Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления (ИУ)

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии (ИУ7)

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №4
«Работа со стеком»

Студент, группа **Дегтярев А., ИУ7-33Б**

2020 г.

Описание условия задачи

Разработать программу работы со стеком, реализующую операции добавления и удаления элементов из стека и отображения текущего состояния стека. Реализовать стек: а) массивом; б) списком.

Все стандартные операции со стеком должны быть оформлены отдельными подпрограммами. В случае реализации стека в виде списка при отображении текущего состояния стека предусмотреть возможность просмотра адресов элементов стека и создания дополнительного собственного списка свободных областей (адресов освобождаемой памяти при удалении элемента, который можно реализовать как списком, так и массивом) с выводом его на экран. Список свободных областей необходим для того, чтобы проследить, каким образом происходит выделение памяти менеджером памяти при запросах на нее и убедиться в возникновении или отсутствии фрагментации памяти.

Используя операции со стеком реализовать поиск пути в лабиринте, который представлен матрицей.

Техническое задание

Формат матрицы лабиринта:

- первая строка содержит 2 числа — количество строк и столбцов матрицы
- следующие строки представляют собой значения матрицы-лабиринта. Содержат только следующие символы:
  - «#» - стена
  - « »(пробел) - проход
  - «A» - начало лабиринта
  - «B» - конец лабиринта

Матрица должна иметь только 1 начало и конец.

Входные данные

Имя файл, из которого можно считать матрицу лабиринта. Если не представлено, матрица считывается из стандартного ввода консоли.

Выходные данные

Построчно записанный список координаты кратчайшего пути. Координаты x и y разделены пробелом. Путь содержит промежуточные пункты и конец лабиринта.

По требованию выводится матрица лабиринта с выделенным на ней путём.

Функция программы

Программа позволяет находить кратчайший путь в лабиринте, замерять время работы стека, исследовать использование памяти.

Обращение к программе

В папке с исходным кодом в консоли:

1. скомпилировать программу: *make*
2. вызвать программу для просмотра справки: ./app.exe -?

Аварийные ситуации

- Неправильный выбор команды
- Неправильный ввод матрицы
- Невозможно найти путь

Структуры данных

```
typedef struct {
    void **buf;
    void **end;
    size_t size;
    bool limit;
    memwatch_t *mem;
} stack_array_t;
```

stack_array_t — стек, реализованный массивом.

Поля структуры:

- *buf* — массив указателей на данные, которые были добавлены в стек
- *end* — указатель на элемент, следующий за последним добавленным указателем
- *size* — размер массива *buf*
- *limit* — ограничивать ли размер массива
- *mem* — указатель на наблюдателя памяти.

Эффективность структуры по памяти: *O(size)*, где *size* — наибольшая степень 2-ки, которая не меньше количества элементов массива.

```
typedef struct list_s list_t;

struct list_s {
    list_t *next;
    void *data;
};

typedef struct {
    list_t *head;
    size_t size;
    size_t n_elems;
    bool limit;
    memwatch_t *mem;
} stack_list_t;
```

list_t — структура списка.

stack_list_t — стек, реализованный списком.

Поля структуры:

- *head* — указатель на первый элемент списка
- *size* — максимальное количество элементов списка
- *n_elems* — текущее количество элементов списка
- *limit* — ограничивать ли максимальным количеством элементов списка
- *mem* - указатель на наблюдателя памяти

Эффективность структуры по памяти: *O(n)*, где *n* — количество элементов

```c
typedef struct {
    bool *walls;
    int width;
    int height;
    int start_x;
    int start_y;
    int finish_x;
    int finish_y;
} map_t;
```

map_t — структура лабиринта.

- *walls* — матрица стен
- *width* — количество столбцов
- *height* — количество строк
- *start_x, start_y* — координаты начальной позиции
- *finish_x, finish_y* — координаты конечной позиции

```c
#define MEMORY_BATCH_SIZE 128

typedef struct membatch_s membatch_t;

struct membatch_s {
    size_t start;
    char pointers[MEMORY_BATCH_SIZE];
    membatch_t *next;
};

typedef struct {
    membatch_t *head;
} memwatch_t;
```

memwatch_t — структура наблюдателя за памятью.

- head — указатель на список групп указателей памяти

membatch_t — структура списка групп указателей памяти

- start — первый адрес в списке
- pointers — информация о статусе адресов памяти
- next — следующая группа указателей памяти

Алгоритм

Сначала происходит чтение лабиринта из файла. Затем применяется алгоритм Дейкстры для поиска пути.

1. Добавляем стартовую позицию в стек
2. Пока стек не пустой:
    1. Взять позицию из стека
    2. Если в этой позиции нет стены и эта позиция раньше не посещалась или в эту позиции ведёт более длинный путь, то добавить соседние позиции в стек.
3. Добавить в конец пути конечную позицию
4. Пока не пришли в начальную позицию:
    1. Добавить в путь наиболее близкую к старту позицию среди соседних

Эффективность по времени: *O(w * h);* эффективность по памяти — *O(w * h);* где *w* — количество столбцов в матрице, *h* — количество строк в матрице.

Тесты

Группа	Входные данные	Выходные данные
Аргументы	отсутствующий аргумент	Неправильный ввод
Ввод матрицы	Не положительный размер матрицы	Неправильный ввод
	Буква в размере матрицы	Неправильный ввод
	Неправильный формат матрицы	Неправильный ввод
Путь	Финишная точка заблокирована стеной	Нет пути

Оценка эффективности

Количество Элементов	Время добавления элементов в стек-массив, мкс	Время удаления элементов из стека-массива, мкс	Время добавления элементов в стек-список, мкс	Время удаления элементов из стека-списка, мкс
10	1	1	1	1
50	1	1	2	2
100	1	1	3	3
500	4	3	10	10
1000	7	5	20	19
5000	28	23	92	87
10000	59	48	186	174
50000	262	216	930	872
100000	560	521	1890	1774
500000	2662	2235	9805	9051
1000000	5325	4492	19456	17953

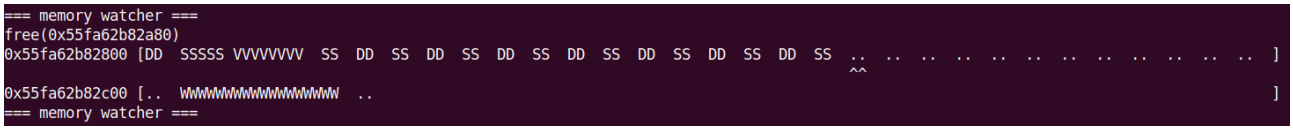
Количество элементов	Память стека-массива, в байтах	Память стека-списка в байтах
10	128	160
50	512	800
100	1024	1600
500	4096	8000
1000	8192	16000
5000	65536	80000
10000	131072	160000
50000	524288	800000
100000	1048576	1600000
500000	4194304	8000000
1000000	8388608	16000000

Поиск фрагментации производится с помощью наблюдателя за памятью. Вместо вызова обычных функций работы с динамической памятью (malloc) используются функции наблюдателя памяти (wmalloc), которые вместе с вызовом стандартной функции выводят в консоль аргументы и адреса занимаемой памяти.

Пример вывода для стека-массива:

```
=== memory watcher ===
malloc(12) -> 0x55fc49214b50
0x55fc49214800 [DD  SSSSS ....  VVVVVVVV  DD  DD  DD  DD  ........  DD  DD  DD  DD  SSSSSSSSSSSSSSS  DD  DD  DD  DD  DD  DD                ]
                                                                                                                    ^^
=== memory watcher ===
```

Пример вывода для стека-списка:



Первая и последняя строчки выделяют то, что между ними находится сообщение от наблюдателя памяти.

Вторая строчка — вызов стандартной функции, ее аргументы и результат.

Третья и последующие — представление в памяти. Первое число — указатель на начало блока отображаемой памяти. В квадратных скобках — блок памяти. Один символ представляет 8 байт. Символы значат следующее:

- « »(пробел) — память не используется
- «.» - память использовалась ранее, но была освобождена
- «D» - данные
- «S» - стек
- «V» - карта, посещенных позиций
- «P» - массив позиций пути
- «W» - память, занимаемая наблюдателем

Проследив за работой с памятью, я сделал вывод, что реализация стека расширяемым массивом может привести к фрагментации.

Во время расширения массива выделяется двойная память. Адреса памяти после массива точно заняты данными. В адресах перед массивом есть свободные участки памяти от прошлых реаллокаций, которые меньше чем необходимая память. Поэтому массив переместится за адреса данных, оставив еще одно свободное место в памяти. Это и есть фрагментация.

У списков такой картины фрагментации не было замечено. Элементы списка выделяются и освобождаются в памяти последовательно.

Функции

```
stack_array_t *create_stack_array(size_t size, memwatch_t *mem);
      Создаёт стек на массиве указанного размера. При size == 0, массив динами-
чески расширятся.

void free_stack_array(stack_array_t *stack);
      Освобождает выделенную на стек память.

int stack_array_push(stack_array_t *stack, void *data);
      Добавляет элемент в стек на массиве.

void *stack_array_pop(stack_array_t *stack);
      Считывает элемент из стека на массиве.
```

```
stack_list_t *create_stack_list(size_t size, memwatch_t *mem);
      Создаёт стек на списке. Ограничивает количество элементов списка до size.
При size = 0 ограничения нет.

void free_stack_list(stack_list_t *stack);
      Освобождает выделенную на стек память.

int stack_list_push(stack_list_t *stack, void *data);
      Добавляет элемент в стек на списке.

void *stack_list_pop(stack_list_t *stack);
      Считывает элемент из стека на списке.

int get_path_using_stack_list(const map_t *map, stack_list_t *stack, point_t
**result_path);
      Ищет путь в лабиринте, используя стек на списке. Результат записывает в
result_path.

int get_path_using_stack_array(const map_t *map, stack_array_t *stack, point_t
**result_path);
      Ищет путь в лабиринте, используя стек на массиве. Результат записывает в
result_path.

map_t *create_map(int width, int height);
      Создаёт лабиринт размером width и height.

void free_map(map_t *m);
      Освобождает выделенную на лабиринт память.

map_t *read_map_from_file(const char *filename);
      Считывает лабиринт из файла
void print_map_with_path(FILE *f, const map_t *map, const point_t *path, int
path_len);
      Выводит в поток лабиринтом с построенным на нем путём

memwatch_t *create_memory_watch(void);
      Создаёт наблюдателя за памятью

void free_memory_watch(memwatch_t *w);
      Освобождает память наблюдателя.

void *wmalloc(memwatch_t *w, char type, size_t size);
      malloc с выводом адресов

void *wrealloc(memwatch_t *w, void *ptr, size_t new_size);
      realloc с выводом адресов

void wfree(memwatch_t *w, void *ptr);
      free с выводом адресов
```

Вывод

Стек на списке использует в 2 раза больше памяти чем на массиве, так как на каждый элемент данных используется указатель на следующую структуру списка. Работа стека на массиве быстрее, примерно, в 4 раза, потому что обычно для добавления и чтения элемента нужно только сдвинуть указатель. Таким образом, стек на массиве эффективнее стека на списке как по памяти, так и по времени, однако приводит к фрагментации, что необходимо учитывать на устройствах с ограниченной памятью.

Контрольные вопросы

Что такое стек?

Стек – структура данных, в которой можно обрабатывать только последний добавленный элемент (верхний элемент). На стек действует правило LIFO — последним пришел, первым вышел.

Каким образом и сколько памяти выделяется под хранение стека при различной его реализации?

При хранении стека с помощью списка, то память всегда выделяется в куче. При хранении с помощью массива, память выделяется либо в куче, либо на стеке (в зависимости от того, динамический массив или статический). Для каждого элемента стека, реализованного списком, выделяется на 1 указатель больше, чем для элемента массива. Этот указатель - указатель на следующий элемент списка.

Каким образом освобождается память при удалении элемента стека при различной реализации стека?

При хранении стека связанным списком, верхний элемент удаляется путем освобождением памяти для него и смещения указателя, указывающего на начало стека. При удалении из стека, реализованного массивом, смещается лишь указатель на вершину стека.

Что происходит с элементами стека при его просмотре?

Элементы стека уничтожаются, так как каждый раз достается верхний элемент стека.

Каким образом эффективнее реализовывать стек? От чего это зависит?

Реализовывать стек эффективнее с помощью массива. Он выигрывает как во времени обработки, так и в количестве занимаемой памяти.