

Лабораторная работа 7.

Черновик

Целью работы является знакомство студентов с использованием динамической памяти (выделение и освобождение памяти в «куче», использование специализированного ПО для отладки использования памяти).

Студенты должны получить и закрепить на практике следующие знания и умения:

1. Выделение и освобождение динамической памяти.
2. Использование адресной арифметики для обработки одномерных массивов.
3. Использование указателей на функцию.
4. Использование указателей типа void.
5. Обработка текстовых файлов.
6. Организация корректной работы с ресурсами (динамически выделенная память, файловые описатели).
7. Использование в программе аргументов командной строки.

1. Общее задание

1. Исходный код лабораторной работы располагается в отдельной ветке lab_07. В ветке lab_07 создается папка lab_07_X_Y, где вместо X указывается номер алгоритма сортировки, вместо Y – номер функции-фильтра. Варианты заданий распределяются преподавателем.
2. Исходный код должен соответствовать правилам оформления исходного кода.
3. Решение задачи оформляется как многофайловый проект.
4. Для задачи студентом подготавливаются тестовые данные, которые демонстрируют правильность ее работы (функциональные тесты). Входные данные должны располагаться в файлах in_z.txt, выходные out_z.txt, где z – номер тестового случая. Тестовые данные готовятся и помещаются под версионный контроль еще до того, как появится реализация задачи.
5. При решении задач этой лабораторной работы в методических целях нельзя использовать выражение вида $a[i]$ и вообще квадратные скобки. Вместо указанного выражения используется выражение $*ra$, где ra - указатель на i -ый элемент массива (именно на i -ый элемент, а не выражение вида $*(ra + i)$). Также нельзя передавать как аргумент размер массива в элементах, если массив уже создан. Вместо этого предлагается использовать пару указателей: на первый элемент массива и на элемент массива, расположенный за последним. Ситуация когда эти указатели совпадают, означает пустоту обрабатываемого массива.
6. Ответы на теоретические вопросы, связанные с защитой лабораторной работы, приводятся в отчете к лабораторной работе. Ответы пишутся «своими словами», при этом запрещается любое копирование ответов откуда бы ни было. Вопросы распределяются преподавателем (преподаватель волен варьировать количество вопросов, которых должно быть не менее трех). Для написания отчета используется wiki в GitLab.

2. Индивидуальное задание

Написать программу, которая упорядочивает (сортирует) массив.

Данные в массив считываются из текстового файла. Количество элементов в файле не указано. Память под массив выделяется динамически. Число элементов в массиве

определяется в первом проходе по текстовому файлу, во время второго прохода числа считываются в массив.

Полученный после сортировки массив записывается в другой файл.

Возможно, что перед сортировкой элементы массива могут быть отфильтрованы с помощью функции-фильтра.

Функция-фильтр работает следующим образом:

- определяет количество элементов массива, которые удовлетворяют заданному условию;
- выделяет память под соответствующее количество элементов;
- копирует элементы, удовлетворяющие условию, из старого массива в новый.

Функция сортировки реализуется универсальной (т.е. имеет одинаковый прототип с функцией `qsort` из стандартной библиотеки (`stdlib.h`)).

Все параметры (имена файлов, необходимость фильтрации) передаются через аргументы командной строки.

Для функции сортировки и функции-фильтра реализуются модульные тесты (отдельный проект).

При защите лабораторной работы необходимо продемонстрировать отчет утилиты Doctor Memory, свидетельствующий об отсутствии ошибок при работе с памятью.

Кроме того, необходимо сравнить время работы реализованного алгоритма сортировки и `qsort`. Постараться найти примеры, когда ваша реализация лучше. Результаты представить в виде графиков (рисунки в формате PNG, оси подписаны). Графики разместить в Wiki на отдельной странице. Графики можно строить любыми средствами (например, в Excel).

Алгоритмы сортировки (варианты):

1. Сортировка выбором: находится максимальный элемент массива и переносится в его конец; затем этот метод применяется ко всем элементам массива, кроме последнего (т.к. он уже находится на своем месте), и т.д.
2. Сортировка обменом (метод пузырька): последовательно сравниваются пары соседних элементов $x[k]$ и $x[k+1]$ ($k = 0, 1, \dots, n-2$) и, если $x[k] > x[k+1]$, то они переставляются; в результате наибольший элемент окажется на своем месте в конце массива; затем этот метод применяется ко всем элементам, кроме последнего, и т.д.
3. Сортировка вставками: пусть первые k элементов массива (от 0 до $k-1$) уже упорядочены по неубыванию; тогда берется $x[k]$ и размещается среди первых k элементов так, чтобы упорядоченными оказались уже $k+1$ первых элементов; этот метод повторяется при k от 1 до $n-1$.
4. Модифицированная сортировка вставками. Для поиска места вставки нового элемента используется двоичный поиск.
5. Модифицированная сортировка пузырьком I. Запоминайте, где произошел последний обмен элементов, и при следующем проходе алгоритм не заходит за это место. Если последними поменялись i -ый и $i+1$ -ый элементы, то при следующем проходе алгоритм не сравнивает элементы за i -м.
6. Модифицированная сортировка пузырьком II. Нечетные и четные проходы выполняются в противоположных направлениях: нечетные проходы от начала к концу, четные – от

конца массива к его началу. При нечетных проходах большие элементы сдвигаются к концу массива, а при четных проходах – меньшие элементы двигаются к его началу.

7. Модифицированная сортировка пузырьком III. Идеи первой и второй модифицированной сортировки пузырьком объединяются.

Функция-фильтр (варианты):

1. В массиве остаются элементы расположенные между минимальным и максимальным элементами массива.
2. В массиве остаются элементы от нулевого до m-го, где m - индекс первого отрицательного элемента этого массива либо число n-1, если такого элемента в массиве нет.
3. В массиве остаются элементы от нулевого до p-го, где p - индекс последнего отрицательного элемента этого массива либо число n-1, если такого элемента в массиве нет.
4. В массиве остаются элементы, которые больше среднего арифметического всех элементов массива.
5. В массиве остаются элементы, которые больше суммы элементов расположенных за ним.

3. Замер времени

Практически каждый процессор имеет специальный встроенный регистр - счетчик тактов, значение которого можно получить специальной командой процессора. Команда процессора RDTSC (Read Time Stamp Counter) возвращает в регистрах EDX и EAX 64-разрядное беззнаковое целое, равное числу тактов с момента запуска процессора. Вызвав эту команду до и после участка программы, для которого требуется вычислить время исполнения, можно вычислить разность показаний счетчика. Оно равно числу тактов, затраченных на исполнение замераемого участка. Для перехода от числа тактов к времени требуется умножить число тактов на время одного такта (величина, обратная тактовой частоте процессора). Для процессора с тактовой частотой 1ГГц время такта - 1 нс.

Достоинством этого способа является максимально возможная точность измерения времени, недостатком - команда получения числа тактов зависит от архитектуры процессора.

Недостатки способа и подходы к борьбе с ними описаны в работе «Использование Time-Stamp Counter для измерения времени выполнения кода на процессорах с архитектурой Intel 64 и IA-32» Михаила Курносова.

```
#include <stdio.h>

unsigned long long tick(void)
{
    unsigned long long d;

    __asm__ __volatile__ ("rdtsc" : "=A" (d) );

    return d;
}

void test(void)
{
    long double test = 0.0;

    for(unsigned long long i = 0; i < 10000; i++)
        test += 0.5;
}
```

```

#define N 5

int main(void)
{
    unsigned long long tb, te;

    tb = tick();
    for(int i = 0; i < N; i++)
        test();
    te = tick();

    printf("test 'time': %llu\n", (te - tb) / N);

    return 0;
}

```

4. Сравнение строк (для обработки параметров командной строки)

Необходимость выполнения фильтрации указывается пользователем. Будем ее выполнять только в том случае если первый параметр командной строки (`argv[1]`) при запуске приложения равен символу `"-"`. (Такое соглашение выбрано для примера. Вы можете придумать свое собственное.)

Способ 1

Строка в Си представляет собой массив типа `char`, который заканчивается символом с кодом 0.

```

char* pstr;
...
pstr = argv[1];
if(pstr[0] == '-' && pstr[1] == 0)
    ...

```

Способ 2

С помощью библиотечной функции `strcmp` из `string.h`. Функция возвращает значение 0, если переданные ей строки равны.

```

if (strcmp(argv[1], "-") == 0)
    ...

```