

Работа №6

«Рекурсия»

Цель работы

Получение навыков написания рекурсивных алгоритмов.

После выполнения работы студенты могут

- Составлять рекурсивные алгоритмы.
- Использовать рекурсию для решения рекуррентных задач.
- Оценивать необходимую глубину рекурсии для реализации алгоритма.

Теоретическая часть

В любом современном языке программирования существует возможность разбить исходный код программы на отдельные функции или процедуры. Такой подход позволяет структурировать программу, представить ее в виде отдельных исполняемых блоков. Существующая парадигма структурного программирования предполагает, что программа разделяется на функции, находящиеся в иерархической зависимости. При вызове функции вызывающая функция приостанавливается до окончания выполнения вызываемой, после чего может считать результат ее работы и продолжить выполнение дальше.

Поскольку потенциально любая функция может вызвать любую, существует ситуация, когда функция вызывает сама себя. Такой вызов называется **рекурсивным**, а сам механизм, при котором функция вызывает сама себя, называется **рекурсией**. Разделяют прямую и косвенную рекурсию. При **прямой** рекурсии функция вызывает сама себя непосредственно. При **косвенной** функция вызывает какую-либо стороннюю функцию, которая снова вызывает исходную.

Рекурсия служит для обработки рекуррентных задач, в которых расчет следующего шага зависит от результата, полученного на предыдущем шаге. Примером такой задачи может служить расчет факториала: каждый элемент последовательности равен произведению своего порядкового номера на

предыдущий элемент, а первый по счету элемент равен 1. Также рекурсия применяется для задач, которые можно разбить на подзадачи, аналогичные исходной. Например, задача поиска файла в папке: при нахождении подпапки, программа переходит в подпапку и фактически решает ту же самую задачу, что и изначально.

Очевидно, что ситуация, в которой функция вызывает сама себя, может привести к заикливанию программы, если не предпринять специальных мер. В рекурсивной функции всегда должно существовать условие, при выполнении которого функция прекратит дальнейшую рекурсию. Такое условие выделяет специальную ветвь расчета — **терминальную**, а остальные ветви, соответственно, называются **рекурсивными**. В приведенном примере расчета факториала, расчет для $n=1$ является терминальной ветвью, а все остальные — рекурсивными. Покажем реализацию данного алгоритма на языке Python:

```
def F(x):  
    if x < 2:  
        return 1;  
    return x * F(x-1)  
  
Y = F(5)  
print(Y)
```

Можно представить рекурсивную функцию как набор вызовов, каждый из которых работает со своим набором аргументов. Диаграмма на рисунке 1 показывает, как происходит расчет факториала числа 5 рекурсивным методом. Стрелка справа показывает цепочку рекурсивных вызовов функции, а стрелка слева — цепочку рекурсивных возвратов.

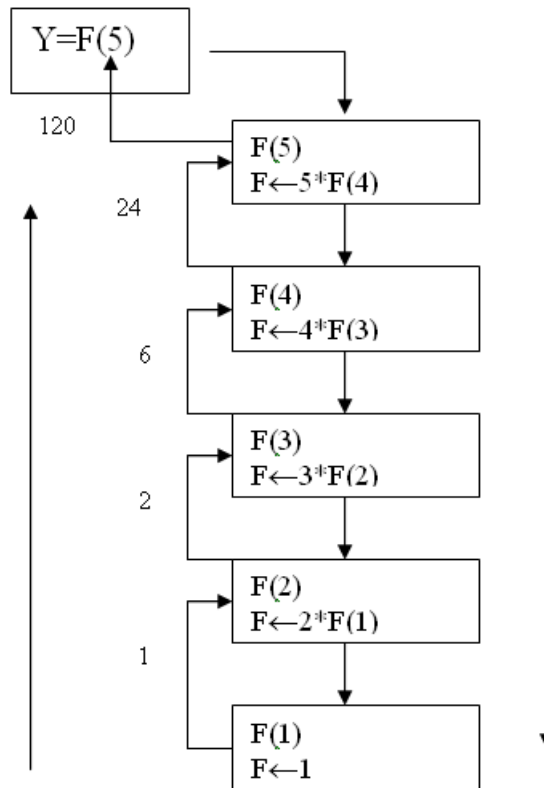


Рисунок 1 - Вычисление факториала рекурсивным методом

Использование рекурсии является источником нетривиальных ошибок. Для того, чтобы понять, как реализуется рекурсия, следует понять, как вообще производится вызов функции.

При вызове функции программа должна передать аргументы вызываемой функции, а также "запомнить", в каком месте прервалось выполнение вызывающей функции, чтобы затем вернуться и продолжить выполнение с того же места. В связи с этим, вызов функции несет с собой дополнительные накладные расходы и обладает своими особенностями, которые следует учитывать при разработке программного обеспечения.

Компилируемые языки программирования. При компиляции программы вызов функции преобразуется в стандартный набор двоичных команд процессора. Для того, чтобы сохранить порядок вызова и передать аргументы, на уровне машинных команд используется стек. Стек — это специальный участок памяти программы, который начинается с зарезервированного адреса (в примере обозначен как Base) и заканчивается специальным адресом — указателем стека

(SP, Stack Pointer). Стек представляет собой колодец: данные записываются в него в конец, после чего указатель SP передвигается выше на размер записанных в стек данных. При удалении данных из стека, указатель SP просто передвигается ниже. Организация стека схематично представлена на рисунке 2. Современные процессоры содержат специальный регистр (ESP), хранящий адрес вершины стека, а также набор команд для работы со стеком (push для добавления значения в стек и pop для удаления значения из стека).

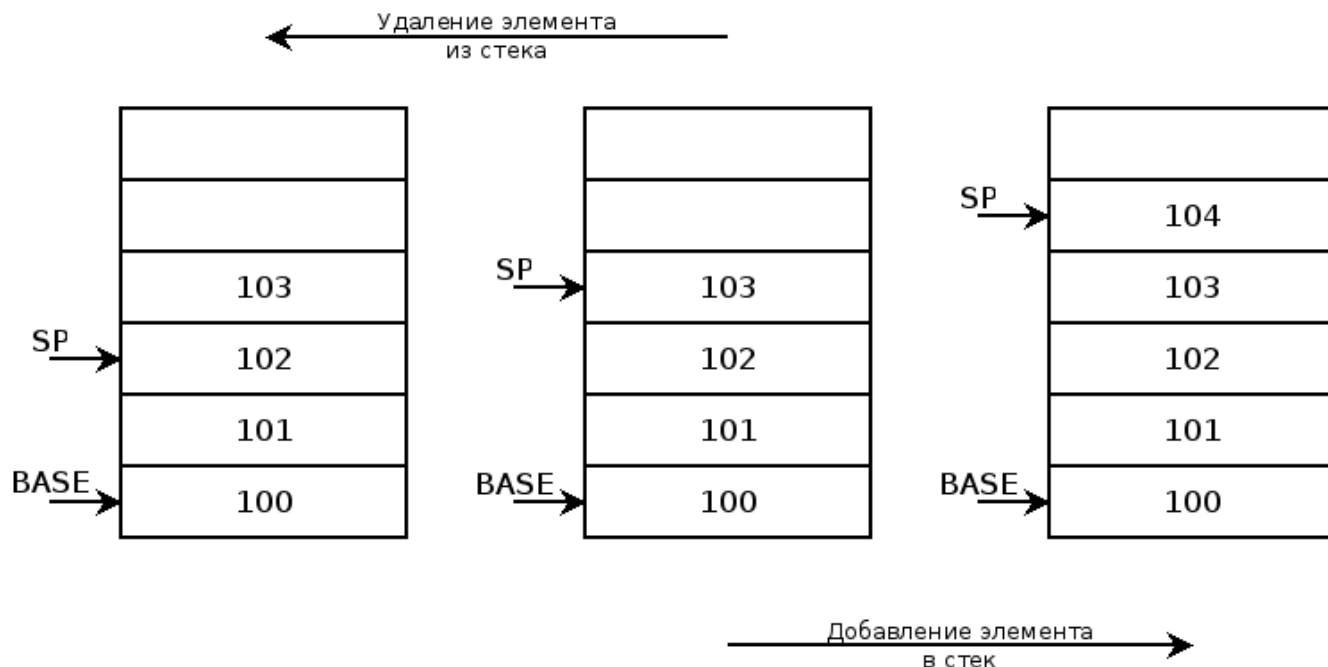


Рисунок 2 - Иллюстрация работы стека

Каждая функция может распоряжаться стеком по своему усмотрению, однако существуют два ограничения:

1. По окончании работы функции стек должен вернуться в исходное состояние;
2. Суммарный объем стека (разница между SP и Base) не должен превысить значения, определенного платформой (порядка единиц мегабайт).

При вызове функции в стек помещается адрес инструкции, к которой следует вернуться после окончания работы вызываемой функции, а затем последовательно помещаются параметры этой функции. Рассмотрим следующий простой пример:

```
20: int main()
```

```

21: {
22:     int a = 4;
23:     int b=pow(a, 2);
24:     printf("%d\n", b);
25:     return 0;
26:}

```

Перед вызовом функции `pow` в строке 23 стек будет содержать две переменные, `a` и `b`, которые были созданы функцией `main`. Непосредственно перед вызовом `pow` в стек будут помещены значения аргументов, передаваемых в функцию (конкретный порядок расположения аргументов зависит от платформы). Затем вызывается машинная команда `call`, аргументом которой является адрес вызываемой функции. Эта команда помещает в стек так называемый **адрес возврата**. После того, как все машинные команды функции будут выполнены, будет вызвана команда `ret`, возвращающая выполнение на тот адрес возврата, который располагался на стеке при вызове функции, указатель `SP` сдвинется ниже, и стек будет возвращен в первоначальное состояние. Возвращаемое значение функции непосредственно после выхода из функции будет располагаться в накопительном регистре процессора (`EAX`), откуда впоследствии оно будет скопировано в участок стека, соответствующий переменной `b`. Рисунок 3 иллюстрирует состояние стека перед вызовом строки 23 и непосредственно в начале работы функции `pow`.

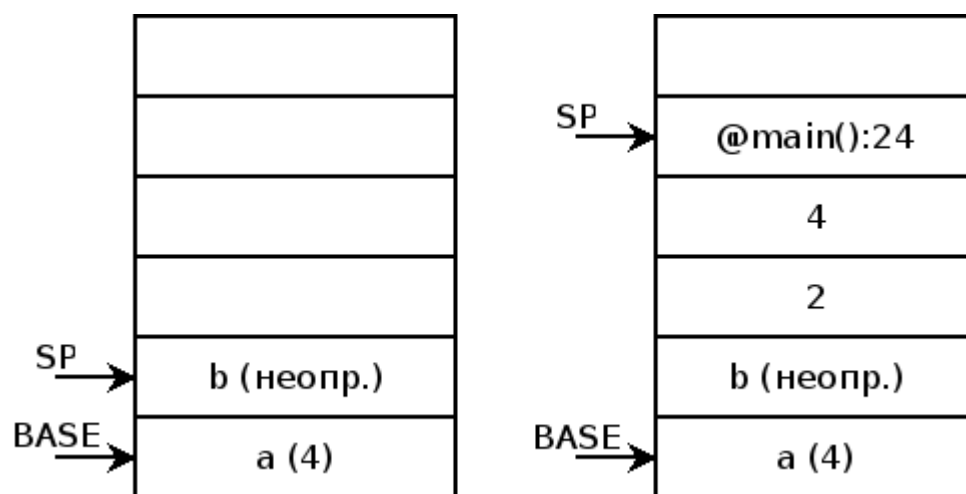
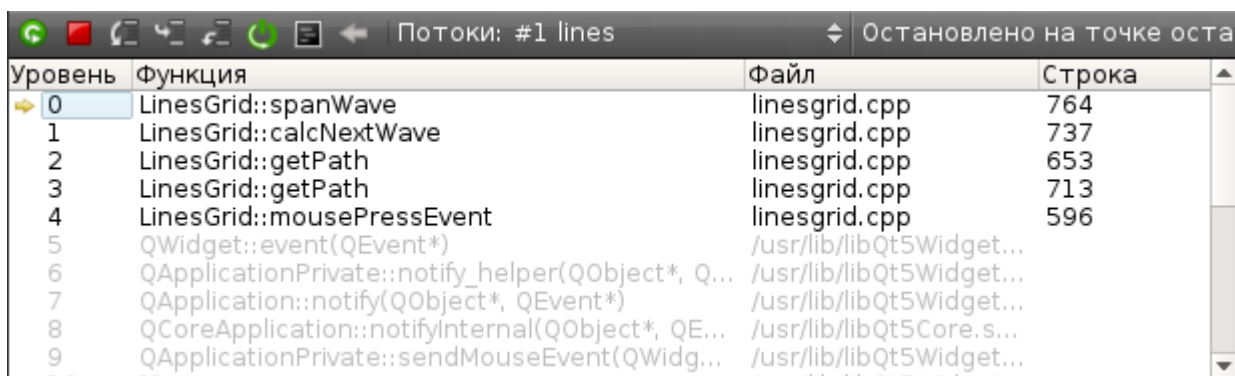


Рисунок 3 - Стек при вызове функции

Существует такое понятие как **стек вызова функций** (англ. Call Stack). Он показывает, какая функция в каком порядке была вызвана, с какими аргументами, и куда в итоге из каждой функции должно вернуться выполнение программы. Стек вызова не равен аппаратному стеку, рассмотренному выше, но может быть построен на основе аппаратного стека, зная исходный код функций. Эту задачу решает отладчик в процессе отладки приложения. На рисунке 4 показано окно стека вызова в отладчике Qt Creator. Используя данное окно, можно легко проследить, каким образом выполнение программы дошло до того места, где программа была остановлена, и какие аргументы при этом передавались в функцию.



Потоки: #1 lines Остановлено на точке оста

Уровень	Функция	Файл	Строка
0	LinesGrid::spanWave	linesgrid.cpp	764
1	LinesGrid::calcNextWave	linesgrid.cpp	737
2	LinesGrid::getPath	linesgrid.cpp	653
3	LinesGrid::getPath	linesgrid.cpp	713
4	LinesGrid::mousePressEvent	linesgrid.cpp	596
5	QWidget::event(QEvent*)	/usr/lib/libQt5Widget...	
6	QApplicationPrivate::notify_helper(QObject*, Q...	/usr/lib/libQt5Widget...	
7	QApplication::notify(QObject*, QEvent*)	/usr/lib/libQt5Widget...	
8	QCoreApplication::notifyInternal(QObject*, QE...	/usr/lib/libQt5Core.s...	
9	QApplicationPrivate::sendMouseEvent(QWidget...	/usr/lib/libQt5Widget...	

Рисунок 4 - Стек вызова в Qt Creator

Очевидно, что каждый дополнительный вызов функции — это дополнительная нагрузка на стек. В случае с обычными функциями глубина стека вызовов редко превышает 10-20. Однако в случае с рекурсивным вызовом нагрузка на стек может существенно возрасти. Дополнительная нагрузка обеспечивается локальными переменными, также размещаемыми в стеке. В конечном счете это может привести к тому, что оперативная память, отведенная на стек, закончится, и произойдет ошибка переполнения стека (англ. **Stack Overflow**). При возникновении такой ошибки приложение аварийно завершается операционной системой.

В **интерпретаторах** (например, в Python) используется сходная схема вызова функций. Интерпретатор запоминает у себя последовательность вызовов в виде стека, аналогично аппаратному стеку, однако вместо адресов хранит у себя номер строки файла исходного кода, к которой следует вернуться по окончании функции. При этом стек интерпретатора также ограничен, причем он меньше, чем стек

двоичной программы, но ошибка, связанная с переполнением стека, может быть штатно обработана приложением. К примеру, следующий программный код на Python проверяет глубину стека и перехватывает ошибку при возникновении его переполнения:

```
def rec(i):  
    print("Recursion depth="+str(i))  
    try:  
        rec(i+1)  
    except RuntimeError as re:  
        print(re.args)  
        return
```

```
rec(0)
```

В приведенном листинге функция `rec` будет вызывать сама себя до тех пор, пока не возникнет ошибка `RuntimeError`. Аргументом функции будет являться текущая глубина вложенности. Вывод данного скрипта выглядит следующим образом:

```
<...>  
Recursion depth=995  
Recursion depth=996  
('maximum recursion depth exceeded while calling a Python object',)
```

Отметим, что аналогичная программа на языке C приведет к переполнению стека при глубине порядка 250 тысяч.

Рекурсивные алгоритмы используются на практике для нескольких классов задач. Одним из них являются задачи по обработке дерева. Дерево представляет собой упорядоченную структуру данных, в которой каждый элемент имеет несколько "дочерних" элементов. При этом элементы в дереве выстраиваются специальным образом. Например, каждый элемент дерева, оптимизированного для поиска, содержит два дочерних элемента, причем все элементы дерева слева всегда меньше родительского, а справа — больше родительского. Пример такого дерева представлен на рисунке 5.

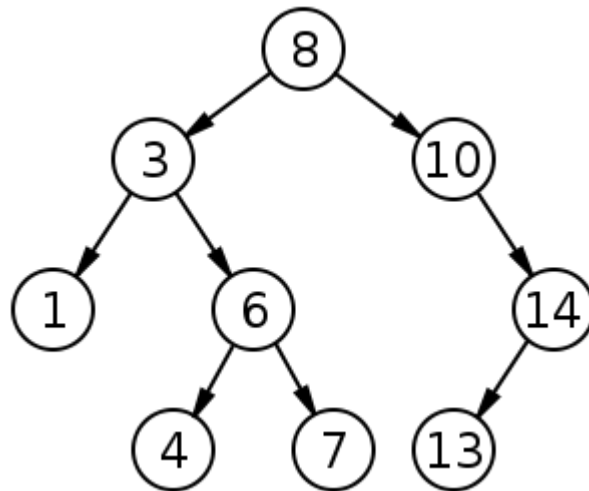


Рисунок 5 - Двоичное дерево

Алгоритм поиска в таком дереве можно представить следующим образом:

1. Посмотреть, является ли данный элемент искомым. Если да — вернуть его значение и завершить функцию.
2. Если нет — посмотреть дочерние элементы. Если дочерние элементы отсутствуют — вернуть значение "элемент не найден".
3. Если у элемента есть дочерние элементы, то необходимо запустить аналогичный поиск для каждого из дочерних элементов.
4. Если поиск по дочерним элементам нашел нужный элемент — вернуть его и завершить функцию, в противном случае вернуть значение "элемент не найден".

Следует отметить, что выбрав, например, в качестве стартовой точки элемент №3 и убрав из рассмотрения элементы выше и правее (номера 8, 10, 13 и 14), мы получаем такое же дерево. То есть обработка любого узла дерева может производиться одним и тем же алгоритмом, у которого меняются начальные данные.

Аналогичным образом обрабатываются другие древовидные структуры, такие как файловое дерево или XML-файл.

Другое применение рекурсии — разделение массива на части для последующей обработки по частям. Рекурсия используется в алгоритмах сортировки, таких как быстрая сортировка или сортировка слиянием для

разделения массива на части или его объединения. Алгоритм сортировки слиянием подразумевает разделение массива на части до тех пор, пока части не станут единичной длины, с последующим их объединением. Фрагмент реализации данного алгоритма на С выглядит следующим образом:

```
void merge(int* in, int *out, int size)
{
    if(size <= 1)
        return;

    int mid = size/2;
    merge(out, in, mid);
    merge(out+mid, in+mid, size-mid);

    // Слияние двух отсортированных подмассивов: [0, mid) и [mid,
    size)
}
```

Условие `if (size<=1)` формирует терминальную ветвь функции. Глубина рекурсии однозначно определяется как логарифм длины массива по основанию 2.

Другая задача, решаемая при помощи рекурсии — поиск числа в отсортированном массиве. Она решается следующим образом:

1. Если массив единичной длины, и его элемент не является искомым числом — вернуть значение "элемент не найден".
2. Взять число из середины массива.
3. Если оно равно искомому элементу — вернуть его и завершить поиск.
4. Если число из середины массива меньше искомого элемента — запустить поиск по правой половине массива, в противном случае — по левой половине массива.

Глубина рекурсии в таком алгоритме также вычисляется как $\log_2 N$, где N — длина массива.

Порядок проведения работы

В данной работе необходимо разработать приложение, которое ищет кратчайший путь через лабиринт. Лабиринт представляет собой прямоугольное поле, разделенное на клетки. Каждая клетка может быть проходимой или не проходимой. Лабиринт задается в виде текстового файла, проходимый участок лабиринта обозначается как символ пробела, а все остальные символы обозначают непроходимые участки. При этом вход в лабиринт расположен на его верхней стороне, а выход — на нижней. Перемещаться по лабиринту можно только по горизонтали и вертикали. Пример лабиринта:

```
# #####
#      #      #      #
# # # ##### # ### #
# # #      # #      #
# # ##### # ###
# # #      #      #
##### ##### # # # #
#      #      # # # #
# # # # # ##### ### #
# #      #      # #      #
# ##### # # # # #
#      #      # #
##### #
```

В процессе поиска выхода необходимо составить маршрут, содержащий точки. В языке C++ рекомендуется использовать для хранения маршрута тип `vector<Point>`, где `Point` определяется следующим образом:

```
struct Point
{
    unsigned x;
    unsigned y;
};
```

В языке Python рекомендуется для хранения маршрута использовать список, каждый элемент которого — это комплексное число, либо список из двух элементов.

Для генерации лабиринта существует программа, внешний вид которой представлен на рисунке 6. Программа может сгенерировать массив с заданными параметрами, отобразить его на экран, а также сохранить в файл, используя заданный символ в качестве символа непроходимой клетки («стенки»).

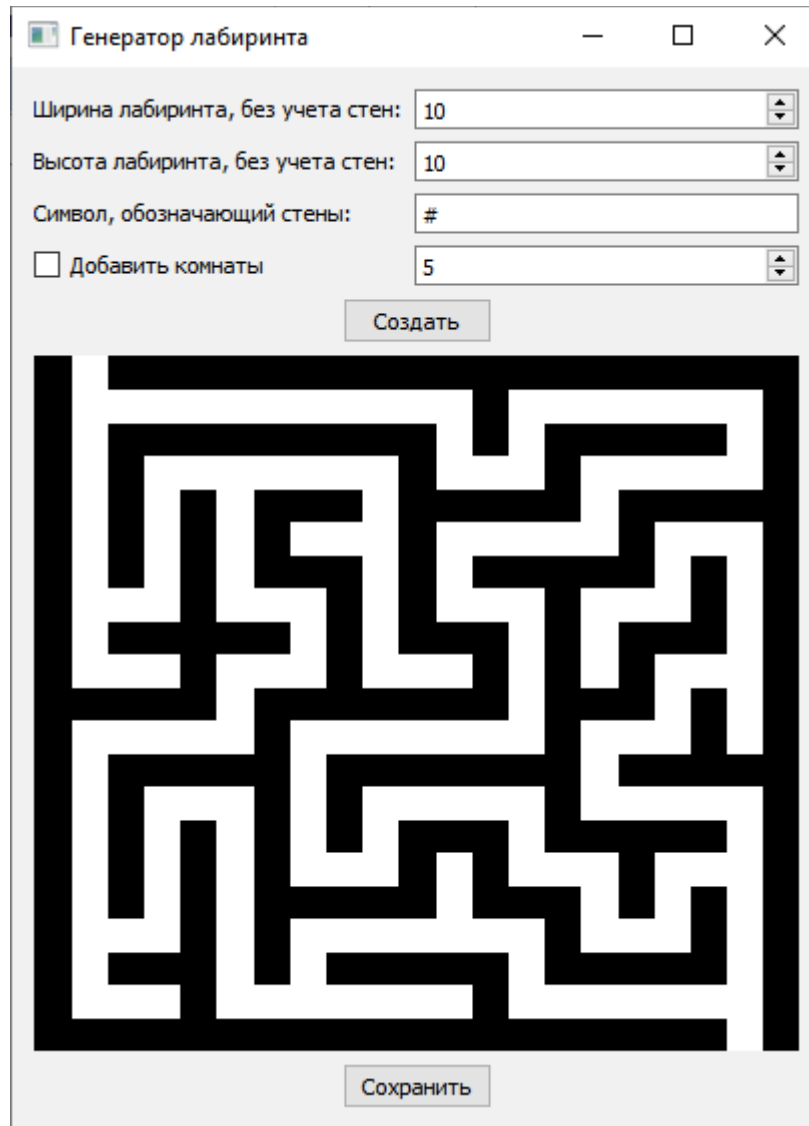


Рисунок 6 - Генератор лабиринта

Алгоритм поиска маршрута по лабиринту следующий:

1. Выбрать стартовую точку в лабиринте и отметить ее символом "_", который

означает, что данная точка уже была посещена в процессе поиска. Дописать данную точку в маршрут.

2. В случае, если рядом с текущей точкой расположена только одна проходимая клетка, необходимо переместить текущую точку на нее и также отметить эту клетку символом "_". Координату новой точки следует дописать в маршрут.
3. Если в процессе поиска найден выход из лабиринта, следует вернуться из функции, передав составленный маршрут в качестве возвращаемого значения.
4. Если в процессе поиска найден тупик, следует очистить запомненный маршрут и вернуть из функции пустой маршрут, что означает, что в данном направлении выход не найден.
5. Если рядом с текущей точкой найдено несколько проходимых, следует запустить рекурсивный поиск по каждой из таких точек, и если поиск вернет непустой маршрут, необходимо дописать его к составляемому маршруту и вернуться из функции, передав составленный маршрут в качестве возвращаемого значения.
6. Отсутствие заикливания обеспечивается тем, что в процессе поиска маршрута алгоритм оставляет за собой непроходимые "следы" в виде символов "_".

Для выполнения работы необходимо выполнить следующее:

1. Подготовить основу приложения

1.1. Создать новый проект консольного приложения.

1.2. Прочитать из текстового файла массив, содержащий описание лабиринта, и сохранить его в виде двумерного массива.

В языке C/C++ можно объявить массив "с запасом", например `char field[100][100]`, а также объявить две переменные, хранящие актуальное количество строк и столбцов.

1.3. Определить функцию `printField`, которая выводит считанный массив на экран.

1.4. Собрать и запустить приложение. Убедиться, что исходные данные считываются правильно.

2. Написать алгоритм поиска.

- 2.1. Объявить, при необходимости, новый тип данных, хранящий маршрут в виде последовательности точек.
- 2.2. Объявить функцию `findPath`, аргументом которой является точка (координаты в массиве), а возвращаемым значением является маршрут.
- 2.3. В функции `findPath` следует установить текущую точку на переданную в качестве аргумента точку, добавить ее к маршруту и отметить соответствующие координаты в массиве символом "_".
- 2.4. Если точка, переданная в качестве аргумента, является конечной точкой маршрута (выходом из лабиринта), следует вернуться из функции, передав составленный маршрут из 1 элемента в качестве возвращаемого значения.
- 2.5. В случае, если эта точка является стартовой, то есть расположена на левой границе массива, следует переместить текущую точку вправо на 1 и добавить ее к маршруту, аналогично пункту 2.3.
- 2.6. Там же в функции `findPath` следует объявить вечный цикл. В начале цикла необходимо посчитать число свободных клеток рядом с текущей точкой. Если оно равно 0, то это означает, что поиск зашел в тупик. Следует очистить накопленный маршрут и вернуть из функции пустой маршрут.
- 2.7. Если количество свободных клеток по соседству равно 1, то необходимо переместить текущую точку на свободную соседнюю клетку, добавить эту клетку к маршруту, аналогично пункту 2.3.
- 2.8. Если точка, на которую перешел алгоритм в пункте 2.7, является выходом из лабиринта, следует вернуться из функции, передав составленный маршрут в качестве возвращаемого значения. В противном случае следует перейти к началу следующей итерации цикла.
- 2.9. Если количество свободных клеток по соседству более 1, необходимо для каждой соседней клетки запустить алгоритм поиска, вызвав функцию `findPath` с соответствующим аргументом.
- 2.10. Если результатом рекурсивного поиска является непустой маршрут,

следует добавить его элементы к составляемому маршруту и вернуться из функции, передав составленный маршрут в качестве возвращаемого значения.

2.11. Если все рекурсивные вызовы вернули пустой маршрут, следует очистить составляемый маршрут и также вернуть пустой маршрут.

2.12. Вызвать составленную функцию `findPath`, передав ей в качестве аргумента входную точку лабиринта.

2.13. Заменить в массиве все символы "_" на пробелы, а затем все точки, входящие в маршрут, заменить на символ ".". Вывести полученный массив на экран.

2.14. Запустить полученное приложение и убедиться, что алгоритм правильно находит выход из лабиринта. Результатом работы программы должен стать лабиринт, отображенный на экране, на который нанесен маршрут выхода из лабиринта.

Примечание: рекомендуется использовать функцию `printField` для отладки функции, чтобы можно было проследить за ходом работы алгоритма.

3. ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ. Модифицировать алгоритм для улучшения эффективности его работы.

3.1. Исправить функцию `findPath` таким образом, чтобы при отсутствии маршрута функция заменяла все расставленные ей символы "_" обратно на пробелы.

3.2. Запустить полученное приложение и убедиться, что функция теперь находит кратчайший маршрут вместо первого попавшегося.

3.3. Предусмотреть в функции корректную обработку "комнат". Комнатой является проходима прямоугольная область размером больше чем 1x1. Алгоритм должен уметь определять такие "комнаты" и запускать рекурсивный расчет сразу по выходам из комнаты.

Вопросы для самоконтроля и подготовке к защите работы №6

1. Что такое рекурсия?

2. Чем отличается рекурсивная ветвь от терминальной?
3. С чем связана ошибка переполнения стека (Stack Overflow)?
4. Что такое глубина рекурсии?
5. Какой алгоритм расчета факториала на ваш взгляд эффективнее: итерационный (с помощью цикла) или рекурсивный?