроку *“ab”* в строке *“He is a st**ab**le excellent student”*. Чтобы проверять соответствие строк нужно указывать символ начала строки – *“^”* и символ конца строки – *“$”* в регулярном выражении.

Подробнее об применении регулярных выражений и о том, как они работают, можно узнать в книге «Регулярные выражения» Д. Фридла [4].

Введение в регулярные выражения. Существуют различные разновидности языков описания шаблонов – регулярных выражений. В программировании разделяют 2 основных формата регулярных выражений:

• **POSIX** регулярные выражения;

• **UNICODE** регулярные выражения.

В данной работе будут рассматриваться только **POSIX** регулярные выражения. Например, рассмотрим задачу распознавания конструкции следующего вида: *<MyTag>Something Incredible<\MyTag>*. В данных конструкциях используются метки в угловых скобках **<>**, которые будем именовать тегами. Существуют теги открывающие *<MyTag>* и закрывающие *<\MyTag>* (закрывающие всегда начинаются с символа *‘\’*), а также некоторый текст между этими тэгами. Будем считать, что внутри тэгов могут быть только слова без пробелов и цифр, а между тэгов только символы и пробелы.

В таком случае **POSIX** регулярное выражение будет выглядеть следующим образом:

<(\w+)>[\w\s]+<\\\1>

Данным выражением были указаны последовательности допустимых символов в строке, которые необходимо считать. Разберем регулярное выражение. В **POSIX** выражении используются следующие спецификаторы:

1) *\w* – спецификатор, указывающий один или более символов латинского алфавита;

2) *\s* – спецификатор, указывающий символ табуляции;

3) + – спецификатор, показывающий, что количество символов, указанных перед данным спецификатором, должно равняться одному или более символам;

4) *()* – спецификатор, указывающий группу (подробнее будет рассмотрено далее);

5) *\1* – спецификатор, указывающий на то, что в данном месте ожидается строка, которая полностью равна строке в первой группе, указанной с помощью ();

6) *[]* – спецификатор, указывающий на набор символов или спецификаторов, которые ожидаются на данном месте регулярного выражения.

Символы, которые не вошли в этот список, являются обычными символами, по которым происходит поиск и сопоставление строки.

В начале указывается, что необходимо получить выражение в скобках <>, а затем внутри этих скобок ожидается слово, т. е. последовательность символов верхнего или нижнего регистра без разделителей и пробелов; *\w+* позволяет указать, что ожидаются символы английского алфавита в количестве одной единицы или более.

Так как слово в открывающем теге должно быть равно слову в закрывающем, то используется **группировка** с помощью круглых скобок *()*. **Группировка** позволяет выделить подшаблон в шаблоне для того, чтобы, например, указать, что этот подшаблон повторяется несколько раз. В данном случае в закрывающем теге указывается *"\1"*, чтобы проверять, что слово в начале соответствует группе под номером один.

Кроме того, так как символ *"\"* используется в синтаксисе регулярных выражений, то для учета самого символа *"\"* в выражении его необходимо **экранировать**, т. е. для указания символа *"\"*, необходимо написать *"\\"*, по аналогии со знаком *"%"* в форматном вводе/выводе. Например, для поиска строки, которая имеет подстроку "\0": *"Надо ставить \0 в конце символьного массива, чтобы считать его строкой."*, надо написать следующее регулярное выражение:

.\\0.**

Данное выражение описывает любую строку, содержащую *"\0"*, так как **спецификатор "."** означает один любой символ, а **спецификатор "*"** означает любое количество указанных перед данным спецификатором символов.

Между тегами ожидаются не только слова, но и пробелы и табуляции, для этого используются **наборы**, которые указываются в квадратных скобках

[]. В таких наборах можно перечислить, какие символы могут содержаться в подстроке в независимости от их порядка, также как и в форматной строке ввода/выводы. В данном примере в набор включены символы алфавитов *\w* и символы-разделители *\s*.

Допустим есть следующий текст с тегами в файле **example.txt**:

<note>
<to>Friends<\to>
<from>James<\from>
<heading>Reminder note<\heading>
<body>Don't forget me this weekend!<\body>
<\note>

Тогда с помощью написанного нами регулярного выражения и утилиты *grep* можно написать команду для проверки содержимого файла на наличие описанных ранее тегов и текста между тегами:

grep -P '<(\w+)>[\w\s]+<\\1>' ./example.txt

В результате получим:

<to>Friends<\to>
<from>James<\from>
<heading>Reminder note<\heading>

Заметим, что строки *"<body>Don't forget me this weekend!<\body>"* не было найдено, так как предложение между тегами содержит знаки препинания, которые не учтены в регулярном выражении. Аналогично и не было найдено все содержимое файла, включая теги *"note"*.

Подробнее о расширениях регулярных выражений, их видах и применении в различных утилитах и языках программирования можно познакомиться в книге [5].

Синтаксис в языке Си (POSIX). В табл. 1.2 указаны основные спецификаторы и примеры, которые показывают синтаксис **POSIX** регулярных выражений и его смысл. С помощью этих спецификаторов можно указать, какие конкретно символы должны быть в строке и в каком порядке.

Таблица 1.2

Спецификатор	Значение
abc	Однозначная последовательность символов *abc*
123	Однозначная последовательность цифр 123
\d	Один любой символ цифры
\D	Один любой символ, не являющийся цифрой
.	Один любой символ

Окончание табл. 1.2

Спецификатор	Значение
[abc]	Квадратные скобки означают перечисление символов, которые могут быть на данном месте. В данном примере – один из символов *a*, *b* или *c*
[^abc]	Символ ^ означает отрицания – любой символ кроме. Выражение описывает один любой символ кроме *a, b*, или *c*
[a-z]	Любой символ от *a* до z в алфавитном порядке в нижнем регистре
[0-9]	Любая цифра от 0 до 9
\w	Любой символ латинского алфавита
\W	Любой символ не из латинского алфавита
\s	Любой пробельный символ
\S	Любой символ, кроме пробельных (символов табуляций)
^	Означает начало строки
$	Означает конец строки
(...)	Круглые скобки позволяют сгруппировать символы внутри. Группы позволяют указывать повторения выражения в строки. С помощью *\1 \2 \3* … можно определить уже описанную ранее группу под порядковым номером 1, 2 или 3
(abc|def)	Означает детектирование либо одного выражения *abc*, либо другого *def*

Скобки используются в синтаксисе регулярных выражений, поэтому для того чтобы указать в тексте символ скобки, нужно **экранировать** этот символ *"\("*. Экранирование позволяет отличить, что символ используется в строке, которая описывается шаблоном, а не в синтаксисе регулярных выражений. Экранировать необходимо следующие символы: ***"?^$.*(){}[]"***.

В табл. 1.3 указаны коды для указания количества символов, которые определены любым выражением из табл. 1.2.

Таблица 1.3

Код	Пример	Значение
{m}	*a{2}* – 2 подряд идущих буквы *a*	Выражение повторяется *m* раз
{m,n}	*a{2, 5}* – от 2 до 5 подряд идущих буквы *a*	Выражение повторяется в промежутке от *m* до *n* раз
*	*a** – 0 или более подряд идущих буквы *a*	Выражение повторяется 0 или более раз
+	*a+* – 1 или более подряд идущих буквы *a*	Выражение повторяется 1 или более раз
?	*a** – 0 или 1 буква *a*	Выражение встречается 0 или 1 раз

Примеры применения регулярных выражений. В предыдущем разделе были описаны коды, с помощью которых можно описать строку любого вида. В табл. 1.4 рассмотрим ряд примеров регулярных выражений.

Таблица 1.4

Пример	Подходящие строки
^(19|20)\d\d([-/.])(0[1-9]|1[012])\2(0[1-9]|[12][0-9]|3[01])$	Дата (не учитывается количество дней в месяце, високосный год, а также подходит только для дат 20-го и 21-го века): 1. 2022-02-30 2. 2022/02/30 3. 2022.02.30
(cat|dog){3}	1. catcatcat 2. dogdogdog 3. catdogcat 4. dogcatcat
(cat|dog){3} abc \1	1. catcatcat abc catcatcat 2. dogdogdog abc dogdogdog 3. catdogcat abc catdogcat 4. dogcatcat abc dogcatcat 5. whateverdogcatcat abc dogcatcat В 5 случае с помощью регулярного выражения можно найти подстроку, которая соответствует пункту 4

Использование регулярных выражений в программе на языке Си. В стандартной библиотеке языка Си реализован заголовочный файл ***regex.h***, который содержит функции для применения регулярных выражений в программе.

Основные функции:

• *int regcomp(regex_t *restrict_preg, const char *restrict_regex, int cflags)* – функция компиляции регулярных выражений;

• *int regexec(const regex_t *restrict_preg, const char *restrict_string, size_t nmatch, regmatch_t pmatch[restrict], int eflags)* – функция проверки строки по регулярному выражению;

• *size_t regerror(int errcode, const regex_t *restrict_preg, char *restrict_errbuf, size_t errbuf_size)* – функция вывода ошибок в строковом виде;

• *void regfree(regex_t *preg)* – функция очистки регулярного выражения для повторного использования (перекомпиляции).

Функция компиляции регулярного выражения выделяет память и подготавливает алгоритм проверки регулярного выражения из строки-шаблона, которая передана на вход. Обратим внимание на то, что функция regcomp принимает структуру для самого регулярного выражения *restrict_preg*, строку-шаблон *regex* и флаг, показывающий формат регулярного выражения. Есть следующие флаги:

• **REG_EXTENDED** – использование POSIX регулярных выражений;

• **REG_ICASE** – флаг игнорирования регистра;

• **REG_NOSUB** – игнорирует поиск групп и местоположения найденного шаблона, который записывается в структуру *regmatch_t* функцией *regexec*;

• **REG_NEWLINE** – игнорирует символы начала и конца строки и проверку шаблона.

Все указанные выше флаги можно использовать, применив побитовое ИЛИ, например:

regexec(preg, string, nmatch, pmatch, REG_EXTENDED | REG_NOSUB).

Функция очистки, в свою очередь, очищает всю выделенную память под регулярное выражение.

Функция *regexec* позволяет применить регулярное выражение к строке. Данная функция принимает 5 аргументов:

• *const regex_t *restrict_preg* – скомпилированное регулярное выражение, которое записано в структуру;

• *const char *restrict_string* – строка, в которой будет производиться поиск шаблон;

• *size_t nmatch* – количество групп, которые необходимо найти и записать в массив pmatc;

• *regmatch_t pmatch[restrict]* – структура, содержащая индексы начала найденного шаблона в полученной строк;

• *int eflags* – флаги для дополнительной конфигурации поиска шаблона.

Рассмотрим следующую программу:

```
#include <regex.h>
#include <stdio.h>
#define REGEX "(.+)\\.([a-zA-Z]+)$"

int main(){
    regex_t regex_comp;
    int comp = regcomp(&regex_comp, REGEX, REG_EXTENDED);
    if (comp != 0) {
        printf("Compilation error.\n");
        return 0;
    }
    char str1[] = "file1.txt";
    char str2[] = "file2.txt.csv";

    regmatch_t regex_groups[3];
    if (!regexec(&regex_comp, str1, 3, regex_groups, 0)) {
        str1[regex_groups[2].rm_so-1] = '\0';
        printf("Filename: %s; extension: %s\n", str1 + re-
gex_groups[1].rm_so, str1 + regex_groups[2].rm_so);
```

```
    }
    if (!regexec(&regex_comp, str2, 3, regex_groups, 0)) {
        str2[regex_groups[2].rm_so-1] = '\0';
        printf("Filename: %s; extension: %s\n", str2 + re-
gex_groups[1].rm_so, str2 + regex_groups[2].rm_so);
    }
    regfree(&regex_comp);
    return 0;
}
```

Данная программа проверяет полученную строку на соответствие шаблону: *"(.+)\\.([a-zA-Z]+)$"*. Данный шаблон описывает строку, содержащую название файла, где само название состоит из любых символов, а после точки идет расширение, состоящее только из латинских букв. После расширения всегда идет конец строки. Если строка соответствует шаблону, то программа выводит отдельно название, отдельно расширение файла. Для того чтобы выводилось только название, между расширением и названием файла вставляется *'\0'*.

Вывод программы:

Filename: file1; extension: txt

Filename: file2.txt; extension: csv

Ограничение на применение регулярных выражений. Регулярные выражения не всегда являются лучшим выбором, так как основаны на достаточно сложных и ресурсоемких алгоритмах. Зачастую, для решения тривиальных задач (например, для идентификации цифр в строке или поиска заранее известных подстрок в строке) лучше использовать функции стандартной библиотеки языка. В Си заголовочный файл *string.h* содержит достаточный набор стандартных функций для решения простых задач.

Регулярные выражения усложняют код программы, поэтому чрезмерное использование регулярных выражений приводит к коду, который сложно читать и понимать человеку. Так как регулярные выражения могут включать в себя огромное количество условий, то они могут быть непонятны человеку, а как следствие их сложно поддерживать и искать в них ошибки.

Кроме того, стандартные библиотечные функции хорошо оптимизированы и в некоторых случаях будут работать быстрее, чем регулярные выражения. Это особенно критично, если программа должна обрабатывать тексты, состоящие из множества тысяч строк и символов.

Скорость работы регулярных выражений также зависит от самого выражения, написанного разработчиком. Так как для одного и того же шаблона

можно написать несколько вариаций регулярных выражений, то и алгоритм, который лежит в основе этих выражений, будет работать по-разному, поэтому от написания регулярного выражения зависит скорость его работы. Подробнее о скорости работы регулярных выражений по сравнению со строковым сравнением можно прочитать в статье [6].

1.3. Задание к лабораторной работе № 1

На вход программе подается текст, представляющий собой набор предложений, начинающихся с новой строки. Текст заканчивается предложением *"Fin."* В тексте могут встречаться *IPv4* адреса. Требуется, используя регулярные выражения, найти все адреса, которые есть в тексте. Если предложение содержит корректный *IPv4* адрес, то гарантируется, что после нее будет символ переноса строки.

Формат написания *IPv4* адреса [1]:

• *IPv4* – адрес в десятичной форме;

• каждое число адреса – один байт (от 0 до 255);

• каждое число адреса не имеет ведущих нулей;

• разделителем между числами является точка;

• после адреса идет символ переноса строки.

1.3.1. Описание последовательности выполнения работы

Для выполнения поставленной задачи в первую очередь необходимо написать регулярное выражение, по которому будет производиться поиск адресов.

IPv4 адрес состоит из 4 байтов, каждый из которых содержит байт информации в промежутке от 0 до 255. Для проверки байта можно написать следующее регулярное выражение: *“[0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5]”*.

Данное регулярное выражение проверяет последовательно следующие интервалы чисел:

• 0–9;

• 10–99;

• 100–199;

• 200–249;

• 250–255.

Теперь необходимо находить и проверять, что после первых трех таких блоков идут точки. Это можно сделать с помощью группировки и указания спецификатора повторений:

“([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\\.){3}”

Обратим внимание, что для указания точки используется \\., а не \., так как \ используется в самом языке C и требует экранирования.

Последним шагом будет проверка последнего байта и символа переноса строки. Таким образом, получим следующее регулярное выражение:

"(([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\\.){3}([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-9]|25[0-5])\n"

Теперь перейдем к описанию программы для решения задачи. В начале файла с исходным кодом объявляем необходимые заголовочные файлы, а также необходимые макроопределения с регулярным выражением и размером буфера:

```
#include <regex.h>
#include <stdio.h>
#include <string.h>
#define N 500
#define IPV4_REGEX "(([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-
9]|25[0-5])\\.){3}([0-9]|[1-9][0-9]|1[0-9][0-9]|2[0-4][0-
9]|25[0-5])\n"
#define FINAL_WORD "Fin.\n"
```

Для поиска адресов реализована следующая функция:

```
void read_and_match(regex_t *regex_comp){
    char sentence[N];
    regmatch_t regex_groups;
    while(strcmp(fgets(sentence, N, stdin), FINAL_WORD)) {
        if (!regexec(regex_comp, sentence, 1, &regex_groups, 0))
{
            printf("%s", sentence + regex_groups.rm_so);
        }
    }
}
```

Данная функция принимает скомпилированное регулярное выражение и проверяет каждую входную строку на наличие *IPv4* адреса. Если адрес был найден, то он выводится в стандартный поток вывода.

Функция *main* отвечает только за компиляцию регулярного выражения и проверку на ошибки при компиляции, а также вызов функции поиска адресов.

```
int main(){
    regex_t regex_comp;
    int comp = regcomp(&regex_comp, IPV4_REGEX, REG_EXTENDED);
    if (comp != 0) {
        printf("Compilation error.\n");
        return 0;
    }
```

```
    read_and_match(&regex_comp);
    regfree(regex_comp);
    return 0;
}
```

1.3.2. Пример выполнения задания на защиту

Напишите программу, которая считывает текст длиной не более 1000 символов. В тексте необходимо найти все номера телефонов и вывести на экран строку: *“Ru: <9-значный номер телефона>”* или *“Not Ru: <9-значный номер телефона>”*. В тексте номер телефона имеет вид: *“+<Код страны>(<код оператора>)<3 цифры>-<2 цифры>-<2 цифры>”*.

Код страны – ненулевое число, содержащее не более 4 цифр без ведущих нулей (код Российской Федерации – 7). Код оператора – любое трехзначное число.

Листинг программы, решающей данную задачу:

```
#include <regex.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define N 1000
#define GROUPS_NUM 6
#define REGEX "\\+([1-9][0-9]{0,3})\\(([0-9]{3})\\)([0-9]{3})-
([0-9]{2})-([0-9]{2})[^0-9]"
int main(){
    regex_t regex_comp;
    int comp = regcomp(&regex_comp, REGEX, REG_EXTENDED);
    if (comp != 0) {
        printf("Compilation error.\n");
        return 0;
    }
    char sentence[N];
    fgets(sentence, N, stdin);
    regmatch_t regex_groups[GROUPS_NUM];
    char *cur_ptr = sentence;
    while(!regexec(&regex_comp, cur_ptr, GROUPS_NUM, re-
gex_groups, 0)) {
        if(atoi(sentence + regex_groups[1].rm_so) == 7)
            printf("Ru: ");
        else
            printf("Not Ru: ");
        printf("+%d%d%d%d%d\n", atoi(cur_ptr + re-
gex_groups[1].rm_so), atoi(cur_ptr + regex_groups[2].rm_so),
```

```
atoi(cur_ptr + regex_groups[3].rm_so), atoi(cur_ptr + re-
gex_groups[4].rm_so), atoi(cur_ptr + regex_groups[5].rm_so));
        cur_ptr += regex_groups[0].rm_eo;
    }

    return 0;
}
```

1.4. Вопросы для самоконтроля

1. Каким регулярным выражением можно описать имя человека в формате Имя.Фамилия, если считать, что и имя, и фамилия могут быть любой длины и состоят из символов латинского алфавита?

2. Что такое группа в регулярных выражениях?

3. Каким регулярным выражением можно описать строку без букв верхнего регистра?

Лабораторная работа № 2.
ЛИНЕЙНЫЕ СПИСКИ

2.1. Цель и задачи

Целью работы является освоение работы с линейными списками.

Для достижения поставленной цели требуется решить следующие задачи:

1. Ознакомиться со структурой данных «список».
2. Ознакомиться с операциями, используемыми для списков.
3. Изучить способы реализации этих операций на языке Си.
4. Написать программу, реализующую двусвязный линейный список и решающую задачу в соответствии с индивидуальным заданием.

2.2. Основные теоретические сведения

2.2.1. Указатель на структуру

В языке Си можно определять указатели как на объекты стандартных типов, так и на структуры. Например, указатель на структуру *Node* выглядит так:

```
struct Node* nodeElement;
```

Для того чтобы обращаться к полям структуры, если имеется указатель на структуру, используется оператор *“->”*:

```
struct Node* nodeElement = …;

nodeElement->field = 5; // присваивание полю field структуры
Node значения 5
```

2.2.2. Линейные списки

Перед тем как задать определение линейного списка, введем пару терминов, которые будут использоваться далее:

• Узел – один элемент из списка, который связан с другими элементами;

• Голова списка (head) – первый элемент в списке;

• Хвост (tail) – все элементы списка, идущие после головы.

Линейный односвязный список – это структура данных, представляющая собой последовательность узлов, каждый из которых хранит какие-то полезные данные и указатель на следующий элемент. Важно, что в памяти элементы не находятся последовательно, в отличие от массивов (см. рис. 2.1).

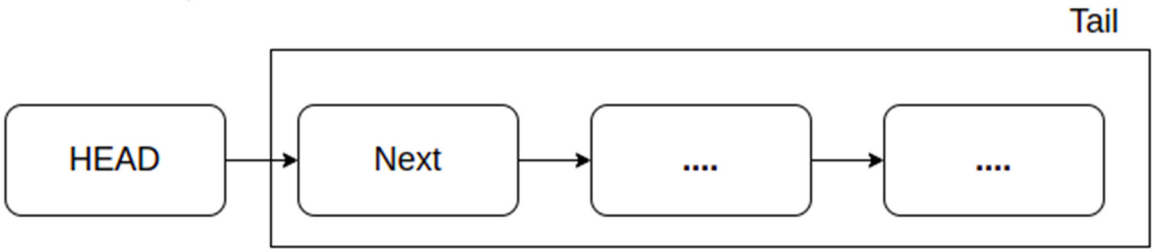

Рис. 2.1. Линейный односвязный список

Список можно рассматривать как более гибкую альтернативу массиву: порядок элементов в списке не связан с их расположением в памяти. Это порождает, например, то, что такие операции как вставка и удаление не связаны со сдвигами остальных элементов, однако все операции со списком являются исключительно последовательными – прямой доступ к элементам списка (доступ по индексу) невозможен.

Для простоты примера будет использоваться ключевое слово typedef, которое служит для определения имен новых типов данных. На самом деле здесь не создается новый тип данных, а определяется новое имя существующему типу.

Рассмотрим пример создания линейного односвязного списка, состоящего из двух элементов. Пусть дана структура *Point* следующего вида, которая описывает координаты некоторой точки на плоскости и радиус:

```
typedef struct Point {
   int x;
   int y;
   float radius;

   Point *next; // указатель на следующий элемент
} Point;
```

Рассмотрим функцию main, в которой:

– вводится количество точек;

– выделяется динамическая память для массивов, каждый из которых имеет значения компоненты точки;

– создается список точек при помощи функции createPointList;

– создается элемент точки при помощи функции createPoint с координатами (1, 1, 3.5) и добавляется в конец списка;

– выводится на экран список;

– освобождение памяти.

Листинг функции:

```
int main() {
    int length;
    fscanf(stdin, "%d\n", &length);

    int *arrayX = (int *) malloc(sizeof(int *) * length);
    int *arrayY = (int *) malloc(sizeof(int *) * length);
    float *arrayR = (float *) malloc(sizeof(float) * length);
    for (int i = 0; i < length; i++) {
        scanf("%d %d %f\n", &arrayX[i], &arrayY[i], &arrayR[i]);

    }
    Point* head = createPointList(arrayX, arrayY, arrayR,
length);
    Point *elementForPush = createPoint(1, 1, 3.5);
    push(head, elementForPush); // добавляем в конец списка эле-
мент
    Point *cur = head;

    while (cur != NULL) { // выводим все элементы списка, начиная
с головы
        printf("%d %d %f\n", cur->x, cur->y, cur->radius);
        cur = cur->next;
    }

    cur = head;

    while (cur != NULL) { // освобождение памяти
        struct Point *free_el = cur;
        cur = cur->next;
        free(free_el);
    }

    return 0;
}
```

Получился линейный список из двух элементов. В данном коде головой списка является *head*, так как он является первым элементом списка.

Для работы со списками используются основные функции:

– вставка элемента в конец списка/голову списка;

– вставка после определенного элемента;

– определение количества элементов списка;

– удаление элемента из конца списка/из головы;

– удаление определенного элемента списка;

– создание элемента списка;

– создание списка из нескольких элементов.

Рекомендуем вам реализовывать все нужные функции работы со структурами отдельно в каждой функции, это делает код более читаемым и позволяет избежать дублирования.

Для удобной инициализации как одного элемента списка, так и различной длины, рассмотрим реализацию функций *createPoint* и *createPointList*, которые использовались в предыдущем примере:

• createPoint:

```
Point *createPoint(int x, int y, float radius) {
   Point *cur = (Point *) malloc(sizeof(Point));
   cur->x = x;
   cur->y = y;
   cur->radius = radius;
   cur->next = NULL;
   return cur;
}
```

• createPointList:

```
Point   *createPointList(int   *array_x,   int   *array_y,   float
*array_r, int length) {
   Point *head = NULL;
   Point *list = createPoint(*array_x, *array_y, *array_r);
   list->next = NULL;
   head = list;

   for (int i = 1; i < length; i++) {
       list->next   =   createPoint(array_x[i],   array_y[i],   ar-
ray_r[i]);
       list = list->next;
       list->next = NULL;
   }
   return head;
}
```

2.2.3. Вставка элемента в список

Объявляем функцию ***push***, которая будет вставлять элемент в конец списка. Ее сигнатура выглядит следующим образом:

```
void push(Point *head, Point *element);
```

где *head* – это голова списка, а *element* – это объект структуры *Node*, который будет вставлен в конец списка. Изменим функцию *main* для демонстрации работы функции *push:*

```
int main(){
   Point * p1 = (Point*)malloc(sizeof(Point));
   Point * p2 = (Point*)malloc(sizeof(Point));
   // пример вставки элемента в список
   p1->x = 2; // используем оператор -> поскольку p1 – это ука-
затель на структуру Node
   p1->y = 2;
   p1->r = 2.5;

   p2->x = 5;
   p2->y = 5;
   p2->r = 5.5;

   p1->next = p2; // связываем указатель на следующий элемент у
p1 и p2
   p2->next = NULL; // указатель на следующий элемент для p2
NULL
   // пример вставки элемента в список при помощи функции push
   Point* element_for_push = (Point*)malloc(sizeof(Point));
   element_for_push->x = 1;
   element_for_push->y = 1;
   element_for_push->r = 1.5;
   push(&p1, element_for_push); // добавляем в конец списка эле-
мент

   struct Point* cur = p1;

   while (cur != NULL){ // выводим все элементы списка, начиная
с головы
        printf("%d %d %f\n", cur->x, cur->y, cur->radius);
        cur = cur->next;
   }

   cur = p1;
```

```
    while (cur != NULL){ // освобождаем память
        struct Point* free_el = cur;
        cur = cur->next;
        free(free_el);
    }

    return 0;
}
```

Отличие от предыдущего *main* из предыдущего раздела заключается в том, что вставка элемента происходит в функции *push.* Рассмотрим функцию *push*:

```
void push(Node **head, Node *element) {
    Node *cur;
    cur = *head;

    if (*head == NULL) { // случай, если head указывает на пустой
список
        *head = element;
        return;
    }

    while (cur->next != NULL) {
        cur = cur->next;
    }
    cur->next = element;
    element->next = NULL;
}
```

Алгоритм функции *push* следующий:

1. Если голова указывает на *NULL*, то присваиваем голове элемент и на этом завершается функция push.

2. Если нет, то переходим к следующему элементу, пока указатель на следующий элемент не будет равен *NULL.*

3. У последнего элемента меняем указатель на следующий элемент с *NULL* на добавляемый элемент.

4. У добавленного элемента меняем указатель на следующий элемент на *NULL.*

2.2.4. Двусвязный список

Двусвязный список – это список с возможностью идти в обе стороны, в отличие от односвязного. Каждый элемент двусвязного списка имеет указатель на предыдущий элемент и на следующий. Схема двусвязного списка представлена на рис. 2.2.

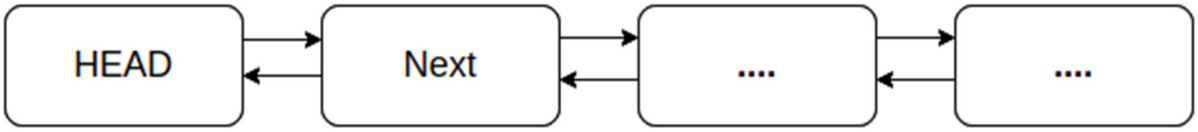

Рис. 2.2. Схема двусвязного списка

Циклический список – это список, в котором можно из хвоста попасть в голову, а из головы попасть в хвост. Схема циклического двусвязного списка на рис. 2.3.

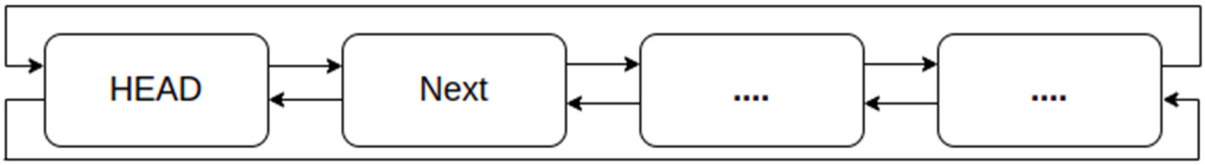

Рис. 2.3. Схема двусвязного циклического списка

Рассмотрим обновленную структуру *Point*:

```
typedef struct Point {
   int x;
   int y;
   float radius;

   Point *next; // указатель на следующий элемент
   Point *prev; // указатель на прошлый элемент
} Point;
```

Рассмотрим процесс добавления элемента в середину списка (рис. 2.4).

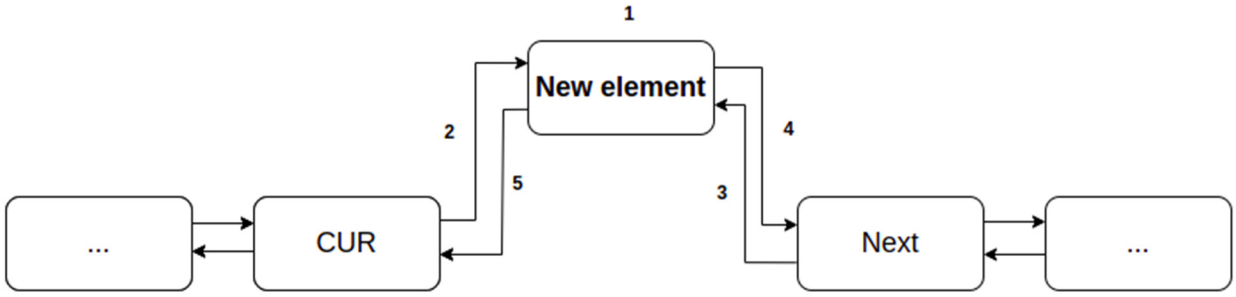

Рис. 2.4. Схема добавления элемента в список

Где CUR – это элемент, после которого надо вставить новый элемент списка, а Next – это указатель на элемент, на который ранее указывал CUR.

Алгоритм добавления элемента:

1. Создание узла добавляемого элемента и заполнение его поля данных.

2. Переустановка указателя «следующий» узла, предшествующего добавленному, на добавляемый узел.

3. Переустановка указателя «предыдущий» узла, следующего за добавляемым, на добавляемый узел.

4. Установка указателя «следующий» добавляемого узла на следующий узел (тот, на который указывал предшествующий узел).

5. Установка указателя «предыдущий» добавляемого узла на узел, предшествующий добавляемому (узел, переданный в функцию).

2.3. Задание к лабораторной работе № 2

Создайте двунаправленный список музыкальных композиций *MusicalComposition* для работы со списком. Структура элемента списка (тип – *MusicalComposition*):

1) *name* – строка неизвестной длины (гарантируется, что длина не может быть больше 80 символов), название композиции;

2) *author* – строка неизвестной длины (гарантируется, что длина не может быть больше 80 символов), автор композиции/музыкальная группа;

3) *year* – целое число, год создания.

Функция для создания элемента списка (тип элемента *MusicalComposition*):

• *MusicalComposition* createMusicalComposition(char* name, char* author, int year)*.

Функции для работы со списком:

• *MusicalComposition* createMusicalCompositionList(char** array_names, char** array_authors, int* array_years, int n); // создает список музыкальных композиций MusicalCompositionList, в котором:*

– *n* – длина массивов *array_names*, *array_authors*, *array_years*;

– поле *name* первого элемента списка соответствует первому элементу списка *array_names* (*array_names[0]*);

– поле *author* первого элемента списка соответствует первому элементу списка *array_authors* (*array_authors[0]*);

– поле *year* первого элемента списка соответствует первому элементу списка *array_authors* (*array_years[0]*).

Аналогично для второго, третьего, ... *n*-1-го элемента массива.

Длина массивов *array_names*, *array_authors*, *array_years* одинаковая и равна *n*, это проверять не требуется.

Функция возвращает указатель на первый элемент списка:

– *void push(MusicalComposition* head, MusicalComposition* element);* // добавляет element в конец списка musical_composition_list;

– *void removeEl (MusicalComposition* head, char* name_for_remove);* // удаляет элемент element списка, у которого значение name равно значению name_for_remove;

– *int count(MusicalComposition* head);* //возвращает количество элементов списка;

– *void print_names(MusicalComposition* head);* //Выводит названия композиций.

2.3.1. Описание последовательности выполнения работы

В данной работе необходимо написать программу на языке Си и составить отчет. Задание и описание последовательности лабораторной работы:

1. Создайте голову элемента, сохраняем указатель на голову.
2. Напишите функцию добавления элемента в список.
3. Напишите функцию, которая наполняет список во входным данным.
4. Реализуйте функцию *count*.
5. Реализуйте функцию *removeEl*.
6. Реализуйте функцию *print_names*.
7. Проверьте, очищается ли вся память.

2.3.2. Пример выполнения задания на защиту

Задание: напишите функцию, которая создает линейный односвязный список из 10 элементов. Элемент списка – структура следующего вида:

```
struct Node{
   int value;
   struct Node* next;
};
```

Программа на языке Си:

```
#include <stdio.h>
#include <stdlib.h>
typedef struct Node{
   int x;
   int y;
   float r;
   struct Node* next;
}Node;
int main(){
   Node* head = (Node*)malloc(sizeof(Node));
   Node* tmp = (Node*)malloc(sizeof(Node));
   head->next = tmp;
```

```
    for(int i = 1; i < 10; i++){
        tmp->next = (Node*)malloc(sizeof(Node));
        tmp->next->next = NULL;
        tmp = tmp->next;
    }
    // освобождение памяти
    while (head) {
        tmp = head;
        head = head->next;
        free(tmp);
    }
}
```

2.4. Вопросы для самоконтроля

1. В чем отличие линейного списка от массива?
2. Какие обязательные поля должны быть в каждом узле двусвязного линейного списка?
3. Опишите алгоритм вставки в начало и конец двусвязного списка.
4. Опишите алгоритм вставки в конец циклического односвязного списка.

Лабораторная работа № 3.
РЕКУРСИЯ, ЦИКЛЫ, РЕКУРСИВНЫЙ ОБХОД ФАЙЛОВОЙ СИСТЕМЫ

3.1 Цель и задачи

Целью работы является освоение работы с рекурсивными функциями и файловой системой, а также ее рекурсивным обходом.

Для достижения поставленной цели требуется решить следующие задачи:

1) Ознакомиться с понятием рекурсии;
2) Освоить написание рекурсивных функций в языке Си;
3) Изучить работу с файловой системой в языке Си;
4) Написать программу для рекурсивного обхода всех файлов в папке в том числе во вложенных папках.

3.2. Основные теоретические сведения

3.2.1. Базовое понятие рекурсии

Рекурсия – это вызов функции в ней самой же [7]. То есть если внутри выполнения функции A происходит вызов функции A, то это рекурсия. Далее представлена рекурсивная функция *foo()*:

```
void foo(int n){
   if(n == 0){ // условие выхода
       puts("End of recursion"); // Отсутствует рекурсивный вы-
зов функции foo()
   }else{
       printf("Recursive call with n=%d\n", n - 1);
       foo(n - 1); // Рекурсивный вызов функции foo()
   }
}
```

Условия, при которых не происходит рекурсивного вызова, называют *условиями выхода*. В приведенной функции условием выхода является проверка *if(n == 0)*. На рис. 3.1 изображены графические представления вызовов рекурсивной функции *foo()* для *n*, равного 3.

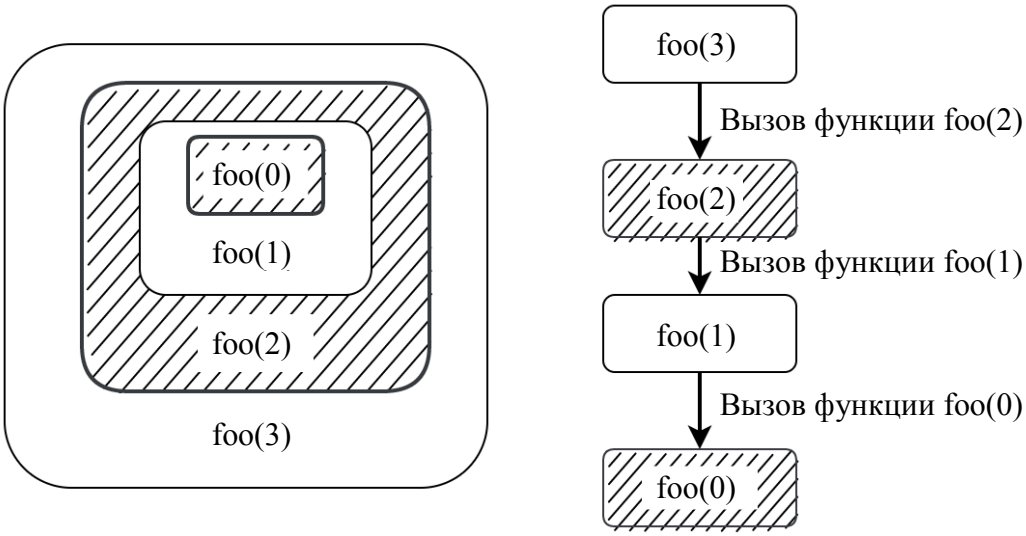

Рис. 3.1. Графическое представление вызовов рекурсивной функции *foo()* для *n*, равного 3

Применение рекурсии достаточно легко при решении задач с рекуррентной формулой. В таких задачах новое значение рассчитывается на основе одного или нескольких предыдущих вычисленных значений. Примером задачи с рекуррентными формулами для начинающих являются вычисление факториала и *n*-го числа Фибоначчи.

Далее будет рассмотрен пример с вычислением факториала. Функция для рекурсивного вычисления факториала может быть реализована следующим образом:

```
#include <stdio.h>

int factorial(int n){ // рекурсивная функция
   if(n == 0) return 1; // условие выхода
```

```
    return n * factorial(n - 1); // рекурсивный вызов
}

int main()
{
    printf("The factorial of 10 is %d\n", factorial(10));
    return 0;
}
```

Может показаться, что если код одинаковый, то при рекурсивном вызове переменные с одинаковыми именами будут перезаписаны, однако это не так. У каждой функции свой набор локальных переменных, который не зависит от других функций. Поэтому когда происходит рекурсивный вызов, то переменные в вызывающей функции останутся неизменными.

На рис. 3.2 изображена схема выполнения при вызове функции *factorial(3)*. На этапах 1–4 при создании нового экземпляра функции *factorial* будет создана новая локальная независимая от других переменная *n*.

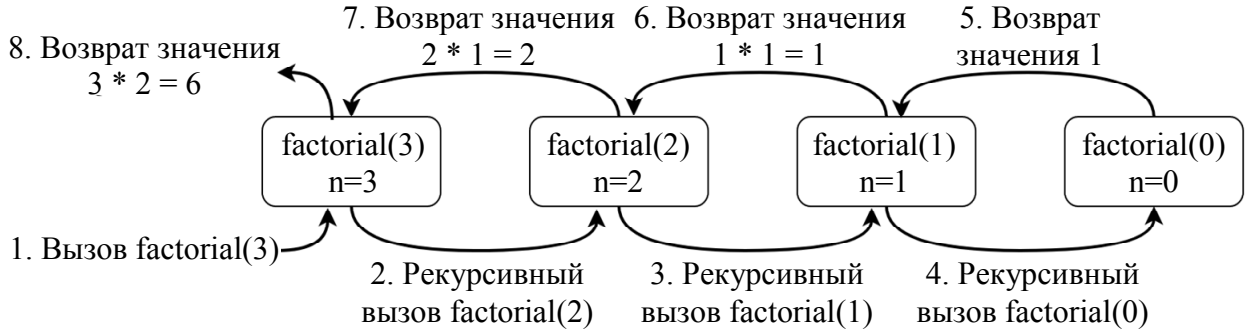

Рис. 3.2. Схема выполнения при вызове функции factorial(3)

Характеристикой рекурсии является ее *глубина* – количество одновременно запущенных экземпляров рекурсивной функции. В примере на рис. 3.2 при выполнении экземпляра функции *factorial(2)* глубина рекурсии равняется двум, а при *factorial(0)* – четырем. Дополнительно можно выделить *максимальную глубину рекурсии* – максимальная возможная глубина рекурсии. В большинстве случаев максимальная глубина не является константой, однако ее можно выразить формулой, например, для функции факториала максимальная глубина составляет $N + 1$, где N – число для которого вычисляется факториал.

3.2.2. Составление рекурсивной функции

При изучении рекурсии возникает сложность самостоятельного написания рекурсивных функций. Для преодоления данной сложности будет более подробно разобран пример составления рекурсивной функции, которая вы-

водит натуральное число в двоичной системе счисления. Алгоритм перевода числа в двоичную систему счисления выглядит следующим образом:

1. Сохранить остаток от деления числа на 2.
2. Целочисленно разделить число на 2.
3. Пока число больше нуля повторять пункты 1 и 2.
4. Записать полученные остатки в обратном порядке.

Например, для числа 10 получатся следующие остатки: 0, 1, 0, 1. После записи их в обратном порядке получается запись в двоичной системе счисления: 1010.

Написание рекурсивной функции лучше начинать с выделения части, которая будет выполняться несколько раз. В данной задаче это операции деления и взятия остатка: остаток необходимо вывести, а для уменьшенного в два раза числа сделать рекурсивный вызов функции. Условием выхода из рекурсии является то, что число равняется нулю. Таким образом функция приобретает следующий вид:

```
void print_bin(int x){
   if(x == 0) return;  // условие выхода
   printf("%d", x % 2);  // вывод остатка
   print_bin(x / 2);  // рекурсивный вызов
}
```

Однако данная функция выводит остатки в прямом порядке, а необходимо в обратном. Можно использовать массив, чтобы сохранить остатки и потом вывести его элементы в обратном порядке, однако рекурсия позволяет это сделать более простым образом: поменять местами рекурсивный вызов и вывод остатка деления. Таким образом сначала будет выведен остаток числа, уменьшенного в два раза, а только затем текущего числа, что приведет к тому, что остатки будут выведены в обратном. Далее представлена исправленная версия функции:

```
void print_bin(int x){
   if(x == 0) return;  // условие выхода
   print_bin(x / 2);  // рекурсивный вызов
   printf("%d", x % 2);  // вывод остатка
}
```

При составлении функции важно обращать внимание на порядок рекурсивных вызовов (как в разобранном примере). Также важно обращать внимание на то, какие аргументы передаются при рекурсивном вызове. Например, рекурсивный вызов мог быть *dec_to_bin(x)*, что привело бы к тому, что функция попала бы в бесконечную рекурсию и аварийно завершилась. Не менее

важным являются условия выхода, которые должны присутствовать и проверять исключительные случаи вызова функции. Например, функция вычисления факториала из раздела 3.2.1 не учитывает, что переданный аргумент *n* может быть отрицательным. Поэтому при вызове функции с отрицательным значением условие выхода никогда не выполнится и программа аварийно завершит свою работу.

Как и любой метод, подход или инструмент, рекурсия не лишена недостатков. Каждый новый вызов функции требует дополнительного места в «стековой памяти», которая выделяется при запуске программы, для хранения локальных переменных. При достаточно большой глубине рекурсии «стековая память» может закончиться, что вызовет ошибку переполнения стека (Stack Overflow) и аварийное завершение программы. Также каждый вызов функции требует копирования аргументов функции и передачи управления в другую функцию. Данные события требуют дополнительного времени для выполнения, что увеличивает время работы программы, а также не позволяет компилятору применить часть оптимизаций.

Таким образом получается следующее: применение рекурсии позволяет проще и понятнее писать код, однако снижается скорость выполнения программы и может увеличиться используемая память. Если выбор сделан в сторону рекурсии, то нужно оценить максимальную глубину рекурсии, при необходимости изменить размер стековой памяти при помощи флагов компиляции и не допустить возникновения бесконечной рекурсии, т. е. предусмотреть все условия выхода на граничные случаи значения аргументов и избежать логических ошибок составления рекурсивной функции (например, рекурсивный вызов с исходными аргументами).

3.2.3. Функции для работы с файлам и директориями в языке Си

Функции для работы с файлами определены в заголовочном файле ***stdio.h***. Для работы с файлами используется файловый поток, реализованный структурой *FILE*. Напрямую работа с данной структурой не производится, поэтому содержимое структуры рассмотрено не будет. Основными функциями для работы с файлами являются:

• *FILE * fopen(const char * filename, const char * mode)* – открывает файл с названием *filename* в режиме *mode* и возвращает указатель на файловый поток *FILE*;

• *int fclose (FILE * stream)* – закрывает файловый поток stream, полученный из *fopen()*.

Основными режимами открытия файла являются *“r”* и *“w”*: первый режим открывает файл на чтение, второй – на запись.

Работа с файлами может осуществляться также как и со стандартными потоками ввода и вывода. Для этого существуют функции ***fprintf***, ***fscanf***, ***fgetc***, ***fgets***, ***fputc***, ***fputs*** – они работают аналогично своим двойникам для стандартных потоков ввода и вывода, однако принимают дополнительный аргумент в виде указателя на файловый поток *FILE*. Также есть функции для чтения и записи бинарных данных: **fread** и **fwrite**. Использование функций для работы с файлами подробно рассмотрено в прил. 1. Далее приведен пример программы, которая считывает число из файла *“input.txt”*, умножает его на два и записывает результат в файл *“output.txt”*:

```
#include <stdio.h>

int main(){
    FILE *fin, *fout;
    int num = 0;
    fin = fopen("input.txt", "r"); // открытие файла на чтение
    if(fin){
        fscanf(fin, "%d", &num);
        fclose(fin); // закрытие файла
    }
    fout = fopen("output.txt", "w"); // открытие файла на запись
    if(fout){
        fprintf(fout, "%d", 2 * num);
        fclose(fout); // закрытие файла
    }
}
```

Определение структур и функций для работы с директориями находятся в заголовочном файле *dirent.h*. Для работы с директориями используется поток директории, реализованный структурой *DIR* (по аналогии с файловым потоком *FILE*), которая используется в качестве аргумента для функций. Так как напрямую работа с данной структурой не производится, то ее структура не будет рассмотрена. Основными функциями для работы с директориями являются:

• *DIR *opendir(const char *dirname)* – открывает директорию *dirname* и возвращает указатель на поток директории *DIR*. Если не удалось открыть директорию, то возвращается *NULL*;

• *int closedir (DIR *dir)* – закрывает поток директории *dir*, который был получен из *opendir()*;

• *struct dirent *readdir (DIR *dirstream)* – считывает следующий элемент из потока директории *dirstream* и возвращает указатель на прочитанный элемент. Если в потоке больше не осталось элементов, то возвращается *NULL*.

Структура *dirent* содержит информацию о файле. Основную информацию содержат следующие поля:

• поле *d_name* (тип *char[]*) – имя файла, которое является строкой, заканчивающаяся символом конца строки;

• поле *d_type* (тип *unsigned char*) – тип файла, основными значениями являются:

DT_UNKNOWN – неизвестный тип файла;

DT_REG – обычный (регулярный) файл, который можно открыть на чтение/запись;

DT_DIR – директория.

Указатель, который возвращает функция *readdir()*, указывает на область памяти, связанную со структурой *DIR*. Данные в этой области памяти изменяются при каждом вызове функции *readdir()*. Поэтому если есть необходимость использовать данные из структуры dirent, то их необходимо предварительно скопировать.

Далее представлен код программы, которая выводит названия всех файлов в текущей директории:

```
#include <stdio.h>
#include <dirent.h>

int main(){
   DIR *dir = opendir("./"); // открытие директории
   if(dir){
       struct dirent *de = readdir(dir);   // получение следующе-
го элемента
       while (de){ // != NULL
           if(de->d_type == DT_REG) // элемент является файлом
               printf("File: %s\n", de->d_name);
           de = readdir(dir);
       }
       closedir(dir); // закрытие директории
   }else
       puts("Failed to open current directory!");

   return 0;
}
```

3.2.4. Обход вложенных директорий

Как было упомянуто ранее, директории могут содержать в себе не только файлы, но и другие директории, которые в свою очередь также могут содержать директории и т. д. Таким образом, получается *иерархия директорий* (рис. 3.3). В Linux и UNIX-подобных системах корневой директорией, которая не имеет родительской директории, является “/”. Посмотрев на графическое представление иерархии директорий, можно увидеть, что оно похоже на «перевернутое дерево»: в самом верху изображения находится корень “*current_dir*”, а от него исходит множество веток, например, “*nested_dir*” и “*nested_dir2*”, которые заканчиваются листьями (файлы “*file1*”, “*file2*”). Иерархия директорий является воплощением структуры данных дерева. Подробнее про деревья и другие структуры данных можно узнать в [8].

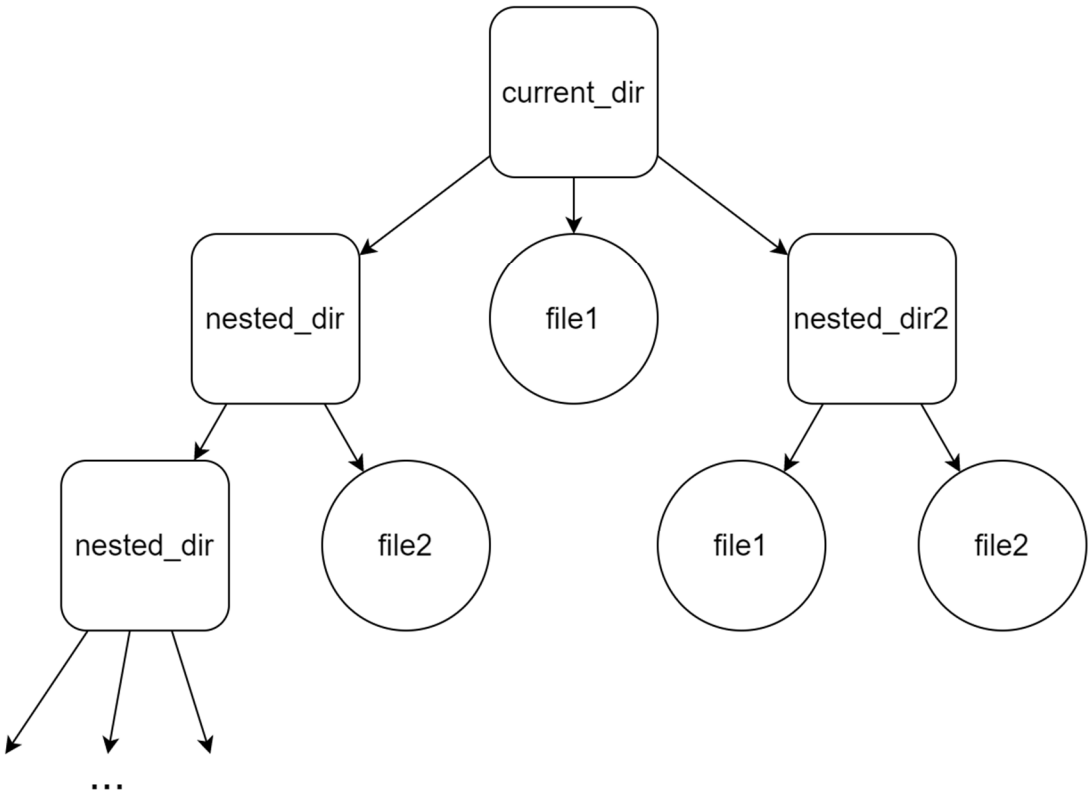

Рис. 3.3. Графическое представление иерархии директорий

Код, приведенный в предыдущем разд. 3.2.3, выводит названия файлов только в текущей директории, но не во вложенных директориях. Для обхода вложенных директорий необходимо выполнить тот же самый набор команд, но уже для вложенной директории. Сделать это можно при помощи рекурсивного вызова функции, которая обрабатывает содержимое директории. Далее представлен модифицированный пример кода для вывода файлов в директории, включая вложенные:

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <dirent.h>

char *pathcat(const char *path1, const char *path2){
   int res_path_len = strlen(path1) + strlen(path2) + 2; // определение длины новой строки с учетом символов / и символа конца строки
   char *res_path = malloc(res_path_len * sizeof(char)); // выделение памяти под новую строку
   sprintf(res_path, "%s/%s", path1, path2); // форматный вывод данных в строку
   return res_path;
}

void list_dir(const char *dir_name){
   DIR *dir = opendir(dir_name);
   if(dir){
       struct dirent *de = readdir(dir);
       while (de){
           if(de->d_type == DT_REG){
               // элемент директории - файл
               printf("File: %s\n", de->d_name);
           }else if (de->d_type == DT_DIR &&
                     strcmp(de->d_name, ".") != 0 &&
                     strcmp(de->d_name, "..") != 0){
               // элемент директории - директория
               char *new_dir = pathcat(dir_name, de->d_name); // соединение имён директорий
               list_dir(new_dir); // рекурсивный вызов
               free(new_dir);
           }
           de = readdir(dir);
       }
       closedir(dir);
   }else
       printf("Failed to open %s directory\n", dir_name);
}

int main(){
   list_dir(".");
   return 0;
}
```

Функция *list_dir()* повторяет код из разд. 3.2.4, за исключением строк кода, обрабатывающих элемент директории: добавился дополнительный блок *if*, который осуществляет рекурсивный вызов функции *list_dir()* для элемента, являющегося директорией. Для конкатенации имен родительской и вложенной директорий была использована *pathcat()*, которая выделяет память под новую строку и заполняет ее содержимое при помощи форматного вывода.

Важно обратить внимание на две проверки, которые сравнивают имя директории с *“.”* и *“..”*. Данные проверки связаны с тем, что каждая директория всегда содержит два дополнительных элемента: *“.”* (ссылка на текущую директорию) и *“..”* (ссылка на родительскую директорию). Если данную проверку не осуществлять, то программа окажется в бесконечном цикле, потому что, например, будет обходить только одну текущую директорию.

3.3. Задание к лабораторной работе № 3

Дана некоторая корневая директория, в которой может находиться некоторое количество папок, в том числе вложенных. В этих папках хранятся некоторые текстовые файлы. В корневой папке находится файл start.txt. Каждая строка данного файла содержит название какого-то файла. Необходимо вывести содержимое всех файлов, названия которых содержатся в файле *start.txt*, в формате *“File <название_файла>: <содержимое файла>”*. Если файл не удалось найти, то необходимо вывести *“File <название_файла> not found”*. Гарантируется, что все имена файлов уникальны.

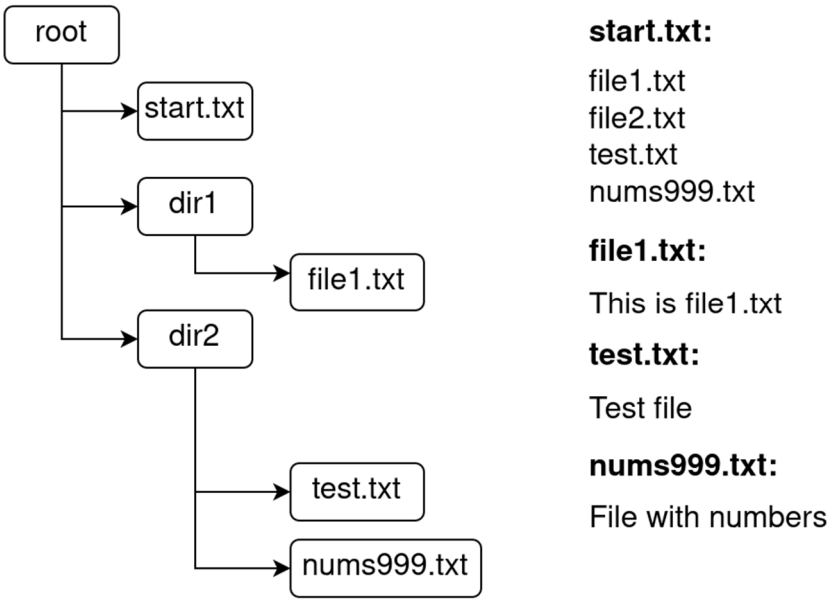

Рис. 3.4. Пример иерархии файлов и директорий и содержимого файлов для задания к лабораторной работе № 3.

Например, есть иерархия файлов и директорий и содержимого файлов, представленные на рис. 3.4. Программа находится в директории root. Для описанной примера результат работы программы будет следующий:

File file1.txt: This is file1.txt

File file2.txt not found

File test.txt: Test file

File nums999.txt: File with numbers

3.3.1. Описание последовательности выполнения работы

Выполнение лабораторной работы можно разбить на три задачи: чтение названия файлов из файла *start.txt*, поиск файла в директориях и вывод содержимого файла.

Хорошей практикой является реализация каждой задачи в отдельной функции и независимо (насколько это возможно) от других задач. Такая практика структурирует код и позволяет максимизировать переиспользование написанного кода. Соответственно для выполнения лабораторной работы необходимы следующие функции: функция *process_start_file()* для чтения названия файлов из файла start.txt, функция *find_file()* для поиска файла в директориях и функция *print_file_content()* для вывода содержимого файлов.

Для функции *find_file()* за основу можно взять ранее реализованную функцию *list_dir()*. В отличие от *list_dir()* функция *find_file()* будет принимать дополнительный аргумент, содержащий название файла для поиска. Также функция *find_file()* будет возвращать не *void*, а строку с путем до найденного файла или *NULL*, если файл не был найден. Итоговая функция выглядит следующим образом:

```
char *find_file(const char *dir_name, const char *filename){
    char *full_path_file = NULL;  // изначально файл не найден
    DIR *dir = opendir(dir_name);
    if(dir){
        struct dirent *de = readdir(dir);
        while (de){
            if(de->d_type == DT_REG && !strcmp(de->d_name, file-
name)){
                // файл найден
                full_path_file = pathcat(dir_name, filename);
            }else if (de->d_type == DT_DIR &&
                      strcmp(de->d_name, ".") != 0 &&
                      strcmp(de->d_name, "..") != 0){
```

```
                char *new_dir = pathcat(dir_name, de->d_name);
                // запись результата поиска во вложенной директо-
рии
                full_path_file = find_file(new_dir, filename);
                free(new_dir);
            }
            if(full_path_file)  // файл найден, завершение поиска
                break;
            de = readdir(dir);
        }
        closedir(dir);
    }else
        printf("Failed to open %s directory\n", dir_name);
    return full_path_file;  // возвращение результата поиска
}
```

Теперь необходимо реализовать функцию *print_file_content()*. Функция будет принимать путь до файла, содержимое которого необходимо вывести, и ничего не будет возвращать:

```
void print_file_content(const char *file_path){
    FILE *file = fopen(file_path, "r");   // открытие файла на
чтение
    if(!file){  // не удалось открыть файл
        printf("Failed to open %s file\n",file_path);
        return;
    }
    char data[SIZE];
    char *read_result;
    while((read_result = fgets(data, SIZE, file)) != NULL){   //
чтение данных
        printf("%s", data);  // вывод прочитанного содержимого
    }
    printf("\n");
    fclose(file);  // закрытие файла
}
```

Необходимо обратить внимание на условие выхода из цикла: функция *fgets* возвращает *NULL*, если был достигнут конец файла и не было прочитано ни одного символа. Таким образом, если *fgets* вернул *NULL*, то все данные из файла были прочитаны.

Осталось реализовать функцию *process_start_file()*, в которой будет происходить считывание названий файлов из файла start.txt. При чтении очеред-

ного названия файла будет происходить поиск файла при помощи функции *find_file()*, после чего будет печататься содержимое файла при помощи функции *print_file_content()*:

```
void process_start_file(){
    FILE *start_file = fopen("start.txt", "r");
    if(!start_file){
        puts("Failed to open start.txt file");
        return 0;
    }
    char *filename = malloc(STEP * sizeof(char));  // выделение
памяти под динамическую строку
    int index = 0;
    int max_size = STEP;
    char c = EOF;
    do{
        c = fgetc(start_file);
        if((c == '\n' || c == EOF) && index > 0){  // прочитана
непустая строка
            filename[index] = '\0';
            char *path = find_file(".", filename);  // поиск фай-
ла
            if(path){  // файл найден, вывод содержимого
                printf("File %s: ", filename);
                print_file_content(path);
            }else{  // файл не найден
                printf("File %s not found\n", filename);
            }
            free(path);
            index = 0;
        }else{
            filename[index++] = c;  // запись символа в строку
        }
        if(index >= max_size){  // увеличение размера строки
            max_size += STEP;
            filename = realloc(filename, max_size);
        }
    }while(c != EOF);  // проверка на достигнутый конец файла
    fclose(start_file);
}
```

Для завершения лабораторной работы необходимо в функции *main()* вызвать *функцию process_start_file()*.

3.3.2. Пример выполнения задания на защиту

Дана корневая директория *root*, в которой может находиться некоторое количество папок, в том числе вложенных. Необходимо вывести полные пути до файлов, отсортированные в порядке возрастания по сумме цифр в названии файла. Если сумма цифр одинаковая, то сравнение производить в лексикографическом порядке.

Например, для иерархии файлов и директорий, представленной на рис. 9, результат будет следующий:

Sum of digits: 0, path: ./root/dir2/file.txt
Sum of digits: 0, path: ./root/dir2/test.txt
Sum of digits: 1, path: ./root/dir1/file1.txt
Sum of digits: 6, path: ./root/dir1/my_number6.txt
Sum of digits: 27, path: ./root/dir2/nums999.txt

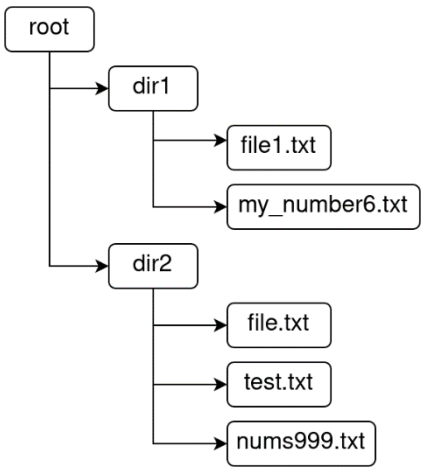

Рис. 3.5. Пример иерархии файлов и директорий для задания на защиту

Выполнение задания можно вновь разбить на две большие задачи: обход всех директорий для поиска файлов и сортировка полученных путей. Так как обход всех директорий будет осуществляться при помощи рекурсии, то данную задачу необходимо реализовать отдельной функцией. Сортировку полученных путей можно реализовать в функции *main()*.

Для того чтобы сортировать пути до файлов, их необходимо где-то сохранить. Для этого можно завести массив, который будет в себе хранить полные пути до файлов. Однако в таком случае при каждом сравнении необходимо будет выделять из полного пути файла его название и вычислять сумму цифр. Более хорошим вариантом является вычисление суммы цифр заранее. Для хранения полного пути и суммы цифр можно использовать следующую структуру:

```
struct FileInfo{
    char *full_path; // полный путь
    int sum; // сумма цифр
};
```

Так как общее число файлов заранее неизвестно, то необходимо воспользоваться динамическим массивом. Для этого можно воспользоваться следующей структурой:

```
struct Array{
   int cur_index; // текущее количество элементов и индекс для
записи следующего элемента
   int max_count; // текущий максимальный размер массив
   struct FileInfo *data; // информация о файлах
};
```

Помимо структуры необходимо реализовать функцию для увеличения размера массива, когда количество файлов превышает текущий размер:

```
void check_and_resize(struct Array *arr){
   if(arr->cur_index >= arr->max_count){// достигнут максималь-
ный размер
       arr->max_count += STEP;
       struct FileInfo *tmp = realloc(arr->data, arr->max_count
* sizeof(struct FileInfo));
       if(tmp){ // проверка, что realloc смог выделить память
           arr->data = tmp;
       }else{
           puts("Failed to allocate memory!");
       }
   }
}
```

Так как массив в структуре Array хранит структуры *FileInfo*, то необходимо реализовать функцию, которая на основе имени файла вернет структуру *FileInfo*::

```
struct FileInfo get_file_info(const char *filename, const char
*dir_name){
   struct FileInfo info;
   info.full_path = pathcat(dir_name, filename); // получение
полного пути
   info.sum = 0;
   for(int i = 0; i < strlen(filename); i++){
       if(isdigit(filename[i]))
           info.sum += filename[i] - '0'; // вычисление суммы
цифр
   }
   return info;
}
```

Для реализации функции обхода всех директорий за основу можно использовать функцию *list_dir()* из разд. 3.2.4. Помимо директории для обхода функция будет вторым аргументом принимать указатель на структуру Array:

```
void list_dir(const char *dir_name, struct Array *arr);
```

Важно обратить внимание на то, что передается не сама структура, а указатель на нее. Если передать не указатель, а структуру, то в каждом экземпляре функции будет свой массив. Из-за этого изменения в одном экземпляре функции не будут влиять на другие экземпляры, что приведет к нарушенной логике программы и утечкам памяти. Помимо сигнатуры функции необходимо внести изменения в обработку элементов директории:

```
if(de->d_type == DT_REG){
   arr->data[arr->cur_index++]     =     get_file_info(de->d_name,
dir_name); // запись информации о файле в массив
   check_and_resize(arr); // увеличение размера массива
}else if (de->d_type == DT_DIR && strcmp(de->d_name, ".") != 0
                              && strcmp(de->d_name, "..") != 0){
   char *new_dir = pathcat(dir_name, de->d_name);
   list_dir(new_dir, arr);
   free(new_dir);
}
```

Функция для поиска всех файлов во всех вложенных директориях завершена. Теперь необходимо реализовать функцию *main()*. В функции для сортировки можно воспользоваться функцией стандартной библиотеки – *qsort()*. Для функции *qsort()* необходима функция, которая сравнивает два элемента:

```
int cmp_file_info(const void *a, const void *b){
   struct FileInfo *info_a = (struct FileInfo *)a;
   struct FileInfo *info_b = (struct FileInfo *)b;
   if(info_a->sum > info_b->sum) return 1;
   if(info_a->sum < info_b->sum) return -1;
   return  strcmp(info_a->full_path,  info_b->full_path);  // если
info_a->sum == info_b->sum
}
```

Функция *main()* выглядит следующим образом:

```
int main(){
   struct Array arr; // создание структуры массива
   arr.cur_index = 0;
   arr.max_count = STEP;
   arr.data = malloc(arr.max_count * sizeof(struct FileInfo));
   list_dir(".", &arr);  // обход всех директорий
   qsort(arr.data, arr.cur_index, sizeof(struct FileInfo),
cmp_file_info);  // сортировка найденных путей до файлов
   for(int i = 0; i < arr.cur_index; i++){  // вывод отсортиро-
ванных путей
```

```
        printf("Sum of digits: %d, path: %s\n", arr.data[i].sum,
arr.data[i].full_path);
        free(arr.data[i].full_path);
    }
    free(arr.data);
    return 0;
}
```

3.4. Вопросы для самоконтроля

1. Что такое рекурсия?
2. Что такое глубина рекурсии?
3. Что означают элементы директории *“.”* и *“..”*?
4. Правда ли, что изменение значения переменной в одном экземпляре функции также изменяет значение переменной с таким же названием в другом экземпляре функции? Почему?

Лабораторная работа № 4
ВВЕДЕНИЕ В ЯЗЫК С++

4.1. Цель и задачи

Целью работы является изучение основных механизмов языка С++ путем разработки структур данных стека и очереди на основе динамической памяти.

Для достижения поставленной цели требуется решить следующие задачи:

– ознакомиться со структурами данных стека и очереди, особенностями их реализации;

– изучить и использовать базовые механизмы языка С++, необходимые для реализации стека и очереди;

– реализовать индивидуальный вариант стека в виде С++ класса, его операции в виде функций этого класса, ввод и вывод данных программы.

4.2. Основные теоретические сведения

Введение в классы С++ (инкапсуляция, методы, спецификаторы доступа). Тема классов в С++, как и где их применять, тонкости работы с ними – тема обширная, поэтому рекомендуется обратить внимание на список литературы в конце данного пособия.

Зачастую решение той или иной задачи в программировании сводится к тому, чтобы над неким сложным объектом (в языке Си это можно представить как структуру):

```
struct Point {
   int x;
   int y;
},
```

производились некоторые действия, например, вычисление дистанции между двумя точками (в языке Си это можно представить как функцию):

```
double distance (struct Point *p1, struct Point *p2);
```

Так выглядит процедурная парадигма программирования, которую реализует язык Си. Развитием ее стала объектно ориентированная парадигма, где сложному объекту соответствует набор функций, а программа в данной парадигме – это взаимодействие сложных объектов друг с другом посредством только этих функций. Это означает, что внешний объект должен иметь доступ только к строго определенному набору функций другого объекта, чтобы взаимодействие было предсказуемым, а значит – надежным. Поэтому объектно ориентированный язык программирования должен в обязательном порядке обеспечивать:

– возможность формирования сложных объектов, которые сочетают в себе данные и функции;

– механизм доступа/скрытия к данным и функциям со стороны внешних объектов.

Эти пункты в совокупности называют механизмом инкапсуляции (от лат. in capsula), т. е. и данные, и функции находятся в одной «капсуле».

В C++ такой «капсулой» для данных и функций является класс. Класс – это пользовательский тип данных, удовлетворяющий требованиям инкапсуляции:

– в классе могут размещаться как данные (их называют полями), так и функции (их называют методы) для обработки этих данных;

– любой метод и поле исходного класса имеют свой спецификатор доступа (более подробно о спецификаторах доступа в разд. 9 “Inheritance” [8]). В данной лабораторной работе потребуются:

○ *Public* – доступен для всех, т. е. нет ограничений на взаимодействие с полем (считывание/запись) или методом объекта (вызов);

○ *Private* – доступен только для методов исходного класса.

Теперь можно перейти к рассмотрению классов непосредственно на примере. Класс *Point*, описывающий 2D-точку (аналогично примеру со структурой ранее):

```
class Point {
private:   // далее идут приватные поля/методы класса
   int x;
   int y;
public:   // далее идут публичные поля/методы класса
   double distance (int x, int y);
};
```

Здесь доступ к данным закрыт, доступ к функции открыт. Теперь в программе, допустим в функции *main()*, надо создать объект класса. Если бы Point был Си-структурой, инициализация его объекта могла бы выглядеть так:

```
int main()
{
   Point p1;
   p1.x = 5;
   p1.y = 6;
}
```

Однако *Point* – это класс, и внешнего доступа из функции main к полям класса *x* и *y* нет, так как они отмечены спецификатором доступа private. Поэтому возникла необходимость в специальном методе класса, который будет заниматься инициализацией его начального состояния (в момент создания экземпляра класса).

Такой метод класса называется *конструктор*:

– метод-конструктор всегда носит имя своего класса. (Например, *Point()*);

– у класса всегда есть конструктор по умолчанию (конструктор без аргументов);

– конструкторов может быть много;

– для конструктора не указывается возвращаемое значение.

Далее пример сигнатуры класса *Point* с двумя конструкторами:

```
class Point {
private:   // далее идут приватные поля/методы класса
   int x;
   int y;
public:   // далее идут публичные поля/методы класса
   Point(); // конструктор по умолчанию. Поскольку это сигнатура
-- здесь только объявление метода
   Point(int x, int y); // другой конструктор
   double distance (int x, int y);
};
```

Также в любом классе есть *метод-деструктор*. Деструктор, как следует из его названия, занимается уничтожением экземпляра класса в рамках программы.

Метод-деструктор вызывается, когда переменная-экземпляр класса вышла за пределы области видимости программы. Он также может быть вызван вручную, если требуется освободить память. Может возникнуть вопрос: зачем он потребовался как механизм языка? Ответ можно проиллюстрировать таким примером: класс Train описывает поезд, а среди полей этого класса есть указатели на выделенную в куче память:

```
class Train {

public:
    Train() {  // конструктор по умолчанию
        mWagonsCount = 0;
        mName = (char*)malloc(sizeof(char) * 10);
        strncpy(mName, "Thompson", 8);
    };

    Train(size_t start_count, char* name)   // второстепенный кон-
структор
    {
        if(start_count <= 15)
            mWagonsCount = start_count;
        mName = (char*)malloc(sizeof(char) * strlen(name));
        strncpy(mName, name, strlen(name));
    }

private:
    size_t mWagonsCount;
    char* mName;
};
```

Класс Train имеет поле *mName*, которое содержит указатель на массив символов, и в обоих конструкторах указывает на выделенную в куче память. Что будет, если экземпляр такого класса выйдет из области видимости программы? Будет запущен деструктор, но поскольку по умолчанию это пустой метод, после удаления экземпляра класса в куче останется выделенная память, т. е. произойдет утечка. Чтобы избежать таких утечек был придуман метод-деструктор. Объявляется как конструктор с тильдой:

```
// это деструктор. Освобождается динамический массив mName
~Train() {
    free(mName);
};
```

Теперь при удалении экземпляра класса утечки памяти не будет, так как деструктор с помощью функции free освободит выделенную память.

Осталось научиться работать с экземплярами классов. В классе Train есть конструкторы, деструкторы, но нет бизнес-логики, то есть методов, которые работают с данными класса и придают ему смысл. Пусть класс *Train* должен уметь добавлять вагоны, но при этом не более 15, чтобы сдвинуться с места:

```
void pushWagons(size_t count) {
    if(mWagonsCount + count < 15)
        mWagonsCount+=count;
}
```

А также, воображаемый машинист должен уметь доложить сколько вагонов в поезде тому, кто спросит:

```
size_t wagonsCount() {
    return mWagonsCount;
}
```

Данные методы должны быть доступны извне класса, поэтому они прописываются под спецификатором доступа *public*. Таким образом, полная реализация класса:

```
class Train {

public:
    Train() {  // конструктор по умолчанию
        mWagonsCount = 0;
        mName = (char*)malloc(sizeof(char) * 10);
        strncpy(mName, "Thompson", 8);
    };


    Train(size_t start_count, char* name)   // второстепенный кон-
структор
    {
        if(start_count <= 15)
            mWagonsCount = start_count;
        mName = (char*)malloc(sizeof(char) * strlen(name));
        strncpy(mName, name, strlen(name));
    }
```

```
    void pushWagons(size_t count) {
        if(mWagonsCount + count < 15)
            mWagonsCount+=count;
    }

    size_t wagonsCount() {
        return mWagonsCount;
    }

    ~Train() {
        free(mName);
    };

private:
    size_t mWagonsCount;
    char* mName;
};
```

Далее представлен пример, как создавать экземпляры класса и взаимодействовать с ними:

```
int main()
{
    {
        Train loko1;    // Вызывается конструктор по умолчанию

        char name[] = "CrazyTrain";
        Train loko2(5, name);

      //loko1.mWagonsCount = 8; // Ошибка компиляции, так как поле приватное.
        loko1.pushWagons(7); // прибавит 7 вагонов к поезду loko1

        size_t count = loko2.wagonsCount();

    } // Вызов деструкторов для loko1 и loko2

    return 0;
}
```

Пользователь не может напрямую изменить количество вагонов в конкретном поезде, но у него для этого есть метод *pushWagons()*. Однако с помощью этого метода не получится сделать вагонов больше 15, поскольку программист позаботился о том, чтобы поезд мог сдвинуться с места и никто не прикрепил к нему лишних вагонов.

Перегрузка функций (методов). Прежде, чем перейти к тому, что такое перегрузка функций, стоит повторно рассмотреть сигнатуру класса *Point* из предыдущего раздела:

```
class Point {
private:
   int x;
   int y;
public:
   Point();
   Point(int x, int y);
   double distance (int x, int y);
};
```

Point() и *Point(int x, int y)* – это конструкторы для класса Point. *Конструктор* – это метод класса, а метод класса – это функция. Получается, что в одной программе существует две функции с одним названием. В логике языка Си такого не может быть, но в C++ это возможно благодаря перегрузке функций.

Перегрузка функций в C++ позволяет определять несколько функций с одинаковым названием при условии, что их аргументы отличаются. Это делает перегрузку удобным инструментом, если требуется решить с помощью функции задачу в разном контексте. В примере выше:

– задача метода-конструктора Point – инициализировать состояние полей экземпляра класса;

– контекст – входные данные, поскольку они отличаются, код инициализации также будет отличаться.

Таким образом, функция Point – единственная, но её логика будет варьироваться в зависимости от того, какие данные были поданы.

Какие различия в аргументах делают функции в C++ перегруженными:

– количество аргументов;

– типы аргументов.

Что не является перегрузкой? Например, разные названия аргументов:

```
Point(int a, int b);
Point(int x, int y);
```

Они не создают перегрузку, такой код не будет скомпилирован. Также нельзя перегрузить функции, отличающиеся только по типу возвращаемого значения:

```
int func(int a) {...}
double func(int a) {...}
```

Такой код также не будет скомпилирован – этого вида информации недостаточно для C++, чтобы во всех случаях компилятор мог решить, какую именно функцию нужно вызвать.

При вызове функции из списка перегруженных, в первую очередь выбирается та, для которой нашлось точное совпадение, например:

```
void print(int val);
void print(char* str);

print(0); // точное совпадение с print(int)
```

Чтобы лучше понять выигрыш от перегрузки функций, полезно рассмотреть еще один пример – три функции из стандартной библиотеки: *abs()*, *labs()* и *fabs()*. Они были впервые определены в языке Си, а затем включены в C++. *abs()* возвращает абсолютное значение (модуль) целого числа, *labs()* возвращает модуль длинного целочисленного значения (типа *long*), а *fabs()* – модуль значения с плавающей точкой (типа *double*).

Поскольку язык Си не поддерживает перегрузку функций, каждая функция должна иметь собственное имя, несмотря на то, что все три функции выполняют, по сути, одно и то же действие. Таким образом, при одних и тех же действиях программисту необходимо помнить имена всех трех (в данном случае) функций вместо одного. Но в C++ можно использовать только одно имя для всех трех функций (для примера – *myabs*):

```
int myabs(int i);
double myabs(double d);
long myabs(long l);

int myabs(int i) // перегрузка для int
{
   if(i<0) return -i;
   else return i;
}

double myabs(double d) // перегрузка для double
{
   if(d<0.0) return -d;
   else return d;
}

long myabs(long l) // перегрузка для long
```

```
{
   if(l<0) return -l;
   else return l;
}

int main() { … }
```

Важно помнить, что этот механизм имеет ограничения, и что делать, если потребовалось написать две функции с одинаковой сигнатурой, но разной логикой? Для этого в C++ есть свой механизм – пространства имен. Они широко применяются в C++, начиная с примеров уровня *“Hello World”*:

```
#include <iostream>

int main() {
   std::cout << "Hello World!";
   return 0;
}
```

Обратите внимание на конструкцию *std::cout*. На самом деле это класс cout из пространства имен std, где “::” – оператор доступа к функции (по аналогии с операторами “.” и “->”). Конструкция “<<” – это оператор (функция), который реализует вывод в поток вывода (более подробно об операторах в разд. 8 “Operator Overloading” [8]).

Пространства имен сделаны для того, чтобы избежать двойного объявления функций с одинаковыми сигнатурами, однако, если вы уверены, что функций с одинаковыми названиями в вашей программе нет, можно воспользоваться ключевым словом *using*, для того, чтобы не указывать пространства имен перед вызываемой функцией:

```
#include <iostream>

using namespace std;
int main() {
   cout << "Hello World!";
   return 0;
}
```

Динамическая память в C++: операторы new, delete. В языке Си память можно выделять с помощью функций библиотеки *stdlib*: *malloc*, *calloc*, *realloc*. Выделенную память освобождают с помощью функции free. Язык C++ предоставляет альтернативный способ: оператор new обеспечивает выделение динамической памяти в куче, а для освобождения выделенной памяти используется оператор *delete*. Пример использования *new* и *delete*:

```
#include<iostream>

using namespace std;
int main(){
   int* number = new int; // выделение памяти
   *number = 1;
   cout << *number; // 1
   delete number; // освобождение памяти
   return 0;
}
```

Создать динамический массив в C++ проще, чем в Си. Нужно сообщить оператору *new* тип элементов массива и требуемое количество элементов. Например, если необходим массив из 5 элементов *int*, следует записать так:

```
int * something = new int [5] ; // получение блока памяти из 5 элементов типа int
```

Оператор *new* возвращает адрес первого элемента в блоке. В данном примере это значение присваивается указателю *something*. Теперь, чтобы освободить память, выделенную ранее, будет недостаточно написать:

```
delete something;
```

Потому что *delete* – это функция (оператор, подробнее в разд. 10 “Memory Management” [8]) стандартной библиотеки C++, которая удалит только 1 элемент по указателю *something*. Если требуется удалить массив данных, нужно использовать функцию *delete[]*. Обратите внимание, что это именно две разные функции:

```
delete[] something;
```

Для успешной работы с *new* и *delete* надо помнить несколько простых правил:

– не перемешивать их использование с функциями стандартной библиотеки языка Си, например использовать *free* вместо *delete*;

– не использовать *delete* для освобождения одного и того же блока памяти дважды;

– использовать *delete[]*, если применялась операция *new[]* для размещения массива;

– использовать *delete* без скобок, если применялась операция *new* для размещения отдельного элемента;

– помнить о том, что применение *delete* к нулевому указателю является безопасным (при этом ничего не происходит).

Шаблоны (Templates). В программировании часто бывает ситуация, когда нужно обобщить группу похожих объектов или алгоритмов, которые от-

личаются друг от друга, например, типом входных данных, полей класса или размером какого-то массива, буфера.

Например, потребовалось написать функцию слияния двух целочисленных массивов (второй массив дописывается в конец первого) без удаления исходных:

```
// передаются адреса указателей на массивы и их размеры
void mergeArrays(int** arr1, int** arr2, size_t s1, size_t s2)
{
   int* tmp = new int[s1+s2];
   memcpy(tmp, *arr1, s1*sizeof (int));
   memcpy(tmp+s1, *arr2, s2*sizeof (int));
   *arr1 = tmp;
}
```

Теперь потребовалось объединить массивы типа *double*. Воспользовавшись перегрузкой, можно решить эту задачу,:

```
void mergeArrays(double** arr1, double** arr2, size_t s1, size_t
s2)
{
   double* tmp = new double[s1+s2];
   memcpy(tmp, *arr1, s1*sizeof (double));
   memcpy(tmp+s1, *arr2, s2*sizeof (double));
   *arr1 = tmp;
}
```

Если сравнить код этих функций, то он ничем не отличается кроме типа данных массивов. Но что, если потребуется такая функция для *N* классов? Для каждого типа будет функция дубликат, что приведет к множественному повторению кода, и в C++ реализован механизм для решения этой проблемы.

Для того, чтобы избавиться от дублирования, в C++ ввели шаблоны. Шаблоны – это инструмент, позволяющий компактно описывать обобщённые алгоритмы, объекты за счет устранения привязки к типам данных, размерам массивов [10].

Описание шаблона выглядит следующим образом:

```
template <"список параметров шаблона">
"сигнатура функции"

template <typename T> // далее вместо имени класса используется
имя шаблона "T"
void mergeArrays(T** arr1, T** arr2, size_t s1, size_t s2)
{
```

```
    T* tmp = new T[s1+s2];
    memcpy(tmp, *arr1, s1*sizeof (T));
    memcpy(tmp+s1, *arr2, s2*sizeof (T));
    *arr1 = tmp;
}
```

Далее функция *mergeArrays* переписывается с помощью шаблонов заменой тип данных на произвольный тип *T*. Чтобы ввести произвольный тип, используется слово typename:

Эту функцию можно использовать для слияния массивов любого типа данных. Для каждого типа компилятор сам создаст отдельный экземпляр функции с нужным типом данных. Пример вызова этой шаблонной функции:

```
// Инициализация исходных массивов
int* a = new int[4];
int* b = new int[6];

for(int i=0; i<4; i++)
a[i] = 1;
for(int i=5; i>=0; i--)
b[i] = 4;

// Вызов шаблонной функции, где в кавычках явно указывается тип
аргумента(ов), так как массив int-ов,
//то ставим в кавычки "int"

mergeArrays<int>(&a, &b, 4, 6);
```

В качестве параметра шаблона можно ввести не только тип данных, но и объект. Например, в стандартном контейнере C++ из библиотеки STL (подробнее о библиотеке STL в разделе 15 “The standart template library”) подается на вход размер массива:

```
#include <iostream>
#include <array>

int main()
{
    std::array<int, 3> a1 = {1, 2, 3};
    return 0;
}
```

В примере выше второй аргумент шаблона это число 3, оно обозначает размер создаваемого массива.

Обработка исключений. Механизм обработки исключений позволяет программисту ограничить работу программы так, чтобы избежать непредсказуемого поведения при ее неверном использовании. Примеры ситуаций, когда данный механизм может быть полезен:

1. Обеспечение «безопасного периметра»: при вводе неверных входных данных программа может вести себя непредсказуемо.

2. При работе с файлами или другими внешними ресурсами, например, отсутствие файла по заданному пути.

3. При динамическом выделении памяти с помощью оператора *new* может произойти ситуация, когда доступная память в компьютере исчерпана.

4. При выполнении сложных алгоритмов могут возникать ситуации, деления на ноль, выход за границы массива и т. п.

Правильно выстроенная обработка исключительных ситуаций делает программу:

– *устойчивой* – ошибки возникающие в программе не приводят к внезапному прекращению её работы;

– *предсказуемой* – попытки использовать программу не по назначению или неправильно блокируются;

– *удобной в отладке* – программист, использующий вашу программу, будет получать exception (объект исключения) и описание ошибки, если использует ее неверно.

Важно понимать, что данный механизм – исключительно программный, и ОС с ним никак не взаимодействует.

Здесь необходимо небольшое отступление о том, что такое наследование. На данный момент достаточно понимания сути данного механизма. Наследование – это расширение функциональности класса с помощью создания классов-потомков. В новый класс можно добавлять новые методы, поля, при этом методы и поля исходного класса в классе-потомке останутся.

Теперь о том, как механизм обработки исключений реализован в C++ и как его использовать. *Исключение* – это переменная, тип которой с точки зрения языка может быть любым, однако в реальной практике чаще используется класс std::exception и его классы-наследники. Система генерации и обработки реализована с помощью трех ключевых слов:

– *try* – блок кода, где будет отлавливаться исключение;

– *catch* – если исключение было поймано во время выполнения кода в *try{}*, выполняется код из этого блока;

– *throw* – команда выброса исключения.

К примерам – метод *pushWagons* класса *Train*:

```
void pushWagons(size_t count) {
    if(mWagonsCount + count < 15)
        mWagonsCount+=count;
}
```

Что будет, если данным классом воспользуется другой программист? Например:

```
int main()
{
    char name[] = "Saint-Petersburg Express";
    Train loko(5, name);


    loko.pushWagons(20);

    size_t count = loko2.wagonsCount();
    printf("Wagons: %d\n", count);

    return 0;
}
```

В консоль будет выведено 5, и программист не получит ожидаемого поведения программы. Нужно сделать так, чтобы программа не выполнялась далее при попытке добавления неверного значения или меняла свое поведение. Для этого в функцию *pushWagons* добавляется команда *throw* – бросание исключения:

```
void pushWagons(size_t count) {
    if(mWagonsCount + count < 15)
        mWagonsCount+=count;
    else
        throw "some problem happened!";
}
```

Если запустить программу при такой конфигурации, произойдет аварийная остановка программы, и в консоль будет выведено:

terminate called after throwing an instance of 'char const'*

Аварийный останов (стек памяти сброшен на диск)

Как можно видеть, данное сообщение не позволяет понять, в чем конкретно ошибка. Поэтому на практике бросают класс-исключение, в C++ это классы-наследники от *std::exception* (объявлен в заголовочном файле *stdexcept* вместе с остальными классами исключений из стандартной библиотеки):

```
void pushWagons(size_t count) {
```

```
    if(mWagonsCount + count < 15)
        mWagonsCount+=count;
    else
        throw std::range_error("Too many train vagons!");
}
```

Запуск:

terminate called after throwing an instance of 'std::range_error'
what(): Too many train vagons!

Класс-исключение *std::range_error* реализован в стандартной библиотеке и по названию означает ошибку выхода за границы допустимого диапазона значений, о чем и сообщено в терминале. Тем не менее, программа все еще выводит в консоль ошибку, поэтому нужно прописать сценарий работы программы в этом случае.

Делается это с помощью отлова исключений там, где они могут возникнуть с помощью *try/catch* блоков. Пусть *MAX_COUNT* – константа, которая задается в классе Train как максимум вагонов:

```
#define MAX_COUNT 15

int main()
{
    char name[] = "Saint-Petersburg Express";
    Train loko(5, name);

    try {
        loko.pushWagons(20); // бросается исключение
    } catch (std::exception) { // обработка исключения
        loko.pushWagons(MAX_COUNT-loko.wagonsCount()-1);
    }

    size_t count = loko.wagonsCount();
    printf("Wagons: %ld\n", count);

    return 0;
}
```

Обратите внимание, что в качестве исключения можно бросить объект любого типа, что значительно удобнее, чем механизм с глобальной переменной errno в языке Си, которая может быть только типа *INT* и имеет все недостатки глобальной переменной.

На вход блоку *catch* подается тип отлавливаемого исключения, и поскольку *std::range_error* является классом-наследником *std::exception*, ис-

ключение отлавливается. Если подходящего обработчика не найдено, ошибка поднимается на уровень выше по стеку, т. е. выше по иерархии вызовов, до тех пор, пока не встретит:

1. *try-catch* блок с подходящим обработчиком. Тогда исключение обрабатывается найденным обработчиком.

2. Функцию *main*. В этом случае процесс получает сигнал *SIGTERM*, который приводит к остановке процесса.

4.2.3.Динамические структуры данных

Данная тема рассматривалась в разд. 2, и тогда речь шла о связных списках и о том, как реализовать их на языке Си. В данном разделе речь пойдет о структурах стек и очередь, они как и списки часто встречаются в практике программирования. Например, ранее упоминался *call stack*, или стек вызовов, который хранит информацию, нужную для возврата управления основной процедуре из подпроцедур.

Стек. Для того чтобы понять суть стека, можно представить себе такую аналогию: в раковине находится стопка тарелок, при этом размер тарелок почти совпадает с размером раковины. Что можно в этой ситуации сделать? Можно взять верхнюю тарелку, положить тарелку сверху, посмотреть, есть ли тарелки в раковине, посмотреть характеристики тарелки, лежащей сверху. Поскольку из-за размеров тарелок нет возможности что-то делать с ними сбоку, например, взять пачку тарелок или взять тарелку из середины, на этом все возможности заканчиваются.

Отсюда можно обобщить понятие стека. Стек – это структура данных, в которой добавление новых элементов и удаление существующих производится с одного конца – вершины стека. Таким образом, первым из стека удаляется элемент, который был помещен туда последним, т. е. в стеке реализуется стратегия «последним вошел – первым вышел», или **LIFO** (Last In First Out). Как происходит добавление и удаление элементов показано на рис. 4.1.

Какой набор функций (для языка Си) или методов (для языка C++) может иметь стек:

push – операция вставки нового элемента;

pop – операция удаления элемента с вершины стека;

empty – проверка стека на наличие в нем элементов;

size – подсчет количества элементов;

top – операция просмотра верхнего элемента.

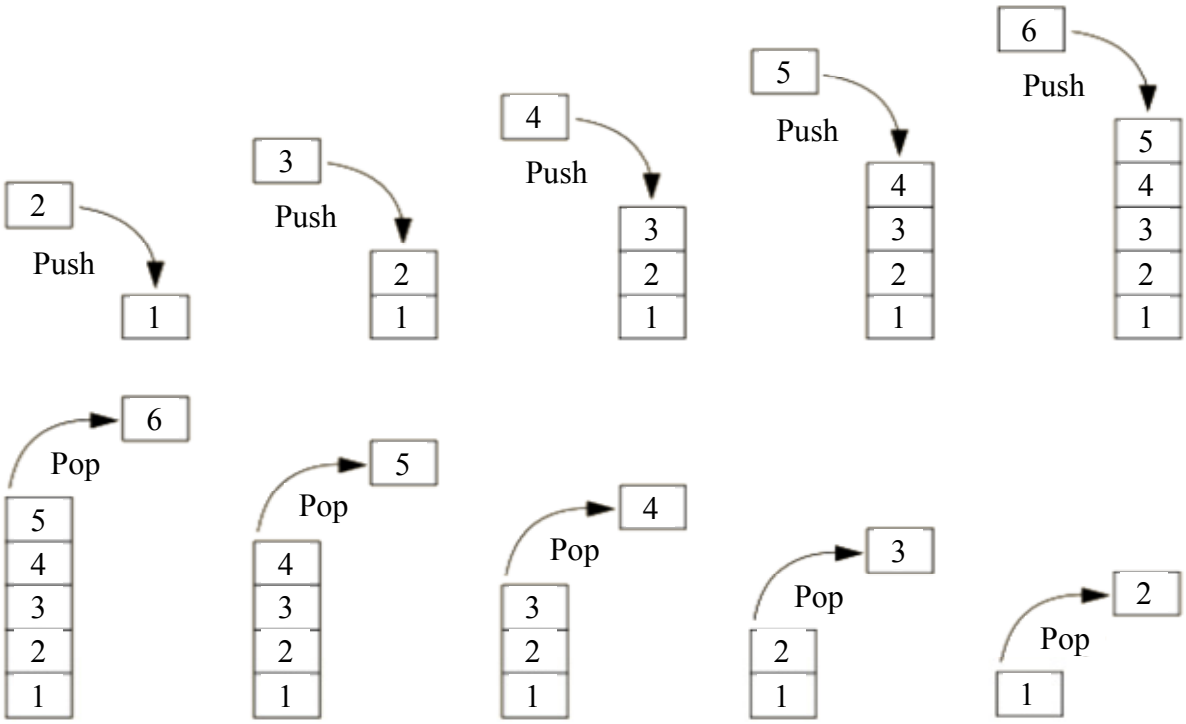

Рис. 4.1. Графическое представление операций push и pop для стека

Данный набор процедур не является исчерпывающим: есть ещё дополнительные процедуры работы со стеком, но они не являются для этой структуры данных определяющими, а нужны для упрощения работы с ней в некоторых ситуациях. О них будет рассказано далее, в разд. 4.3.

Очередь. Аналогия для очереди напрашивается сама собой – очередь к кассе в магазине. Все покупатели в очереди ценят свое место и не позволяют влезать в середину очереди. Таким образом, в очереди реализуется стратегия «первым вошел – первым вышел» или **FIFO** (First In First Out). Как происходит добавление и удаление элементов показано на рис. 4.2.

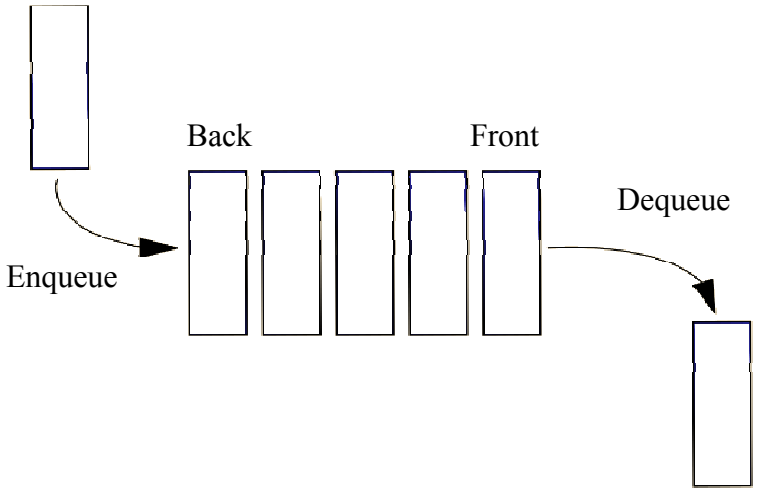

Рис. 4.2. Графическое представление операций push и pop для очереди

Базовые операции для работы с очередью аналогичные стеку:

enqueue – операция вставки нового элемента;

dequeue – операция извлечения элемента;

empty – проверка стека на наличие в нем элементов;

size – подсчет количества элементов;

top – операция просмотра нового элемента.

Так как для названий функций изменения очереди в английском языке есть специальные глаголы enqueue и dequeue, обычно используют их, но по сути это функции push и pop соответственно.

4.3. Задание к лабораторной работе № 4

Требуется написать программу, моделирующую работу стека на базе списка или на базе массива в зависимости от варианта. Для этого необходимо:

1. Реализовать класс *CustomStack*, который будет содержать перечисленные ниже методы. Стек должен иметь возможность хранить данные типа char и работать с ними.

2. Обеспечить обработку возможных ошибок при работе программы, например, вызов метода pop или top при пустом стеке (для операции в стеке не хватает аргументов). Программа должна вывести "error" и завершиться.

Описание структур классов на базе списка:

```
// Структура класса узла списка
struct ListNode {
   ListNode* mNext;
   char mData;
};

// структура класса стека
class CustomStack {
public:
// методы push, pop, size, empty, top + конструкторы, деструктор
private:
// поля класса, к которым не должно быть доступа извне
protected:  // в этом блоке должен быть указатель на голову
   ListNode* mHead;
};
```

Описание структуры класса стека на базе массива:

```
class CustomStack {
public:
// методы push, pop, size, empty, top + конструкторы, деструктор
```

```
private:
// поля класса, к которым не должно быть доступа извне
protected:  // в этом блоке должен быть указатель на массив дан-
ных
    char* mData;
};
```

Перечень методов класса стека, которые должны быть реализованы:

void push(char val) – добавляет новый элемент в стек;

void pop() – удаляет из стека последний элемент;

char top() – доступ к верхнему элементу;

size_t size() – возвращает количество элементов в стеке;

bool empty() – проверяет отсутствие элементов в стеке;

extend(int n) – расширяет исходный массив на *n* ячеек (только для стека на базе массива);

4.3.1. Описание последовательности выполнения работы

Реализация стека на базе массива:

```
using namespace std;

class CustomStack {
public:
    CustomStack(size_t initialCapacity) {
        this->mCapacity = initialCapacity;
        this->mData = new int[initialCapacity];
        this->mIndex = -1;
    }

    CustomStack() : CustomStack(10) { // 10 -- начальный размер
стека, вызов другого конструктора
    }

    ~CustomStack() {
        delete[] this->mData;
    }

    void push(int val) {
        this->ensureSpace(); // проверка, что размера массива до-
статочно для нового элемента
        this->mIndex++;
        this->mData[this->mIndex] = val;
```

```
    }

    void pop() {
        if (this->empty()) {
            throw logic_error("pop() called on empty stack");
        }
        this->mIndex--; // "удаление" элемента
    }

    int top() {
        if (this->empty()) {
            throw logic_error("top() called on empty stack");
        }
        return this->mData[this->mIndex];
    }

    size_t size() const {
        return this->mIndex + 1;
    }

    bool empty() const {
        return this->mIndex == -1;
    }

    void extend(int n) {
        if (n <= 0) {
            throw logic_error("extend() called with a non-
positive argument");
        }
        this->resize(this->mCapacity + n);
    }

protected:
    size_t mCapacity;
    size_t mIndex;
    int *mData;

    size_t getNewCapacity() const {
        // получение нового размера
        return this->mCapacity * 3 / 2 + 1;
    }

    void ensureSpace() {
```

```
        if (this->mIndex + 1 == mCapacity) {
            // если достигнут максимальный размер
            size_t newCapacity = this->getNewCapacity();
            this->resize(newCapacity);
        }
    }

    void resize(size_t newCapacity) {
        if (newCapacity == mCapacity) {
            return;
        }
        if (newCapacity < mCapacity) {
            throw logic_error("resize() called with a lower ca-
pacity");
        }
        int *newData = new int[newCapacity];
        copy(this->mData, this->mData + this->mCapacity, newDa-
ta); // копирование данных при помощи функции из заголовочного
файла <algorithm>
        delete[] this->mData;
        this->mData = newData;
        this->mCapacity = newCapacity;
    }
};
```

Реализация на базе списка:

```
struct ListNode { // узел списка
    ListNode* mNext;
    char* mData;
};

class CustomStack
{
public:
    CustomStack()
    {
        mHead = nullptr;
        mSize = 0;
    }

    bool empty()
    {
```

```
        return mHead == nullptr;
    }

    size_t size()
    {
        return mSize;
    }

    char *top()
    {
        if (empty())
            return nullptr;
        return mHead->mData;
    }

    void push(const char *str)
    {
        // создание нового узла
        ListNode *newElement = new ListNode;
        newElement->mData = new char[strlen(str)];
        strcpy(newElement->mData, str);
        newElement->mNext = mHead; // новый узел указывает на
предыдущую голову
        mHead = newElement; // новый узел становится головой
        mSize++;
    }

    void pop()
    {
        if (empty())
            return;
        delete[] mHead->mData; // освобождение памяти с данными
        ListNode *deletingElement = mHead->mNext; // сохранение
указателя на предыдущую голову
        mHead = mHead->mNext; // обновление головы
        delete deletingElement; // освобождение предыдущей головы
        mSize--;
    }

    ~CustomStack()
    {
        // удаление элементов, пока они есть
        while (!empty())
```

```
            pop();
    }
private:
    size_t mSize;
protected:
    ListNode *mHead;
};
```

4.3.2. Пример выполнения задания на защиту

Требуется написать программу, которая при помощи стека на базе списка будет проверять, является ли переданная в нее строка правильной скобочной последовательностью. Последовательность может состоять только из символов “(”, “)”.

Алгоритм реализации задачи состоит в переборе символов строки и работе с ними. Если встретившийся символ равен “(”, то кладем его в стек. Если символ равен “)”, извлекаем верхний элемент из стека и сравниваем его с текущим символом. Если извлеченный из списка элемент также имеет значение “)”, то последовательность неправильная. Иначе, если предыдущий элемент в списке имеет значение “(”, проверку последовательности можно продолжить.

```
int main() {
    CustomStack p = CustomStack();
    std::string s("");
    std::cin>>s;
    // пустая строка является правильной скобочной последователь-
ностью
    if (s.length() == 0){
        std::cout<<"True"<<std::endl;
        return 0;
    }
    // перебираем символы из входной строки
    for (int i = 0; i < s.length(); i++){
        if (s[i] == '('){
            p.push(s[i]);
        } else {
            char elem;
            if (!p.empty()){
                elem = p.top();
                p.pop();
            } else {
                std::cout<<"False"<<std::endl;
```

```
                return 0;
            }
            if (elem == s[i]){
                std::cout<<"False"<<std::endl;
                return 0;
            }
        }
    }
    if (p.empty()){
        std::cout<<"True"<<std::endl;
    } else { // последовательность неправильная, в стеке остались
элементы
        std::cout<<"False"<<std::endl;
    }
    return 0;
}
```

4.4. Вопросы для самоконтроля

1. Как выделить и освободить память в C++?
2. К переменной с каким спецификатором доступа можно обратиться только внутри класса в C++?
3. Чем стек отличается от очереди?
4. Что такое шаблон в C++?

ПРИЛОЖЕНИЯ

1. ПОДРОБНЕЕ О РЕКУРСИИ

1.1. Числа Фибоначчи

Числами Фибоначчи называют последовательность чисел, где каждое следующее является суммой двух предыдущих. Первыми двумя числами последовательности являются ноль и единица. Начало последовательности чисел Фибоначчи: *0,1,1,2,3,5,8,13*. Математически формула для чисел Фибоначчи выглядит следующим образом: $F_n = F_{n-1} + F_{n-2}$, где $F_0 = 0$, $F_1 = 1$. Аналогично формуле факториала в данной формуле в обеих частях уравнения присутствуют числа Фибоначчи, что указывает на возможность рекурсивного решения. Числа F_0 и F_1 (т. е. значения n, равные 0 или 1) являются условиями выхода. Далее приведена функция для вычисления n-ого числа Фибоначчи:

```
#include <stdio.h>

int fib(int n){ // рекурсивная функция
   if(n == 0) return 0; // условие выхода F_0
   else if(n == 1) return 1; // условие выхода F_1
   return fib(n - 1) + fib(n - 2); // два рекурсивных вызова
}

int main()
{
   printf("Десятое число Фибоначчи: %d\n", fib(10));
   return 0;
}
```

1.2. Виды рекурсии

Рекурсию разделяют на два вида: прямую и косвенную. Рекурсию называют прямой, когда функция напрямую вызывает себя. Примеры, разобранные ранее, используют **прямую** рекурсию. Это видно по строкам с return, где напрямую происходит рекурсивный вызов.

Рекурсию называют косвенной, когда функция напрямую себя не вызывает, однако ее вызов происходит в другой функции, которая была вызвана исходной. Для примера далее будет рассмотрена рекурсивная программы вывода чисел от 1 до 10 с пометкой о четности:

```
#include <stdio.h>
```

```
void print_odd(int n);

void print_even(int n){
   printf("%d (число чётное)\n", n);
   if(n > 0) print_odd(n - 1);
}

void print_odd(int n){
   printf("%d (число нечётное)\n", n);
   if(n > 0) print_even(n - 1);
}

int main()
{
   print_even(6);
   return 0;
}
```

Функция *print_even* печатает число с пометкой о том, что оно чётное. Затем происходит вызов функции *print_odd*, которая также печатает число, но с пометкой о том, что оно нечётное. После этого происходит вызов функции *print_even*. Таким образом, во время выполнения функции *print_even* была вызвана она же сама, однако через функцию *print_odd*, поэтому это не прямая рекурсия, а **косвенная**. На схеме ниже изображено графическое представление косвенной рекурсии.

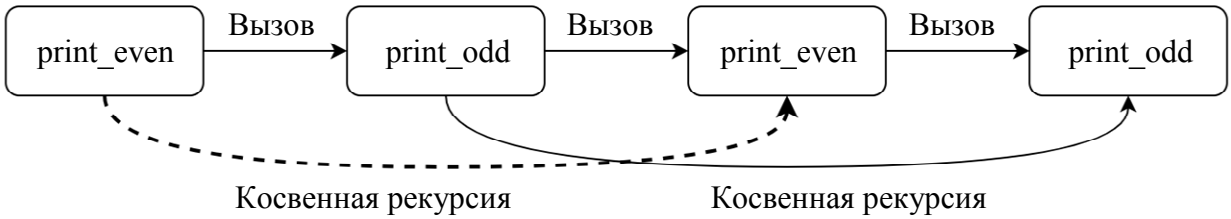

Рис. П.1. Схема косвенной рекурсии

Важно обратить внимание на то, что перед функцией *print_even* определена сигнатура (прототип) функции *print_odd,* так как функция *print_even* при выполнении вызывает функцию *print_odd*, которая в коде до этого не появлялась. Если убрать определение сигнатуры, то при компиляции возникнет ошибка того, что функция *print_odd* не определена.

Также выделяют **хвостовую рекурсию**, в которой рекурсивный вызов происходит последним выполняемым выражением, т. е. после рекурсивного вызова функция завершает свое выполнение.

Только пример программы с вычислением НОД из раздела 3.2.2 является хвостовой рекурсией. Может показаться, что функция для вычисления факториала является примером хвостовой рекурсии, однако это не так, потому что последним выполняемым выражением является умножение. Для того чтобы функция факториала являлась хвостовой рекурсией, необходимо выполнять умножение не перед оператором return. Для этого необходимо добавить дополнительный аргумент у функции, который будет накапливать результат, который вернется при достижении условия выхода. Далее приведен код для вычисления факториала с использованием хвостовой рекурсии:

```
#include <stdio.h>

int factorial_tail_recursion(int n, int result){ // рекурсивная
функция
   if(n == 0) return result; // условие выхода
   return factorial(n - 1, n * result); // рекурсивный вызов
}

int main()
{
   printf("Факториал 10 равняется %d\n",
          factorial_tail_recursion(10, 1));
   return 0;
}
```

1.3. Дополнительные функции работы с файлами

Функции и структуры для работы с файлами определены в заголовочном файле *stdio.h*. Для работы с файлами используется файловый поток, реализованный структурой *FILE*. Напрямую работа с данной структурой не производится, поэтому содержимое структуры рассмотрено не будет. Структура *FILE* используется в качестве аргумента для функций работы с файлами. Перед началом работы с файлом его необходимо открыть. Для этого необходимо использовать функцию *fopen*:

```
FILE *fopen(const char *filename, const char *opentype);
```

Данная функция открывает файл по пути *filename* в режиме opentype. Путь *filename* может быть абсолютный (например, *"/home/user/file.txt"*) или относительный (например *"../file.txt"*). Аргумент opentype является строкой, которая определяет режим файлового потока для открываемого файла. Режимы открытия файловых потоков описаны в таблице ниже. Обязательно должно начинаться с одного из последующих символов:

Таблица П.1

Режимы открытия файлов

Режим открытия файлового потока	Описание
"r" (сокращение от read)	Файловый поток будет доступен только на чтение
"w" (сокращение от write)	Файловый поток будет доступен только на запись. Если файл уже существует, то его содержимое удаляется, иначе создается новый файл
"a" (сокращение от append)	Файловый поток будет доступен только на запись, однако в отличии от режима *"w"*, если файл существует, то его содержимое остается неизменным, а данные на запись записываются в конец файла
"r+"	Файловый поток будет доступен как на чтение, так и на запись. Если файл существует, то его содержимое остается неизменным, а текущая позиция в файле указывает на начало файла
"w+"	Файловый поток будет доступен как на чтение, так и на запись. Если файл существует, то его содержимое удаляется, иначе создается новый файл
"a+"	Аналогично *"r+"*, однако если файл существует, то его содержимое остается неизменным, а текущая позиция в файле указывает на конец файла
Дополнительные системно-зависимые режимы	Символы относящиеся к дополнительному режиму идут после символов основного режима. Одним из распространенных является режим "b", который означает, что работа с файлом будет производится как с бинарными данными, а не текстовыми. Например, режим "wb" откроет файл на запись в режиме бинарных данных

Функция возвращает указатель на файловый поток *FILE*, отвечающий за работу с файлом. Если не удалось открыть файл, то функция возвращает *NULL*. После завершения работы с файлом его необходимо закрыть при помощи функции *fclose()*:

```
int fclose(FILE *stream);
```

Данная функция принимает указатель на файловый поток, полученный при открытии файла. Функция возвращает 0 при успешном закрытии файла и константу *EOF* при возникновении ошибки. Также существует функция *fcloseall()*, которая закрывает все файлы, связанные с данной программой (процессом):

```
int fcloseall(void);
```

Данная функция не принимает аргументов и возвращает 0 при успешном закрытии всех файлов и константу *EOF* при возникновении ошибки.

Работа с файлами может осуществляться так же, как и со стандартными потоками ввода и вывода. Для этого существуют функции: ***fprintf***, ***fscanf***, ***fgetc***, ***fgets***, ***fputc***, ***fputs***, – они работают аналогично своим двойникам для стандартных потоков ввода и вывода, однако принимают дополнительный аргумент в виде указателя на файловый поток *FILE*.

Помимо описанных ранее функций существуют дополнительные функции для работы с файловым потоком. Осуществлять чтение и запись можно при помощи функций ***fread*** и ***fwrite***:

```
size_t fread(void *ptr, size_t size, size_t count, FILE
*stream);
size_t fwrite(const void *ptr, size_t size, size_t count, FILE
*stream);
```

Данные функции осуществляют работу с блоками бинарных данных. Функции принимают одинаковые аргументы:

• *(const) void *ptr* – указатель на массив данных, в который будет произведено чтение (*fread*) или из которого будет произведена запись (*fwrite*) данных;

• *size_t size* – размер одного элемента массива в байтах;

• *size_t count* – количество элементов, которое необходимо записать/прочитать;

• *FILE *stream* – указатель на файловый поток, для которого необходимо выполнить данную функцию.

Функция возвращает количество успешно прочитанных или записанных элементов. Далее приведен пример кода для записи 10 чисел типа int в файл *test.txt:*

```
void write_to_file(){
   FILE *file = fopen("test.txt", "wb"); // открытие файла на
запись бинарных данных
   if(file){ // если файл успешно открыт
        int arr[10];
        for(int i = 0; i < 10; i++)
            arr[i] = i + 1;
        int written = fwrite(arr, sizeof(int), 10, file); // за-
пись данных в файл
        printf("Успешно записано %d элементов\n", written);
        fclose(file); // закрытие файла
   }else
        puts("Не удалось открыть файл test.txt");
}
```

При успешном выполнении вывод будет следующим:

Успешно записано 10 элементов.

Аналогичным образом при помощи функцию *fread* можно прочитать данные из файла. Далее представлен пример кода, который считывает 5 чисел типа *int* (при этом выводится массив из 10 чисел) из файла *test.txt*:

```
void read_from_file(){
   FILE *file = fopen("test.txt", "rb"); // открытие файла на чтение бинарных данных
   if(file){ // если файл успешно открыт
        int arr[10] = {0};
        int readed = fread(arr, sizeof(int), 5, file);
        printf("Успешно прочитано %d элементов\n", readed); // чтение данных из файла
        for(int i = 0; i < 10; i++)
            printf("%d ", arr[i]);
        fclose(file); // закрытие файла
   }else
        puts("Не удалось открыть файл test.txt");
}
```

При успешном выполнении вывод будет следующим:

Успешно прочитано 5 элементов

1 2 3 4 5 0 0 0 0 0

Важно помнить, что функции *fread* и *fwrite* работают с бинарными данными, которые плохо читаются человеком. Например, если записать число в файл при помощи *fwrite*, то при открытии файла в текстовом редакторе не будет отображено записанное число. Вместо этого будет выведена символьная интерпретация байтов, которые составляют записанное число. Однако у этого есть положительная сторона: данные занимают меньше места. Например, число 37 000 можно уместить в 16 бит. Таким образом, при записи числа как бинарных данных размер файла будет составлять 2 байта (без учета дополнительной системной информации). Если же выполнить запись числа в символы, то размер файла составит минимум 5 байтов (в зависимости от кодировки размер одного символа может варьироваться).

Также можно взаимодействовать не только с содержимым файлов, но и выполнять некоторые операции над самим файлом. При помощи функции remove можно удалить файл:

```
int remove(const char *filename);
```

Данная функция принимает строку с названием файла, который необходимо удалить. Функция при успешном выполнении возвращают ноль и ненуловое значение при ошибке. Следующая строка кода удаляет файл test.txt:

```
remove("test.txt");
```

Помимо удаления можно переименовать файл при помощи функции rename:

```
int rename(const char *oldname, const char *newname);
```

Данная функция принимает две строки: первая – это имя файла, который необходимо переименовать, вторая – новое имя для файла. Функция при успешном выполнении возвращает ноль и ненуловое значение при ошибке. Следующая строка кода переименовывает файл *test.txt* в *new_name.txt*:

```
rename("test.txt", "new_name.txt");
```

Если есть необходимость в создании временного файла, то можно воспользоваться функцией *tmpfile*:

```
FILE *tmpfile();
```

Данная функция открывает файловый поток для нового бинарного файла (файл открывается в режиме *"w+b"*), имя которого гарантированно не совпадает ни с каким другим в данной папке. При закрытии полученного файлового потока файл удаляется.

При работе с содержимом файлов может показаться, что данные записываются в файловый поток при вызове соответствующих функций (*fwrite*, *fprintf*, и т. д.), однако это не так. В операционных системах реализованы оптимизации, связанные с вводом и выводом информации через файловые потоки. Когда вызывается функция на запись, данные изначально попадают в специальный буфер. При определенных событиях, например, когда буфер закончился или файл был закрыт, данные из буфера записываются непосредственно в файл. Однако есть еще один способ вызвать запись данных из буфера в файл – при помощи функции *fflush*:

```
int fflush( FILE *stream);
```

Данная функция принимает указатель на файловый поток, для которого необходимо записать данные из буфера в файл. Функция возвращает ноль при успешной записи **EOF** при возникновении ошибки.

Помимо чтения и записи данных можно управлять местоположением откуда происходит чтение и куда происходит запись. Функция *ftell* позволяет получить текущую позицию в файловом потоке:

```
long int ftell( FILE *stream);
```

Данная функция принимает указатель на файловый поток, для которого необходимо получить текущую позицию. При успешном выполнении возвращает неотрицательное значение текущей позиции, а при ошибке возвращает –1. Если происходит работа с бинарными данными, то *ftell* возвращает количество байт от начала файла. При работе с текстовыми файлами такая интерпретация полученного значения ошибочна. Для перехода на сохраненную позицию необходимо использовать функцию *fseek*:

```
int fseek(FILE *stream, long int offset, int origin);
```

Данная функция принимает следующие аргументы:

• *FILE *stream* – указатель на файловый поток, для которого необходимо установить новую позицию;

• *long int offset* – смещение, которое было получено при помощи *ftell*, относительно аргумента *origin;*

• int *origin* – одна из следующих констант:

  ○ **SEEK_SET** – указывает на начало файла.

  ○ **SEEK_CUR** – указывает на текущую позицию.

  ○ **SEEK_END** – указывает на конец файла. Данное значение поддерживается не на всех системах.

Функция возвращает ноль при успешном выполнении и ненулевое значение при ошибке. При работе с текстовыми данными аргумент *origin* всегда должен быть установлен в *SEEK_SET*. Далее представлен пример кода с использованием *ftell* и *fseek*:

```
#include <stdio.h>

void main(){
   FILE *file = fopen("test.txt", "w"); // открытие файла на запись
   if(file){ // если файл успешно открыт
       fputs("Start of file, ", file);
       long int pos = ftell(file); // сохранение позиции
       fputs("My best string", file);
       fseek(file, pos, SEEK_SET); // восстановление позиции
       fputs("New string", file);
       fclose(file); // закрытие файла
   }else
       puts("Не удалось открыть файл test.txt");
   return 0;
}
```

После выполнения данного кода в файле *test.txt* будет находится следующая строка:

Start of file, New stringring

Помимо функций *ftell* и *fseek* существуют еще две функции с таким же функционалом – *fgetpos* и *fsetpos*:

```
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos );
```

Отличие данных функций от *ftell* и *fseek* в том, что они используют не числовое обозначение для работы с позицией, а структуру *fpos_t*. Работа напрямую со структурой *fpos_t* не предполагается, поэтому ее содержание рассмотрено не будет. Данные функции принимают указатель на файловый поток, для которого необходимо получить (функция *fgetpos*) или установить (функция *fsetpos*) текущую позицию и указатель на структуру *fpos_t*, в которую будет записано текущее значение позиции (функция *fgetpos*) или из которой будет установлено новое значение текущей позиции (функция *fsetpos*). Функции при успешном выполнении возвращают ноль и ненуловое значение при ошибке. Далее представлен модифицированный пример предыдущего кода, где *ftell* и *fseek* заменены *fgetpos* и *fsetpos*:

```
#include <stdio.h>

void main(){
   FILE *file = fopen("test.txt", "w"); // открытие файла на за-
пись
   if(file){ // если файл успешно открыт
        fpos_t pos;
        fputs("Start of file, ", file);
        fgetpos(file, &pos); // сохранение позиции
        fputs("My best string", file);
        fsetpos(file, &pos); // восстановление позиции
        fputs("New string", file);
        fclose(file); // закрытие файла
   }else
        puts("Не удалось открыть файл test.txt");
   return 0;
}
```

Результат выполнения данного кода аналогичен коду с *ftell* и *fgetpos*.

Для работы с директориями используется поток директории, реализованный структурой *DIR* (по аналогии с файловым потоком *FILE*), которая используется в качестве аргумента для функций. Так как напрямую работа с

данной структурой не производится, то её структура не будет рассмотрена. Перед началом работы с директорией её необходимо открыть. Для этого необходимо использовать функцию *opendir* (для её использования необходимо подключить заголовочный файл *dirent.h*):

```
DIR *opendir(const char *dirname);
```

Данная функция открывает директорию, которая находится по пути dirname. Путь может быть абсолютный (например, “/home/user/work”), или относительный (например, “../work”). Функция возвращает указатель на ранее описанную структуру *DIR*. Если открыть директорию не удалось, то функция возвращает *NULL*. После завершения работы с директорией ей необходимо закрыть. Сделать это можно при помощи функции *closedir*:

```
int closedir (DIR *dir);
```

Данная функция принимает в качестве аргумента указатель на полученную при открытии директории структуру *DIR*. Функция возвращает 0 при успешном закрытии директории и -1 при возникновении ошибки.

Как было ранее сказано, структура *DIR* представляет поток директории, из которого можно получить информацию об элементах в директории. Получить очередной элемент, содержащийся в директории, можно при помощи функции *readdir*:

```
struct dirent *readdir (DIR *dirstream);
```

Данная функция принимает в качестве аргумента указатель на полученную при открытии директории структуру *DIR*. Функция возвращает указатель на структуру dirent, которая содержит информацию об элементе (файле). Если элементов в директории больше нет или произошла ошибка, то функция возвращает *NULL*. Данная функция имеет две особенности. Первая – функция *readdir()* на некоторых системах может не возвращать структуру *dirent* для *./* (текущая директория) и *../* (родительская директория). Вторая – возвращаемый указатель функцией readdir() указывает на внутренний буфер, выделенный при открытии директории. При последующем вызове функции *readdir()* данные могут быть перезаписаны, поэтому необходимо их скопировать, если они понадобятся позже. Также нет необходимости в освобождении указателя, возвращаемого функцией *readdir()*. Освобождение внутреннего буфера происходит при вызове функции *closedir()*.

В Linux, с точки зрения операционной системы, все объекты файловой системы являются файлами. Например, директория также является файлом и содержит информацию и содержащихся в нее файлах.

Структура *dirent* содержит информацию о файле. Основную информацию содержат следующие поля:

• Поле *d_name* (тип *char*[]) – имя файла, которое является строкой, заканчивающаяся символом конца строки;

• Поле *d_type* (тип *unsigned char*) – тип файла, основными значениями являются:

○ **DT_UNKNOWN** – неизвестный тип файла;

○ **DT_REG** – обычный (регулярным) файл, который можно открыть на чтение/запись;

○ **DT_DIR** – директория;

○ **DT_FIFO** – именновый pipe;

○ **DT_SOCK** – сокет;

○ **DT_CHR** – символьное устройство;

○ **DT_BLK** – блочное устройство;

○ **DT_LNK** – символьная ссылка.

Подробнее про типы файлов можно прочитать в [11].

2. ЮНИТ-ТЕСТИРОВАНИЕ

Юнит-тестирование (модульное тестирование) – процесс проверки отдельных частей программы на наличие ошибок. Отдельными частями (модулями, юнитами) программы являются небольшие фрагменты кода [12]. Важно понимать, что юнит-тестированием обычно занимаются сами программисты. Юнит-тестирование – самый базовый уровень тестирования программного обеспечения, потому что тестирует не общий результат программы, а результат каждого фрагмента кода.

Цели юнит тестирования:

• автоматизация тестирования;

• быстрое тестирование.

В языке Си юнит-тестирование можно реализовать с помощью различных фреймворков, например, CUnit [13], или же с помощью макроса assert заголовочного файла ***assert.h*** стандартной библиотеки Си. Разница между использованием стороннего фреймворка и стандартной библиотеки в том, что assert позволяет проверять только бинарное значение (справедливо ли поданное на вход выражение), а в фреймворках уже реализован ряд функций для проверки различных типов данных – чисел с плавающей точкой, строк и др. Кроме того, фреймворки иногда упрощают написание тестов.

Далее будем рассматривать только стандартный способ юнит-тестирования – макрос *assert*. Макрос *assert* принимает на вход выражение и проверяет его на справедливость и если выражение несправедливо, то выводит ошибку в поток *stderr*. Рассмотрим следующий код:

```
#include <assert.h>

int mul(int a, int b){
   return a * b;
}

int main ()
{
   assert (mul(5, 6) == 30);
   assert (mul(-5, 6) == -30);
   assert (mul(64000, 64000) == 4096000000);


   return 0;
}
```

В данной программе реализована функция, которая возвращает произведение двух чисел. В функции main проверяются ситуации с различными значениями – большими и маленькими, положительными и отрицательными. Результатом работы такой программы будет:

a.out: main.c:13: main: Assertion `mul(64000, 64000) == 4096000000' failed.
Аварийный останов (образ памяти сброшен на диск)

Очевидно, что произошло переполнение при возврате значений, так как результат произведения больше максимального значения для типа int, и результат функции не соответствует ожиданиям. Иногда тесты пишутся до написания кода, чтобы исправлять ошибки сразу в процессе разработки программы. Поэтому данную программу сразу легко исправить, заменив тип возвращаемого значения и аргументов функции на long, чтобы пройти все тесты. Конечно же юнит-тестирование не покрывает всевозможные ситуации, но значительно улучшает качество кода и работу программы.

3. УКАЗАТЕЛИ НА ФУНКЦИИ

Указатель на функцию позволяет работать с функцией как с обычной переменной, в том числе передавать функцию в качестве аргумента в другую функцию. Например, функция *qsort* стандартной библиотеки, которая сортирует массив данных определенного типа. Ее сигнатура выглядит так:

```
    void qsort(void *base, size_t num, size_t size, int
(*compare) (const void *, const void *))
```

При этом есть важные детали:

• функция *qsort* для сортировки сравнивает элементы друг с другом, но алгоритм сравнения определяет программист, поскольку для одного и того же типа данных нет единственного варианта сравнения (например, для целых чисел можно сравнивать как по убыванию так и по возрастанию);

• *qsort* – функция, которая должна работать с любым типом данных (обратите внимание на *void*-указатели в сигнатуре), поскольку значительная часть алгоритма сортировки, за исключением операции сравнения, от типа данных сортируемого массива не зависит.

Для решения этой задачи используется механизм указателя на функцию: реализуется общая функция (*qsort* из примера), то есть код, не зависящий от типа данных, куда вставляется вызов вспомогательной функции(й). Чтобы лучше понять этот механизм рассмотрим подробно другой пример.

Предположим, перед вами стоит задача: написать функцию нахождения минимума в массиве элементов. Функция должна возвращать указатель на минимальный элемент. При этом тип элементов массива может быть любым: целые числа, вещественные числа, структуры и т. п.

Указатели типа *void** пригодятся при объявлении такой функции:

```
void* min(void*arr, int array_length, int size_of_element)
```

Потому что они позволяют подать функции на вход в качестве аргумента массив элементов любого типа. Длина массива и размер элемента нужны для перебора всех элементов массива (так как тип в общем случае может быть любой).

Удобно для понимания начать с функции нахождения минимума для типа int и затем ее обобщить. Итак функция нахождения минимума для типа int:

```
int min_index = 0;
for(int i = 0; i < array_length; i++){
    if(arr[i] > arr[min_index])
    min_index = i;
}
return &arr[min_index];
```

Базовая суть заключается в том, что программа циклом проходит по массиву и находит индекс минимального элемента, а затем возвращает указатель на него. Как можно эту функцию обобщить для работы с любым типом

данных? Для этого надо подумать, что будет, если в функцию подать не массив целых чисел (*int[]*), а массив строк (*char*[]*)? Тогда сравнение:

```
arr[i] > arr[min_index]
```

является сравнением адресов (указателей как чисел), что не несет в себе смысла. И тут возникает 2 вопроса:

• что в таком случае должно быть операцией сравнения для строк?

• как сделать так, чтобы функция позволяла сравнивать и строки и числа (а в законченной форме – любые типы)?

Операция сравнения должна быть настраиваемой, то есть это должна быть функция, вызов которой заменит операцию сравнения и будет возвращать результат, аналогичный операциям сравнения.

Разыменовывать указатель типа *void** (массив arr) нельзя, так как получить данные по указателю, не зная, какого они размера, невозможно. Остается вариант – выяснить все типы, с которыми программист будет работать, привести результирующий указатель (*void**) к каждому типу и получать результат сравнения отдельно для каждого типа. Но тогда функция становится неудобной в использовании и плохо масштабируемой, потому что будет дублироваться часть функциональности.

Указатели на функцию решают обе проблемы. В нашем примере функциональность, которая зависит от типа данных, это операция сравнения элементов, поиск минимума:

```
arr[i] > arr[min_index]
```

Как указано ранее, такое сравнение является сравнением адресов (чисел), что само по себе лишено смысла, потому что адрес в памяти никак не коррелирует с содержанием самой строки. Осмысленный пример – сравнение строк по их длине, но чтобы это сделать нужен более сложный код, чем операция сравнения “>”. Поэтому операцию сравнения нужно выделить в отдельную функцию и подавать её указатель в исходную функцию вместе с массивом сортируемых данных.

Похожая логика используется во многих языках программирования, а функцию сравнения двух элементов обычно называют компаратор (от англ. compare – сравнивать). Компаратор работает по следующему принципу: если элементы равны, результатом сравнения будет 0, если первый больше – результат 1 или любое положительное число, иначе –1 или любое отрицательное.

Итак, есть функция-компаратор для сравнения строк по длине. Почему указатели приводятся к типу *char*** станет понятно позже:

```
int str_compare(const void* a, const void* b) {
   char* s1 = *(char**)a;
   char* s2 = *(char**)b;
   int res = strlen(s1) - strlen(s2);
   return res;
}
```

Также нужно разобрать синтаксис указателей на функцию, так как функция *min* должна получать такой указатель на вход. Как в общем виде объявить указатель на функцию:

```
тип_возвращаемого значения (*имя_указателя) (аргументы функции);
void (*example)(int)
```

В случае с функцией-компаратором

```
int (*example)(const void*,const void*)
```

Теперь функция min. Обратите внимание на изменения: вместо обращения по индексу в массиве – сдвиг указателя на *I * size*, где *size* – фиксировано (размер указателя на строку), а *i* – номер элемента. Так становится возможным перемещение по элементам массива без обращения по индексу. Подумайте зачем это понадобилось и почему в данном случае по индексу обратиться нельзя:

```
void* min(void* arr, int array_length, int size_of_element, int
(*compar)(const void*,const void*)){
   void* min = arr;
   for(int i = 1; i < array_length; i++){
       if(compar(arr + i*size_of_element, min) < 0)
           min = arr + i*size_of_element;
   }
   return min;
}
```

Теперь можно вызывать функцию min для любого типа. В контексте задачи поиска минимальной строки в массиве её использование может выглядеть так:

```
#define STR_SIZE 90


char** gen_text(int n){
   char** text = (char**)calloc(n, sizeof (char*));
   printf("P: %p\n", text);
```

```
    for(int i = 0; i < n; i++) {
        size_t size = 10 + rand()% STR_SIZE;    // случайная длина
от 10 до 100
        text[i] = (char*)calloc(size,sizeof (char));
        for(int k = 0; k < size; k++) {
            text[i][k] = 'A' + rand()% 10; // случайный заглавный
символ от A до J
        }
        printf("Str gen: %p %s\n", &text[i], text[i]);
    }
    return text;
}

int main() {


    char** text = gen_text(6);
    char** min_el = min(&text[0], 6, sizeof (char*),
str_compare);

    printf("_______________\n");
    printf("RES: %s", *min_el);

    return 0;
}
```

Функция *gen_text* – вспомогательная, генерирует массив строк длины от 10 до 100. Обратите внимание, функция min получает на вход адрес нулевого элемента массива text. Это нужно, потому что функция min ведет сдвиг указателя на размер элемента *sizeof(char*)*. По этой же причине функция-компаратор разыменовывает двойной указатель:

```
char* s1 = *(char**)a;
char* s2 = *(char**)b;
```

Тип элемента *text[0]* – *char**, таким образом его адрес будет типа *char***.

Список литературы

1. Datatracker. Textual Representation of IPv4 and IPv6 Addresses. URL: https://datatracker.ietf.org/doc/html/draft-main-ipaddr-text-rep-02

2. Linux manual page. scanf(3). URL: https://man7.org/linux/man-pages/man3/scanf.3.html

3. Linux manual page. printf(3). URL: https://man7.org/linux/man-pages/man3/printf.3.html

4. Фридл Дж. Регулярные выражения / пер. с англ. СПб.: СимволПлюс, 2008.

5. Jan Goyvaerts, Regular Expressions: The Complete Tutorial, 2007 Jan Goyvaerts. URL: https://www.amazon.com/Regular-Expressions-Complete-Jan-Goyvaerts/dp/1411677609

6. Blog. Making string validation faster by not using a regular expression. A story. URL: https://blog.maartenballiauw.be/post/2017/04/24/making-string-validation-faster-no-regular-expressions.html

7. Керниган Б., Ритчи Д. Язык программирования С / пер. с англ. СПб.: Невский Диалект, 2004.

8. Томас К. Алгоритмы. Построение и анализ. М.: Вильямс. 2005.

9. Robert Lafore. Object-Oriented Programming in C++, Fourth Edition, 2002. URL: https://faculty.ksu.edu.sa/sites/default/files/ObjectOrientedProgrammingin C4thEdition.pdf

10. David Vandevoorde Nicolai M. Josuttis Douglas Gregor. C++ Templates. The Complete Guide, 2nd Edition, 2018. URL: https://www.amazon.com/C-Templates-Complete-Guide-2nd/dp/0321714121

11. Лав Р. Linux. Системное программирование. СПб.: Питер. 2014.

12. Хориков В. Принципы юнит-тестирования. СПб.: Питер. 2021.

13. CUnit / A Unit Testing Framework for C. URL: https://cunit.sourceforge.net/index.html

Оглавление

Заславский Марк Маркович
Лисс Анна Александровна
Гаврилов Андрей Владимирович
Глазунов Сергей Алексеевич
Государкин Ярослав Сергеевич
Тиняков Сергей Алексеевич
Голубева Валентина Петровна
Чайка Константин Владимирович
Допира Валерия Евгеньевна

Базовые сведения к выполнению лабораторных работ по дисциплине «Программирование». Второй семестр

Учебно-методическое пособие

Редактор М. Б. Шишкова

Компьютерная верстка Е. С. Рыбец

Подписано в печать 02.02.24. Формат 60×84 1/16.
Бумага офсетная. Печать цифровая. Печ. л. 5,25.
Гарнитура «Times New Roman». Тираж 91 экз. Заказ .

Издательство СПбГЭТУ «ЛЭТИ»
197022, С.-Петербург, ул. Проф. Попова, 5Ф