



**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ**
Брянский государственный технический университет

Д.И. Булатицкий, Е.В. Коптенок, Р.А. Исаев, А.О. Радченко

**ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ СИ:
УПРАВЛЕНИЕ ПАМЯТЬЮ, ГРАФИКА
И ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ**

Утверждено редакционно-издательским советом университета
в качестве практикума

Брянск
ИЗДАТЕЛЬСТВО БГТУ
2018

Булатицкий, Д.И. Программирование на языке Си: Управление памятью, графика и динамические структуры данных [Текст] + [Электронный ресурс]: практикум / Д.И. Булатицкий, Е.В. Коптенок, Р.А. Исаев, А.О. Радченко – Брянск: БГТУ, 2018. – 148 с.

ISBN 978-5-907111-63-9

Приводится описание семнадцати лабораторных работ по дисциплине «Программирование».

Практикум предназначен для студентов очной формы обучения по направлениям подготовки: 09.03.01 «Информатика и вычислительная техника», 09.03.04 – «Программная инженерия», 02.03.03 «Математическое обеспечение и администрирование информационных систем».

Ил. 48. Табл. 6. Библиогр. – 22 назв.

Научный редактор А.А.Азарченков

Рецензенты: кафедра математики, информационных технологий и информационного права Брянского филиала ФГБОУ ВО «Российская академия народного хозяйства и государственной службы при Президенте Российской Федерации»;
кандидат технических наук Б.Н.Прусс

ISBN 978-5-907111-63-9

© Брянский государственный
технический университет, 2018

ПРЕДИСЛОВИЕ

Практикум предназначен для студентов очной формы обучения по направлениям подготовки: 09.03.01 «Информатика и вычислительная техника», 09.03.04 «Программная инженерия», 02.03.03 «Математическое обеспечение и администрирование информационных систем» для изучения во втором семестре второй части дисциплины «Программирование». Он также может использоваться студентами других форм обучения и родственных направлений подготовки.

Практикум способствует более глубокому изучению дисциплины «Программирование»: позволяет установить и закрепить связь между теорией программирования и практикой решения задач с помощью ЭВМ, развивает у студентов навыки разработки программного обеспечения.

В практикуме приводится описание семнадцати лабораторных работ. В каждом описании лабораторной работы содержатся краткие теоретические сведения, необходимые для выполнения лабораторной работы; подробное пошаговое иллюстрированное описание действий, общих для всех студентов; индивидуальные задачи; задачи для самостоятельного решения и контрольные вопросы.

Лабораторные работы для удобства сгруппированы по темам в главы. В каждой следующей главе (и даже каждой лабораторной работе) используется понятийный аппарат и инструментарий, изученный в предыдущих лабораторных работах, поэтому студентам рекомендуется выполнять лабораторные работы в том порядке, в каком они приведены в практикуме.

В конце практикума приведен список литературы, который может оказать помощь студентам в изучении дисциплины «Программирование».

Рекомендуется следующий порядок выполнения лабораторной работы. Сначала рекомендуется изучить теоретические сведения по теме лабораторной работы по учебникам, конспекту лекций и параграфу «Краткие теоретические сведения», затем выполнить по шагам действия, описанные в параграфе «Общая часть лабораторной работы». После этого необходимо решить задачи индивидуальной части лабораторной работы и задачи для самостоятельного решения по своему варианту. Если при решении задач своего варианта студент испытывает затруднения,

то ему рекомендуется дополнительно прорешать задачи других вариантов. И, наконец, студенту предлагается ответить на контрольные вопросы. Если при ответе на какие-то контрольные вопросы студент испытывает затруднения, он может обратиться к рекомендуемой литературе. Если даже с помощью рекомендуемой литературы не удалось ответить на них, студенту рекомендуется обратиться за помощью к своим одногруппникам или к преподавателю. Для успешного завершения выполнения лабораторной работы студенту необходимо защитить её результаты. При защите лабораторной работы студент показывает все разработанные программы, выполняет их тестовые запуски, даёт обоснование правильности получаемых программой результатов, а также отвечает на вопросы преподавателя.

Основой для выбора варианта задачи для самостоятельного решения является номер студента в списке группы. Если другое не определено непосредственно в задаче, то номер варианта выбирается следующим образом. Вычисляется остаток от целочисленного деления номера студента в списке группы на количество вариантов. Если этот остаток отличен от нуля, то он и принимается в качестве номера варианта. Иначе выбирается наибольший номер варианта.

Практикум подготовлен коллективом авторов. За общую координацию действий авторов, разработку тематики лабораторных работ и последовательность их подачи отвечал Д.И. Булатицкий. Подбором теоретического материала занимались совместно Д.И. Булатицкий и Е.В. Коптенюк. Е.В. Коптенюк отвечала за подбор большинства примеров задач, описание решения этих задач и компьютерный набор. Особой заслугой Р.А. Исаева является разработка вариантов задач для самостоятельного решения. А.О. Радченко отвечал за подготовку многих примеров программ, экранных снимков их работы, листингов.

ГЛАВА 1. УКАЗАТЕЛИ. ГРАФИКА И АНИМАЦИЯ

Лабораторная работа №1. Работа с указателями

1. Цель лабораторной работы

Цель лабораторной работы – научиться использовать указатели для решения задач.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Указатель – переменная, содержащая адрес объекта. Указатель не несет информации о содержимом объекта, а содержит сведения о том, где размещен объект. Указатели широко используются в программировании на языке Си.

Память компьютера можно представить в виде последовательности пронумерованных однобайтовых ячеек, с которыми можно работать по отдельности или блоками.

Каждая переменная в памяти имеет свой адрес – номер ячейки, где она расположена, а также свое значение. Указатель – это тоже переменная, которая размещается в памяти [2].

Указатель, как и любая переменная, должен быть объявлен.

Общая форма объявления указателя

```
тип *имя_объекта;
```

Тип указателя – это тип переменной, адрес которой он может содержать.

При описании указателей в качестве имени типа можно использовать ключевое слово `void`, например:

```
void *vd;
```

При таком описании с указателем не связывается никакой тип данных, то есть получаем указатель на произвольный тип данных.

Для указателей одного и того же типа допустимой является операция присваивания, кроме того, указателю типа `void` может быть присвоено значение адреса любого типа данных, но не наоборот, например:

```

int *a, *b;
double *d;
void *v;
...
a = b; /* Правильно */
v = a; /* Правильно */
v = d; /* Правильно */
b = v; /* Неправильно */
d = a; /* Неправильно */

```

При неправильном присваивании указателей компиляторы обычно выдают предупреждающие сообщения, которыми никогда не следует пренебрегать. Например, компилятор фирмы Borland выдает сообщение «Suspicious pointer conversion», которое переводится как «Подозрительное преобразование указателей» [2].

Если по какой-либо причине необходимо выполнить операцию присваивания между указателями разного типа, то используют явное преобразование типов, например для указателей из предыдущего примера можно сделать так:

```

b = (int *) v;
d = (double *) a;

```

Арифметика указателей

Для работы с указателями в Си определены две операции [3]:

- операция * (звездочка) – позволяет получить значение объекта по его адресу – определяет значение переменной, которое содержится по адресу, содержащемуся в указателе;
- операция & (амперсанд) – позволяет определить адрес переменной.

Определим адрес однобайтовой переменной:

```

char c; // переменная
char *p; // указатель
p = &c; // p = адрес c

```

Для рассмотренного примера обращение к одним и тем же значениям переменной и адреса представлено в табл. 1.

Указателю необходим тип, чтобы корректно работала операция **разыменования** (получения содержимого по адресу). Если указатель хранит адрес переменной, то требуется знать, сколько байт необходимо взять, начиная от этого адреса, чтобы получить всю переменную [3].

Таблица 1

Обращение к значениям переменной и адреса

	Переменная	Указатель
Адрес	&c	p
Значение	c	*p

Указатели поддерживают арифметические операции. Для их выполнения необходимо знать размер данных, на которые ссылается указатель.

Операция $+N$ сдвигает указатель вперёд на $N * \text{sizeof}(\text{тип})$ байт.

Так, если указатель `int *p`; хранит адрес `CC02`, то после `p += 10`; он будет хранить адрес `CC02 + sizeof(int)*10 = CC02 + 28 = CC2A` (Все операции выполняются в шестнадцатиричном формате).

Пусть создан указатель на начало массива. После этого можно «двигаться» по этому массиву, получая доступ к отдельным его элементам.

Пример

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

void main() {
    int A[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int *p;
    p = A;

    printf("%d\n", *p);
    p++;
    printf("%d\n", *p);
    p = p + 4;
    printf("%d\n", *p);
    getch();
}
```

Массив, по сути, сам является указателем, поэтому не требуется использовать оператор `&`. Можно переписать пример по-другому:

```
p = &A[0]; //Получить адрес первого элемента и
           //относительно него двигаться по массиву
```

Кроме операций сложения и вычитания указатели поддерживают операции сравнения. Если у нас есть два указателя `a` и `b`, то `a > b`, если адрес, который хранит `a`, больше адреса, который хранит `b`.

Следующий листинг иллюстрирует применение операций сравнения:

```
void main() {
    int A[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int *a, *b;

    a = &A[0];
    b = &A[9];

    printf("&A[0] == %p\n", a);
    printf("&A[9] == %p\n", b);

    if (a < b) {
        printf("a < b");
    } else {
        printf("b > a");
    }

    getch();
}
```

Если же указатели равны, то они указывают на одну и ту же область памяти.

3. Общая часть лабораторной работы

Задача 1. Создайте проект, содержащий листинг, представленный на рис. 1. Запустите программу.

Решение. Обратите внимание на выведенные программой сообщения. Объясните их. Почему адрес расположения указателя `b` и значение указателя `b` не равны?


```

#include <stdio.h>
#include <stdlib.h>
int main() {
    int a, *b;
    a=134;
    b=&a;
    printf("\n Значение переменной a равно %d.", a);
    printf("\n Адрес переменной a равен %d.", &a);
    printf("\n Данные по адресу указателя b равны %d.", *b);
    printf("\n Значение указателя b равно %d.", b);
    printf("\n Адрес расположения указателя b равен %d.", &b);
    getchar();
    return 0;
}

```

Рис. 1. Листинг программы к Задаче 1

Задача 2. Создайте проект, содержащий листинг, представленный на рис. 2. Запустите программу.

```

#include <conio.h>
#include <stdio.h>

void main() {
    int A[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21};
    int *p;

    p = A;

    printf("%d\n", *p);
    p++;
    printf("%d\n", *p);
    p = p + 4;
    printf("%d\n", *p);

    getch();
}

```

Рис. 2. Листинг программы к Задаче 2

Измените код таким образом, чтобы на экран выводилось сначала значение первого элемента массива, а затем значения $1+N$ и $1+(2N\%21)$ элементов, где N – номер варианта.

Задача 3. Разработать функцию обмена значений двух переменных.

Решение. Обычно функция возвращает одно значение, но иногда требуется вернуть больше одного. Рассмотрим представленный

на рис. 3 код функции, которая меняет местами значения двух переменных.

```
int swap (double a, double b) {
    double temp = a;
    a = b;
    b = temp;
}
```

Рис. 3. Листинг функции

Пусть есть переменные *x* и *y* с некоторыми значениями. Если выполнить функцию, передав в нее *x* и *y*, окажется, что никакого обмена не произошло. И это правильно.

При вызове этой функции в стеке будут сохранены значения *x* и *y*. Далее *a* и *b* получают значения *x* и *y*. Будет выполнена перестановка. Затем функция завершится и значения *x* и *y* будут восстановлены из стека.

Чтобы заставить функцию работать так, как необходимо, в нее передают не значения переменных *x* и *y*, а их адреса. Однако и саму функцию тогда необходимо адаптировать для работы с адресами (рис. 4).

```
int swap (double *a, double *b) {
    double temp = *a;
    *a = *b;
    *b = temp;
}
```

Рис. 4. Листинг функции обмена переменных

Не стоит забывать о том, что и вызов функции теперь должен выглядеть иначе (рис. 5):

```
swap(&x, &y);
```

Теперь в функцию передаются адреса, обращение к данным ведется относительно переданных адресов.

Если функция должна вернуть несколько значений, необходимо передавать в нее адреса переменных, куда она должна записать эти значения.

Если функция должна менять значение переменной, необходимо передавать ей адрес этой переменной.

Задача 4. Создайте проект, содержащий листинг, представленный на рис. 6. Запустите программу. Программа должна вывести на экран значение переменной *x*.

Определите, какие ошибки были допущены в коде, и исправьте их.

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

void main () {
    int x, *p;
    x = 10;
    p = x;
    printf ("%d", *p);
}
```

Рис. 6. Листинг программы к задаче 4

Задача 5. Создайте проект, содержащий листинг, представленный на рис. 7. Запустите программу. Программа должна вывести на экран числа от одного до десяти.

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

int main () {
    int x, *p;
    x = 10;
    for (int i=1; i<=x; i++)
    {
        p=&i;
        *p+=-1;
        printf ("%d", *p);
    }
    return 1;
}
```

Рис. 7. Листинг программы к задаче 5

Определите, какие ошибки были допущены в коде, и исправьте их.

4. Индивидуальная часть лабораторной работы

Написать программу на языке Си для решения задачи согласно варианту ($N_{\text{вар}} = N_{\text{спис}} \% 10 + 1$). Для решения задачи использовать указатели.

5. Задачи для самостоятельного решения

1. Определить максимальное и минимальное значения из двух различных вещественных чисел.
2. Поменять местами значения трех переменных.
3. Вычислить сумму и произведение всех чисел от А до В.
4. Даны координаты трех вершин треугольника. Вычислить длины всех его сторон и площадь, если треугольник существует.
5. Вычислить сумму и произведение цифр трех чисел.
6. Найти наибольшую и наименьшую цифру в записи числа
7. Найти корни квадратного уравнения.
8. Решить систему из двух линейных уравнений (частные случаи можно не рассматривать).
9. Три введенных числа увеличить в А раз и прибавить В.
10. Упорядочить три числа по возрастанию их модулей.

6. Контрольные вопросы

1. Что такое указатель?
2. Что называется адресом переменной?
3. Какова взаимосвязь между указателем, переменной, значением переменной и адресом переменной?
4. Каким образом можно изменить в функции значения переданных ей переменных?

Лабораторная работа №2. Использование графических примитивов для создания простых изображений

1. Цель лабораторной работы

Цель лабораторной работы – получить навыки создания простейших изображений с помощью библиотеки SDL.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Simple DirectMedia Layer (SDL) – это свободная кроссплатформенная мультимедийная библиотека, реализующая единый программный интерфейс к графической подсистеме, звуковым устройствам и средствам ввода для широкого спектра платформ [4].

Создание простейшего приложения с использованием SDL можно разделить на несколько этапов.

1. Подключение библиотек.

2. Инициализация.

Для этого есть функция `int SDL_Init(Uint32 flags)`, которая принимает в себе параметр, указывающий, какую подсистему инициализировать в SDL.

```
SDL_Init( SDL_INIT_VIDEO );
```

Всего есть несколько подсистем, которые необходимо знать [4]:

- `SDL_INIT_TIMER` -> The timer subsystem -> подсистема таймера
- `SDL_INIT_AUDIO` -> The audio subsystem -> аудио
- `SDL_INIT_VIDEO` -> The video subsystem -> видео
- `SDL_INIT_CDROM` -> The cdrom subsystem -> cd-привод
- `SDL_INIT_JOYSTICK` -> The joystick subsystem -> джойстик.

Чтобы не инициализировать каждую из подсистем в отдельности, их можно объединять через операцию побитового «или» (`|`) вот так:

```
SDL_Init( SDL_INIT_VIDEO | SDL_INIT_AUDIO );
```

Если необходимо инициализировать все системы, то используют константу `SDL_INIT EVERYTHING`:

```
SDL_Init( SDL_INIT_EVERYTHING );
```

Теперь стоит сказать о том, что функция `SDL_Init()` возвращает. Если выбранные подсистемы инициализируются правильно, то функция возвращает 0, в противном случае функция возвращает -1. Это необходимо отслеживать, ведь если инициализировать SDL не удалось, то и не имеет смысла продолжать дальнейшее выполнение программы. Тогда здравый смысл диктует обернуть `SDL_Init` в конструкцию `if`, которая проверяет результат ее выполнения:

```
if(SDL_Init( SDL_INIT_EVERYTHING ) == -1)
{
    printf("SDL_Init failed: %s\n", SDL_GetError() );
}
```

Если все-таки что-то пошло не так и получено -1, то имеется возможность узнать, что же произошло. Для этого существует функция `SDL_GetError()`, которая возвращает описание ошибки в текстовой форме.

Обратная инициализации функция – это деинициализация. Выполняется она функцией `SDL_Quit(void)` она сбрасывает все инициализированные подсистемы и освобождает ресурсы, занятые ими. Она должна быть выполнена по завершении работы с SDL.

Для отрисовки окна используется функция с таким заголовком:

```
SDL_Window* SDL_CreateWindow (const char* title, int x,
int y, int w, int h, Uint32 flags)
```

Вот пример вызова этой функции:

```
window=SDL_CreateWindow("Hello",SDL_WINDOWPOS_UNDEFINED,
SDL_WINDOWPOS_UNDEFINED, SCREEN_WIDTH, SCREEN_HEIGHT,
SDL_WINDOW_SHOWN);
```

Параметры функции `SDL_CreateWindow` указаны в табл. 1. Флаги функции указаны в табл. 2.

Параметры функции SDL_CreateWindow

title	Заголовок окна
x	Позиция x окна, SDL_WINDOWPOS_CENTERED, или SDL_WINDOWPOS_UNDEFINED
y	Позиция y окна, SDL_WINDOWPOS_CENTERED, или SDL_WINDOWPOS_UNDEFINED
w	Высота окна
h	Ширина окна
flags	Флаги

Таблица 2

Флаги функции SDL_CreateWindow

SDL_WINDOW_FULLSCREEN	Полноэкранный режим
SDL_WINDOW_FULLSCREEN_DESKTOP	Окно в полноэкранном режиме с текущим разрешением рабочего стола
SDL_WINDOW_HIDDEN	Окно скрыто
SDL_WINDOW_BORDERLESS	Нет рамки
SDL_WINDOW_RESIZABLE	Возможность масштабирования окна
SDL_WINDOW_MINIMIZED	Свернутое окно
SDL_WINDOW_MAXIMIZED	Развернутое окно
SDL_WINDOW_INPUT_GRABBED	Фокус ввода в окне
SDL_WINDOW_ALLOW_HIGHDPI	Окно должно быть создано в высоком разрешении, если возможно

Функция возвращает объект типа SDL_Window:

```
SDL_Window* window
```

Для работы потребуются следующие функции:

- SDL_GetWindowSurface(window) – получение поверхности в окне;

- `SDL_FillRect(SDL_Surface* dst, const SDL_Rect* rect, Uint32 color)` – заливка области цветом.

Параметры функции `SDL_FillRect` указаны в табл. 3.

Таблица 3

Параметры функции `SDL_FillRect`

dst	Поверхность
rect	SDL_Rect структура, представляющая прямоугольник, или NULL, чтобы заполнить всю поверхность
color	Цвет для заполнения

`SDL_UpdateWindowSurface(window)` – обновить поверхность в окне;

`SDL_Delay(5000);` – задержка экрана в миллисекундах.

3. Общая часть лабораторной работы

Задача 1. Создайте новый проект, подключите библиотеку SDL2 к нему и создайте первое оконное приложение.

Решение. Для начала создадим пустой проект и создадим в нем файл *.cpp. После этого подключим библиотеку к проекту.

Рассмотренный способ подключения SDL библиотеки подходит для MS Visual Studio 15. Инструкция для DevCpp располагается по ссылке:

<https://www.youtube.com/watch?v=ALMLZiqFNKk&noredirect=1>

Для начала следует скачать библиотеку:

<http://www.libsdl.org/download-2.0.php>

или

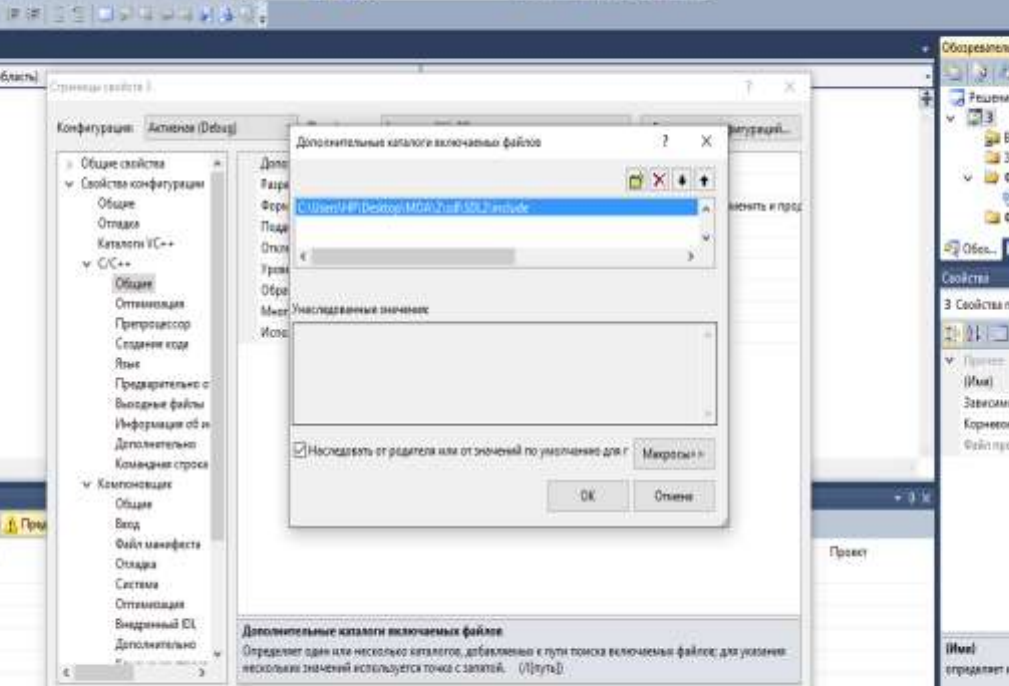
http://grafika.me/modules/pubdlcnt/pubdlcnt.php?file=http://grafika.me/files/SDL_Lesson1.zip&nid=493.

Скачанный архив необходимо распаковать.

Теперь вернемся в созданный проект.

Нажимаем правой кнопкой мыши (ПКМ) в обозревателе решений по проекту -> свойства (рис. 1).

Выбираем C/C++ -> Общие -> Дополнительные каталоги включаемых файлов. Находим папку SDL2 и заходим в папку Includes. Нажимаем ОК (рис. 2).



The screenshot shows the Microsoft Visual Studio 2017 interface. The main window displays the 'Properties' window for the 'Debug' configuration. The 'Configuration Properties' tree on the left shows 'General' selected. The 'Additional Include Directories' property is set to '\$(ProjectDir)\Src\SQL\Include'. A dialog box titled 'Additional Include Directories' is open in the foreground, showing the same path 'C:\Users\HP\Desktop\MOA\Src\SQL\Include' in the list. The 'Show Environment Variables' checkbox is checked. The 'Properties' window on the right shows the 'General' tab with the 'Additional Include Directories' property set to '\$(ProjectDir)\Src\SQL\Include'.

Рис.2. Установка дополнительных каталогов подключаемых файлов

Дальше переходим в Компоновщик -> Общие -> Дополнительные файлы библиотек (рис.3.). Сюда уже вставляем папку SDL2\lib\x86

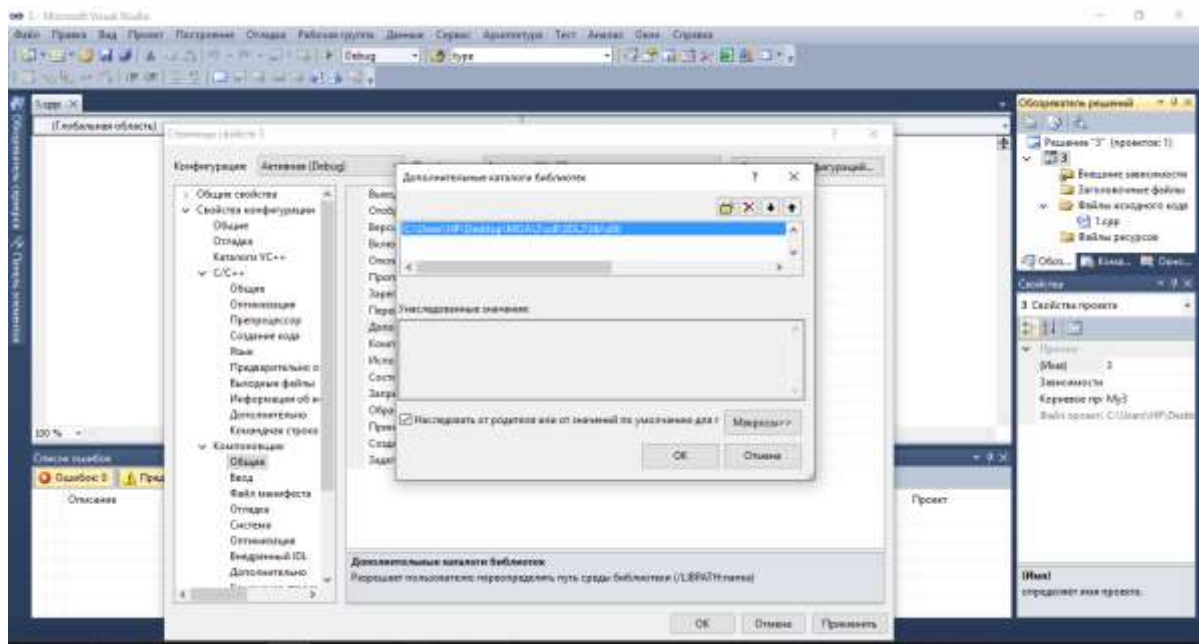


Рис.3. Окно свойств дополнительных каталогов библиотек

Заходим в Компоновщик -> Система -> Подсистема. Выбираем (если не стоит) режим «консоль» (рис. 4.).

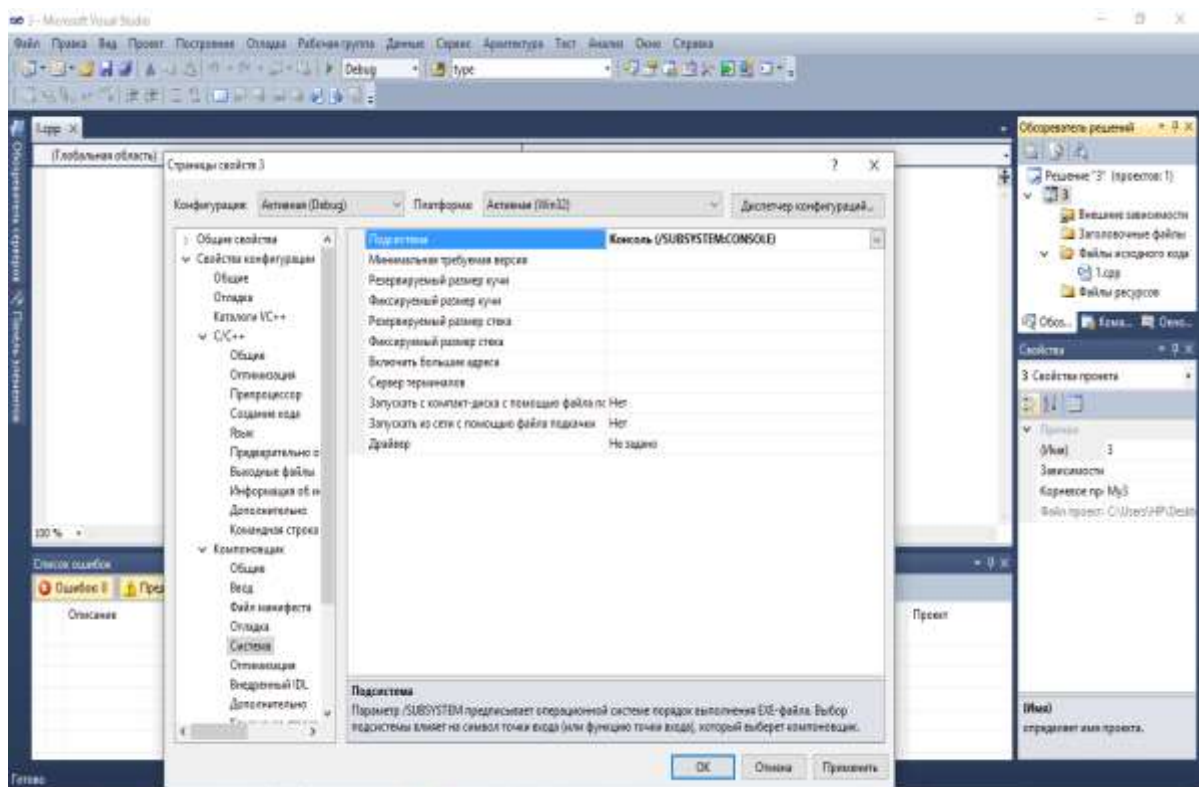


Рис. 4. Страница свойств 3

Переходим в Компоновщик -> Ввод -> Дополнительные зависимости.

Здесь необходимо нажать Изменить и в Дополнительные зависимости (рис. 5.) прописать следующие строки:

SDL2.lib

SDL2main.lib

SDL2test.lib

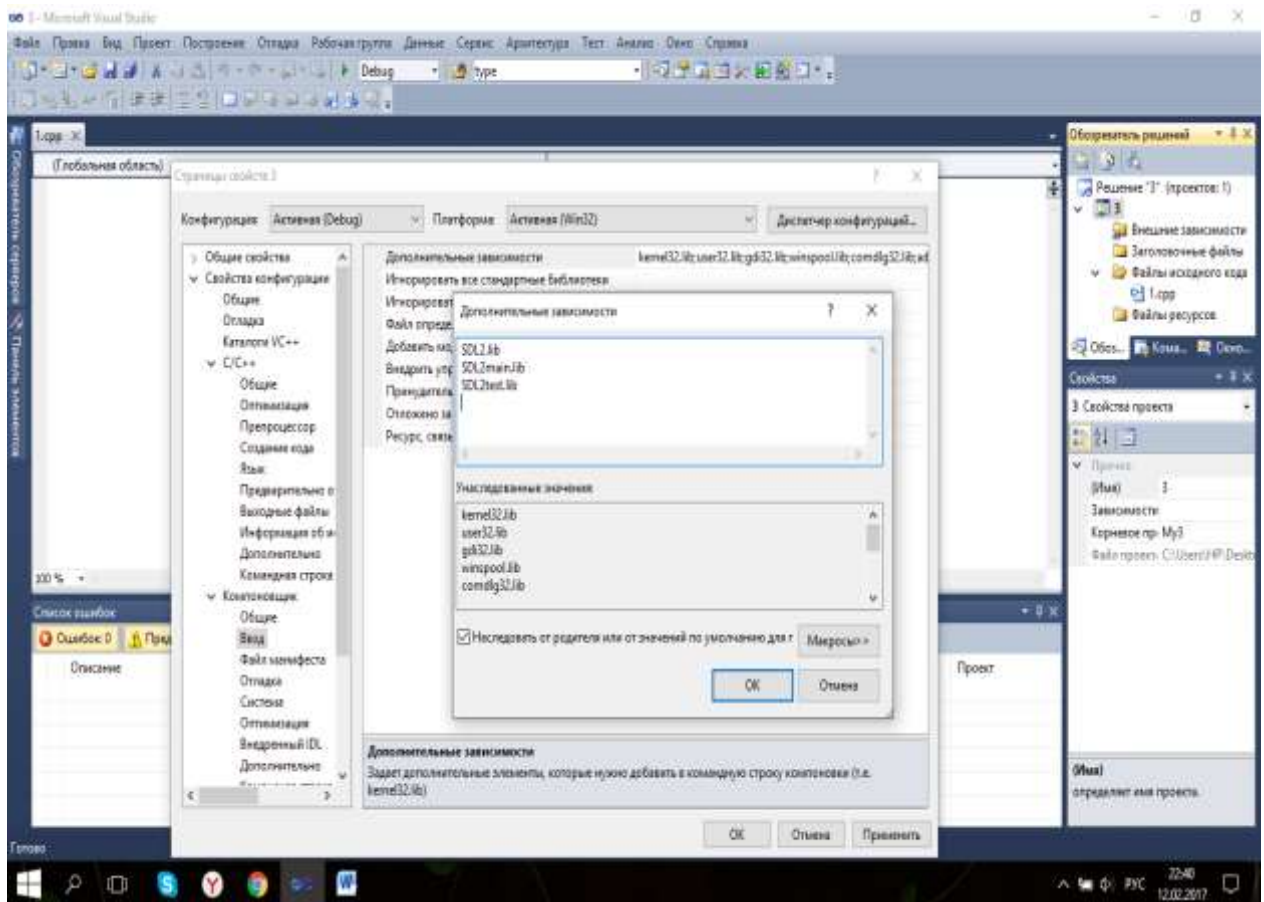


Рис.5. Окно свойств дополнительных зависимостей

Из распакованного архива файл SDL.dll копируем в папку Debug проекта. На этом подключение библиотеки завершено.

Теперь построим простейшую программу, отображающую пустое окно.

Подключим SDL и stdio:

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

Зададим размер окна:

```
const int SCREEN_WIDTH = 640;
```

```
const int SCREEN_HEIGHT = 480;
```

Внутри функции `main()` объявим объекты окна и поверхности:

```
//окно
SDL_Window* window = NULL;

//Поверхность окна

SDL_Surface* screenSurface = NULL;
```

Иницилируем SDL. При ошибке выведем соответствующее сообщение. Внутри `else` в дальнейшем разместим код для вывода окна.

```
if (SDL_Init(SDL_INIT_VIDEO) < 0)
{
    printf("SDL не смог запуститься! SDL_Error: %s\n",
SDL_GetError());
}
else
{
}
```

Создадим окно:

```
window = SDL_CreateWindow("Урок1",
SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN);
if (window == NULL)
{
    printf("Окно не может быть создано! SDL_Error: %s\n",
SDL_GetError());
}
else
{
}
```

По аналогии с ранее рассмотренным кодом при ошибке выведем сообщение. Внутри `else` будем размещать код для закрашивания окна цветом.

Далее зальем окно цветом и установим задержку экрана на три секунды:

```
//Получаем поверхность
screenSurface = SDL_GetWindowSurface(window);

//Заполняем ее белым цветом
SDL_FillRect(screenSurface, NULL,
SDL_MapRGB(screenSurface->format, 0xFF, 0xFF, 0xFF));

//Обновляем поверхность
SDL_UpdateWindowSurface(window);

//Ждем две секунды
SDL_Delay(2000);
```

Очистим память:

```
//И удаляем из памяти окно
SDL_DestroyWindow(window);

//Выход из SDL
SDL_Quit();
```

В итоге получим следующий код:

```
//подключим SDL и stdio
#include <SDL.h>
#include <stdio.h>
//Некоторые константы нашего окна
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
int main(int argc, char* args[])
{    //окно
    SDL_Window* window = NULL;
    //Поверхность окна
    SDL_Surface* screenSurface = NULL;
    //Включим SDL
    if (SDL_Init(SDL_INIT_VIDEO) < 0)
    {
        printf("SDL не смог запуститься! SDL_Error:
%s\n", SDL_GetError());
    }
```

```

else
{
    //Создаем окно
    window = SDL_CreateWindow("Урок1",
        SDL_WINDOWPOS_UNDEFINED, SDL_WINDOWPOS_UNDEFINED,
        SCREEN_WIDTH, SCREEN_HEIGHT, SDL_WINDOW_SHOWN);
    if (window == NULL)
    {
        printf("Окно не может быть создано!\n");
        SDL_Error: %s\n", SDL_GetError());
    }
    else
    {
        //Получаем поверхность
        screenSurface =
        SDL_GetWindowSurface(window);

        //Заполняем ее белым цветом
        SDL_FillRect(screenSurface, NULL,
        SDL_MapRGB(screenSurface->format, 0xFF, 0xFF, 0xFF));

        //Обновляем поверхность
        SDL_UpdateWindowSurface(window);

        //Ждем две секунды
        SDL_Delay(2000);
    }
}
//И удаляем из памяти окно
SDL_DestroyWindow(window);

//Выход из SDL
SDL_Quit();

return 0;
}

```

После запуска будет выведено пустое окно.

Задача 2. Вывести изображение в окне.

Решение. Создадим изображение в формате bmp и сохраним его в папку с нашим проектом. Создадим новую поверхность и загрузим в нее изображение из файла:

```

SDL_Surface *myImage = SDL_LoadBMP("2.bmp");

```

Проверим, осуществилась ли загрузка:

```
if(myImage == NULL)
{
    printf("error");
}
```

Заведем структуру, в которой будут храниться координаты выводимого изображения:

```
SDL_Rect dest;
dest.x = 160;
dest.y = 160;
```

Далее используем следующую функцию:

```
SDL_BlitSurface(myImage, NULL, screenSurface, &dest);
```

Первый параметр указывает на поверхность, в которую загружено изображение, второй задает прямоугольник, который выделяет часть этого изображения для отображения, третий параметр – поверхность, на которую копируется изображение (в данном случае это экран).

Последний параметр задает расположение изображения на экране.

Общий вид функции: `SDL_BlitSurface(SDL_Surface* src, const SDL_Rect* srcrect, SDL_Surface* dst, SDL_Rect* dstrect)`

Изменим цвет окна на желтый и запустим проект.

Результат выполнения программы представлен на рис. 6.

Теперь установим прозрачный цвет для нашего изображения. Будем использовать функцию:

`DL_SetColorKey(SDL_Surface* поверхность, флаг, ключ)`

В нашем случае функция примет вид:

```
SDL_SetColorKey (myImage, SDL_TRUE, SDL_MapRGB (
myImage->format, 255, 255, 255));
```

Результат выполнения программы представлен на рис. 7.

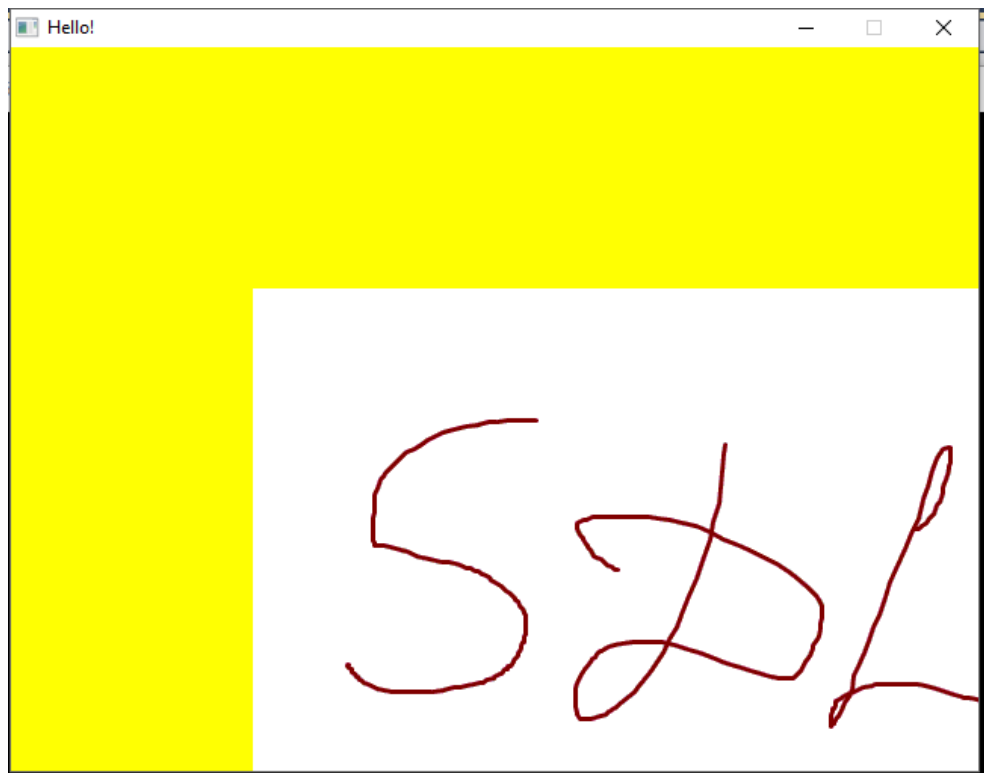


Рис. 6. Окно программы. Шаг 1



Рис. 7. Окно программы. Шаг 2

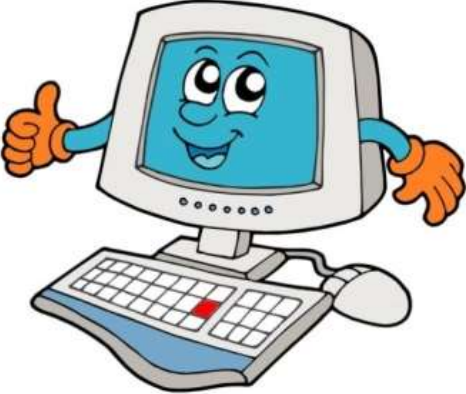

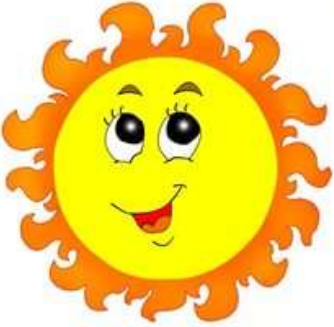
4. Индивидуальная часть лабораторной работы



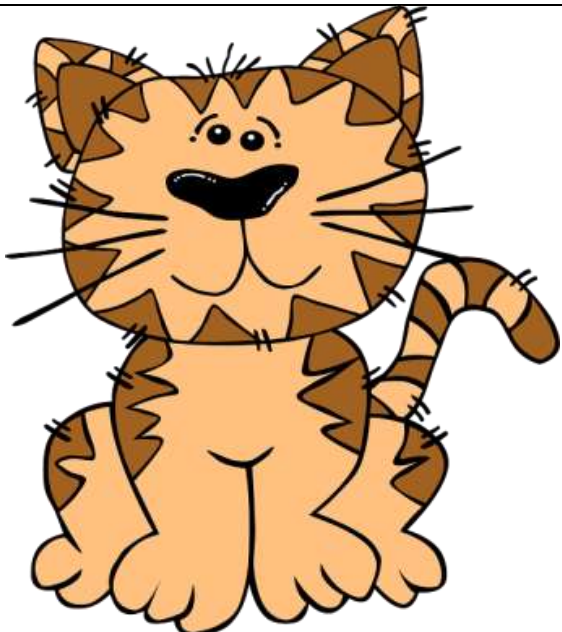
Создать приложение, выводящее на экран окно, заголовком которого является фамилия учащегося. Размер окна, цвет, координаты выводимой картинки и внешний вид картинки взять из табл. 4.




5. Задачи для самостоятельного решения




Таблица 4





Индивидуальные задачи

№	Размер окна	Цвет в RGB формате	Положение картинки в окне	Предлагаемая картинка для отображения
1	640x480	84, 141, 212	Центр экрана	
2	700x800	31, 73, 125	Правый нижний угол	
3	600x600	148, 54, 52	Правый верхний угол	

№	Размер окна	Цвет в RGB формате		Положение картинки в окне	Предлагаемая картинка для отображения
4	640x480		118, 146, 60	Левый нижний угол	
5	700x800		218, 238, 243	Левый верхний угол	
6	600x600		95, 73, 122	Центр экрана	

	Размер окна	Цвет в RGB формате	Положение картинки в окне	Предлагаемая картинка для отображения
7	640x480	142, 128, 168	Правый нижний угол	
8	700x800	255, 192, 0	Правый верхний угол	
9	600x600	234, 241, 221	Левый нижний угол	

№	Размер окна	Цвет в RGB формате		Положение картинки в окне	Предлагаемая картинка для отображения
10	640x480		219, 229, 241	Левый верхний угол	
11	700x800		255, 192, 0	Центр экрана	
12	600x600		148, 138, 84	Правый нижний угол	

№	Размер окна	Цвет в RGB формате	Положение картинки в окне	Предлагаемая картинка для отображения
13	640x480	0, 32, 96	Правый верхний угол	
14	700x800	242, 219, 219	Левый нижний угол	
15	600x600	250, 191, 143	Левый верхний угол	
16	640x480	204, 0, 255	Центр экрана	

№	Размер окна	Цвет в RGB формате		Положение картинки в окне	Предлагаемая картинка для отображения
17	700x800		146, 208, 80	Правый нижний угол	
18	600x600		0, 128, 128	Правый верхний угол	

6. Контрольные вопросы

1. Для чего предназначена библиотека SDL?
2. Из каких этапов состоит создание простейшего приложения с использованием графики?
3. Каким образом библиотека подключается к проекту?
4. С помощью каких функций возможен вывод изображения на экран средствами SDL?

Лабораторная работа №3. Формирование сложных параметризованных изображений с помощью пользовательских функций

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков создания простейших изображений с помощью библиотеки SDL.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Всё рисование в SDL происходит через рисовальщик – `SDL_Renderer`. Для начала работы его необходимо создать.

Создать рисовальщик можно с помощью функции `SDL_CreateRenderer(window, index, flags)`, где `window` – созданное окно, `index` – номер желаемого рисовальщика (используйте -1, чтобы создать первый подходящий рисовальщик), `flags` – перечисление флагов [4]:

- `SDL_RENDERER_SOFTWARE` – без аппаратного ускорения (медленный рисовальщик, но доступен на всех системах);
- `SDL_RENDERER_ACCELERATED` – использовать аппаратное ускорение;
- `SDL_RENDERER_PRESENTVSYNC` – использовать вертикальную синхронизацию изображения (синхронизирует смену кадра и отображение кадра на мониторе);
- `SDL_RENDERER_TARGETTEXTURE` – поддерживать рисование в текстуру (для продвинутых специальных эффектов).

Если указать 0 вместо флагов, предпочтение будет отдано рисовальщику с аппаратным ускорением.

Возвращаемое значение функции – указатель на созданного рисовальщика либо `NULL` при ошибке.

Почти всегда подойдут следующие параметры для этой функции: созданное окно, -1, 0.

Как и окно, рисовальщик необходимо уничтожить, когда он больше вам не понадобится. Для этого необходимо использовать функцию `SDL_DestroyRenderer(renderer)`.

Хорошим тоном является уничтожение объектов в порядке, обратном порядку их создания, например:

```
#include "SDL.h"
int main(int argc, char** argv)
{
    SDL_Init(SDL_INIT_EVERYTHING);
    SDL_Window* window = SDL_CreateWindow(u8"Привет!
Русский заголовок!",
        SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
        800, 600, SDL_WINDOW_SHOWN);

    SDL_Renderer* renderer = SDL_CreateRenderer(window, -
1, 0);

    SDL_Delay(2000);
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}
```

Другой способ создать рисовальщик — создать его вместе с окном с помощью функции `SDL_CreateWindowAndRenderer`. Эта функция позволяет создать и окно, и рисовальщик за один вызов (что может быть более удобно) и имеет следующие параметры: ширину и высоту окна, флаги, указатель на переменную, куда необходимо записать указатель на созданное окно и указатель на переменную, куда необходимо записать указатель на созданного рисовальщика.

Эта функция возвращает 0, если удалось создать окно и рисовальщик и не 0, если произошла ошибка, например:

```
SDL_Window* window;

SDL_Renderer* renderer;
if (SDL_CreateWindowAndRenderer(800, 600,
SDL_WINDOW_SHOWN, &window, &renderer))
{
    //произошла ошибка
    return 1;
}
```


Обратите внимание, что в этой функции нельзя указать заголовок окна. Если необходимо это сделать, воспользуйтесь специальной функцией для работы с окнами – `SDL_SetWindowTitle`.

Показ результатов рисования на экран

Рисовальщик предоставляет набор функций для рисования в относящемся к нему окне. Чтобы понять, как правильно использовать эти функции, рассмотрим метафоры, использованные при разработке SDL.

Рисовальщик рисует не напрямую в окно, а в отдельную область – буфер. Представьте, что у вас есть два листа. Один вы показываете пользователю, а во второй рисуете следующую картинку. Когда картинка нарисована, вы меняете листы местами. Таким образом, вы начинаете рисовать очередную картинку на том листе, что раньше показывали пользователю. Пользователю же показывается только что завершённая картинка.

Таким образом, нарисованное с помощью рисовальщика, невозможно увидеть на экране, пока не поменяете «листы» местами. Функция, которая отвечает за это – `SDL_RenderPresent(renderer)`, показывает результат рисования на экран.

Рисовальщик и работа с цветами

Ещё одна существенная метафора – использование текущего цвета. Представьте, что у вас есть набор цветных карандашей. Чтобы начать рисовать, выбираете карандаш необходимого цвета. Когда вы нарисовали часть картинки нужным цветом, вы выбираете следующий карандаш и рисуете им.

Рисовальщик всегда рисует определённым цветом. Если необходимо начать рисовать другим цветом, вызывают специальную функцию, чтобы указать рисовальщику, что ему необходимо сменить цвет. Это позволяет не указывать цвет в каждой команде рисовальщику, что удобно, если необходимо выполнить много команд с одним и тем же цветом.

Существует несколько моделей представления цвета. В компьютерной графике часто используется модель RGB (красный, зелёный, синий). В этой модели каждый цвет получается в результате суммы красного, зелёного и синего цветов разной интенсивности. Интенсивности этих основных цветов задаются числами

из определённого диапазона (например, 0..1 или 0..255). Эта модель отличается от принятой в живописи модели, основными цветами в которой являются красный, жёлтый и синий. Это связано с тем, что краски «вычитают» цвет из цвета на бумаге (если смешать все краски, получится чёрный цвет), а светодиоды – «прибавляют» (если наложить красный, зелёный и синий лучи света, получится луч белого цвета).

Рассмотрим несколько примеров представления цветов в модели RGB [2]. Если использовать диапазон 0..255 для задания интенсивности компонентов, ярко красный цвет будет иметь представление (255, 0, 0), т.е. в нём присутствует только красная составляющая максимальной интенсивности. Аналогично, ярко синий цвет будет иметь представление (0, 0, 255). Оттенки белого цвета содержат компоненты в равных долях, например ярко белый – (255, 255, 255), серый – (128, 128, 128), а чёрный – (0, 0, 0). Жёлтый цвет состоит из красного и зелёного – (255, 255, 0). Более тёмные оттенки цвета можно получить, уменьшив значение компонентов, например один из тёмно-жёлтых цветов имеет представление – (170, 170, 0).

Рисование с помощью непрозрачных цветов достаточно неудобно и больше напоминает аппликацию. Использование прозрачности позволяет смешивать имеющийся цвет изображения с цветом рисуемого изображения. Так, чтобы сделать уже нарисованное изображение более тёмным, можно нарисовать поверх него полупрозрачный чёрный прямоугольник.

Для представления прозрачности в SDL используется дополнительный компонент цвета – alpha. Величина alpha компонента определяет, насколько прозрачным является цвет. Так, если компонент цвета alpha равен нулю – цвет полностью непрозрачный, если равен максимальному значению – полностью прозрачный. В SDL используется запись RGBA для обозначения компонент цвета. В этой записи alpha располагается на последнем месте. Так (255, 0, 0, 10) является менее прозрачным красным цветом, чем (255, 0, 0, 128).

Для установки текущего цвета рисования рисовальщика используется функция `SDL_SetRenderDrawColor(renderer, r, g, b, a)`. Аргументы функции `r`, `g`, `b` и `a` – значения компонентов красного, зелёного, синего цвета и прозрачность. Эти компоненты задаются

в диапазоне 0..255. В SDL для этого используется тип данных Uint8, который является синонимом unsigned char.

Функция `SDL_RenderClear(renderer)` очищает кадр, т.е. заливает весь кадр одним цветом – текущим цветом рисовальщика.

Рисование примитивов

В SDL используется следующая система координат для задания позиции точек в окне приложения. Верхний левый угол окна имеет координаты (0; 0), правый нижний – (w; h), где w и h – ширина и высота созданного окна соответственно. Другими словами, центр координат находится в левом верхнем угле, ось X направлена вправо, ось Y вниз.

Рисовальщик в SDL предоставляет методы для рисования следующих примитивов: точка, линия, прямоугольник со сторонами параллельными осям координат, закрашенный прямоугольник [3].

- `SDL_RenderDrawPoint(renderer, x, y)` – нарисовать точку в координатах x, y.
- `SDL_RenderDrawLine(renderer, x1, y1, x2, y2)` – нарисовать отрезок из точки (x1; y1) в точку (x2, y2).
- `SDL_RenderDrawRect(renderer, rect)` – нарисовать прямоугольник, `rect` – указатель на структуру, описывающую координаты прямоугольника (описано ниже).
- `SDL_RenderFillRect(renderer, rect)` – нарисовать закрашенный прямоугольник.

Кроме того, рисовальщик позволяет нарисовать набор примитивов за один вызов функции. Необходимые координаты передаются через массив специальных структур.

- `SDL_RenderDrawPoints(renderer, points, count)` – нарисовать count точек из массива points. Каждая точка задаётся структурой, описывающей её координаты.
- `SDL_RenderDrawLines(renderer, points, count)` – нарисовать ломаную из count-1 отрезков, заданных count точками из массива points.
- `SDL_RenderDrawRects(renderer, rects, count)` – нарисовать набор прямоугольников.
- `SDL_RenderFillRects(renderer, rects, count)` – нарисовать набор закрашенных прямоугольников.

3. Общая часть лабораторной работы

Задача 1. Создать окно, с изменяющимся цветом фона. Цвет окна изменяется от белого к черному.

Решение. Поставленную задачу решает следующий код:

```
#include "SDL.h"
int main(int argc, char** argv)
{
    SDL_Init(SDL_INIT_EVERYTHING);
    SDL_Window* window = SDL_CreateWindow(u8"Привет!
Русский заголовок!",
        SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
        800, 600, SDL_WINDOW_SHOWN);

    SDL_Renderer* renderer = SDL_CreateRenderer(window, -
1, 0);
    for (int i = 255; i >= 0; i-=10) {
        SDL_SetRenderDrawColor(renderer, i, i, i, 0);
        SDL_RenderClear(renderer);
        SDL_RenderPresent(renderer); // Показываем
результат
        SDL_Delay(100);
    }

    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}
```

Отметим, что серый цвет задается как равенство двух компонент.

Задача 2. Создать эффект «северного сияния».

Решение. Для эффекта «северного сияния» можно менять цвет окна по трем компонентам.

```
#include "SDL.h"
int main(int argc, char** argv)
{
    SDL_Init(SDL_INIT_EVERYTHING);
    SDL_Window* window = SDL_CreateWindow("Привет!
Русский заголовок!",
        SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED,
```

```

        800, 600, SDL_WINDOW_SHOWN);
    SDL_Renderer* renderer = SDL_CreateRenderer(window, -1,
0);
    for (int i = 255; i >= 0; i-=10)
        for (int j = 255; j >= 0; j-=10)
            for (int k = 255; k >= 0; k-=10){
                SDL_SetRenderDrawColor(renderer, i, j, k, 0);
                SDL_RenderClear(renderer);
                SDL_RenderPresent(renderer); // Показываем
результат
                SDL_Delay(10);
            }
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}

```

Пример запуска программы представлен на рис. 1.

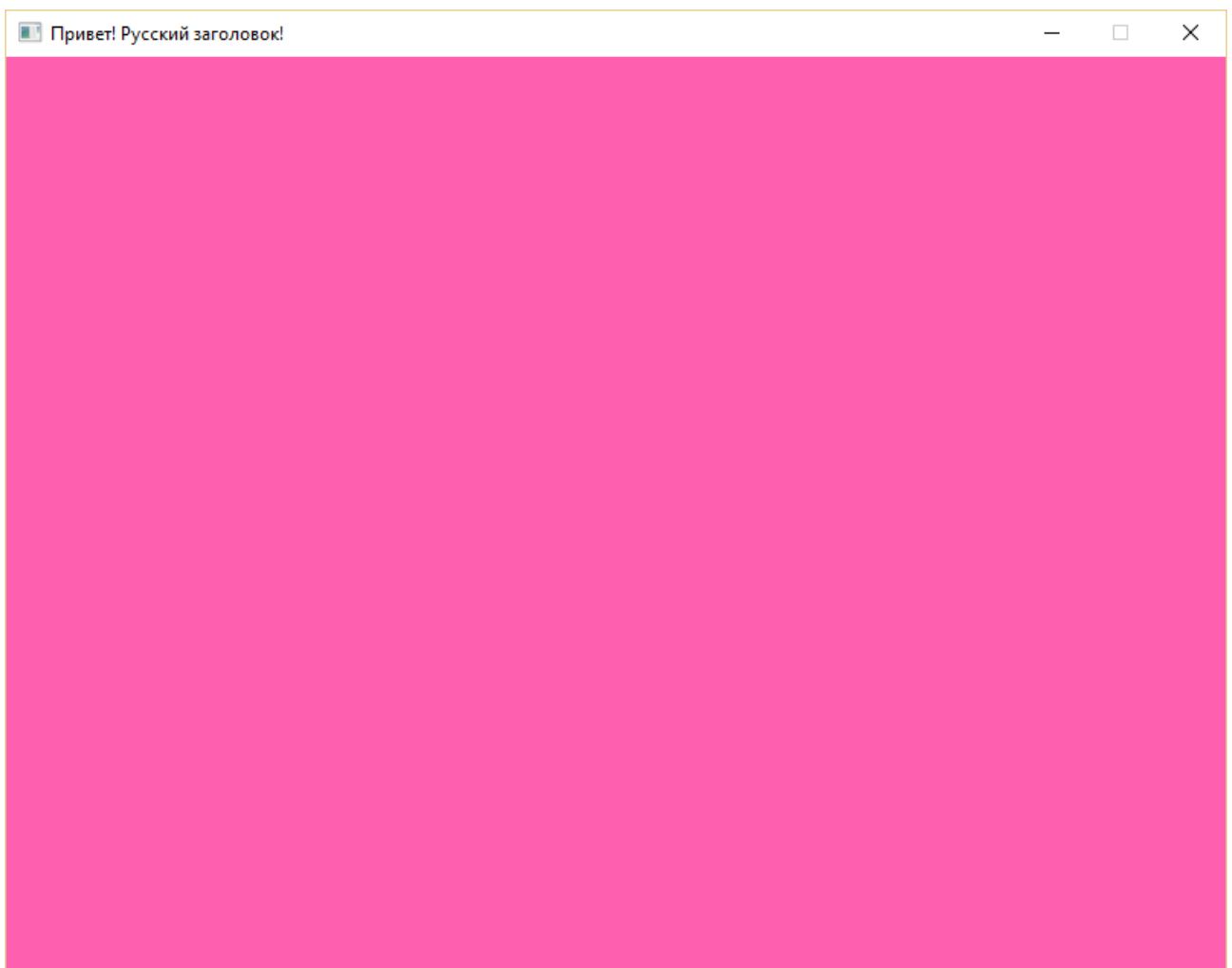


Рис. 1. Запуск программы

Задача 3. Вывести несколько прямоугольников.

Решение. Поставленную задачу решает следующий код:

```
#include "SDL.h"
int main(int argc, char** argv)
{
    SDL_Init(SDL_INIT_EVERYTHING);
    SDL_Window* window = SDL_CreateWindow("Привет!
Русский заголовок!",
        100, 100, 800, 600, SDL_WINDOW_SHOWN);
    SDL_Renderer* renderer = SDL_CreateRenderer(window, -
1, 0);

    SDL_SetRenderDrawColor(renderer, 255, 255, 255, 0);
    SDL_RenderClear(renderer);

    SDL_SetRenderDrawColor(renderer, 0, 0, 0, 0);
    SDL_Rect rect = { 100, 100, 50, 40 };
    SDL_RenderDrawRect(renderer, &rect);

    SDL_Rect rect2 = { 200, 100, 50, 60 };
    SDL_RenderFillRect(renderer, &rect2);

    SDL_SetRenderDrawColor(renderer, 200, 200, 0, 0);

    rect.y = 200;
    SDL_RenderDrawRect(renderer, &rect);

    rect2.x = 300;
    rect2.y = 50;
    rect2.w = 10;
    rect2.h = 30;
    SDL_RenderDrawRect(renderer, &rect2);

    SDL_RenderPresent(renderer);
    SDL_Delay(3000);

    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}
```

Пример запуска программы представлен на рис. 2.

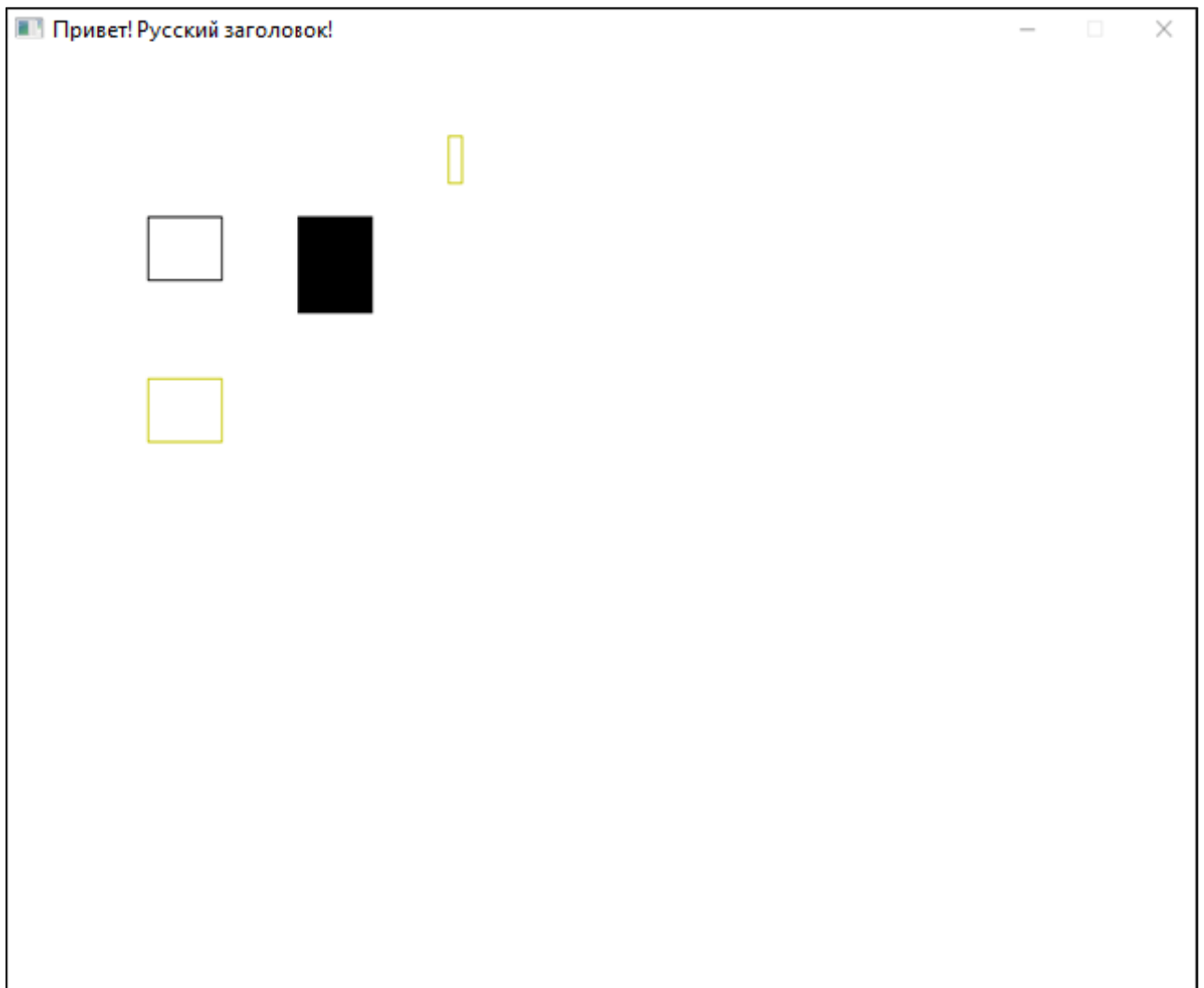


Рис. 2. Запуск программы

Задача 4. Вывести прямоугольники по диагонали, «лесенкой», закрашенные в шахматном порядке.

Решение. Поставленную задачу решает следующий код:

```
#include "SDL.h"

int main(int argc, char** argv)
{
    SDL_Init(SDL_INIT_EVERYTHING);
    SDL_Window* window = SDL_CreateWindow("Привет! Русский
заголовок!",
    100, 100,
    800, 600, SDL_WINDOW_SHOWN);
```

```

SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, 0);
SDL_SetRenderDrawColor(renderer, 255, 255, 255, 0);
SDL_RenderClear(renderer);
SDL_SetRenderDrawColor(renderer, 0, 0, 0, 0);
int x = 100;
int y = 100;
int size = 40;
for (int i = 0; i < 8; i++)
{
    SDL_Rect rect = { x + i*size, y + i*size, size + 1, size +
1 };
    if (i % 2 == 0) {
        SDL_RenderDrawRect(renderer, &rect);
    }
    else {
        SDL_RenderFillRect(renderer, &rect);
    }
}

SDL_RenderPresent(renderer);
SDL_Delay(3000);
SDL_DestroyRenderer(renderer);
SDL_DestroyWindow(window);
SDL_Quit();
return 0;
}

```

Пример запуска программы представлен на рис. 3.

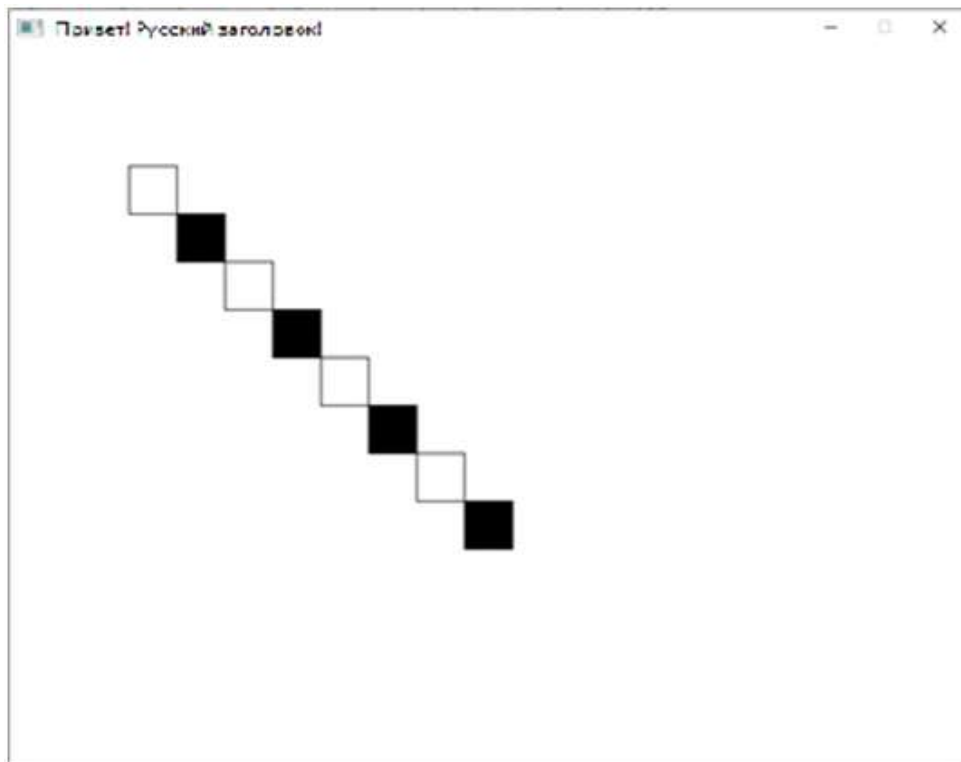


Рис. 3. Запуск программы

4. Задачи для самостоятельного решения

Задача 1. С помощью графических примитивов вывести на экран следующие изображения:

- 1) дом;
- 2) конверт;
- 3) кораблик;
- 4) светофор;
- 5) прямоугольный параллелепипед;
- 6) тетраэдр;
- 7) старую эмблему Windows;
- 8) имя студента;
- 9) компьютер (монитор + системный блок + клавиатура);
- 10) ключу;
- 11) ёлочка;
- 12) навесной замок;
- 13) ноутбук;
- 14) цветные карандаши;
- 15) пирамиду.

Обязательно в изображении использовать минимум 3 цвета, а также заливку фона.

Задача 2. Сформировать изображение согласно варианту:

1. Шахматная доска.
2. Звездное небо.
3. Вертикальные полосы.
4. Концентрические окружности.
5. Снеговик.
6. Олимпийская эмблема.
7. Кирпичная кладка.
8. Диагональные полосы

Использовать циклы и пользовательские функции.

Задача 3. Для заданного интервала отобразить на экране оси координат и график указанной функции согласно варианту:

1. $Y = A \sin(Bx + C) + D$, $[-5; 5]$.
2. $Y = Ax^3 + Bx^2 + Cx + D$, $[-10; 10]$.

3. $Y=(Ax+B)^{0.5}+C$, $[0;15]$.
4. $Y=A\cos(Bx+C)+D$, $[-12;12]$.
5. $Y=A\ln(x)+B$, $[1;15]$.
6. $Y=Ax^5+Bx^3+Cx+D$, $[-10;20]$.
7. $Y=A\cos(Bx)+C\sin(Dx)$, $[-8;8]$.
8. $Y=A(x+B)^2+Cx^{0.5}+D$, $[0;10]$.
9. $Y=A\cos(Bx)+Cx^{0.5}+D$, $[0;20]$.
10. $Y=A\ln(x^2+B)+C$, $[1;10]$.

Коэффициенты A,B,C,D вводит пользователь.

Задача повышенной сложности. Вывести изображение часов, поместив на них часовую, минутную и секундную стрелки в соответствии с заданным пользователем временем. Стрелки должны иметь разную ширину и длину. На циферблате должны отображаться римские цифры от 1 до 12. Отрисовка часов должна выполняться с помощью пользовательской функции, принимающей в качестве параметров координаты центра часов и размер часов (радиус циферблата).

5. Контрольные вопросы

1. Для чего предназначена библиотека SDL?
2. Из каких этапов состоит создание простейшего приложения с использованием графики?
3. Каким образом библиотека подключается к проекту?
4. С помощью каких функций возможен вывод изображения на экран средствами SDL?

Лабораторная работа №4. Простейшая анимация

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков создания простейших анимированных изображений с помощью библиотеки SDL.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Анимация – это процесс создания иллюзии движения и изменения формы с помощью быстрой смены статичных изображений, которые обычно минимально отличаются друг от друга.

Каждый кадр должен быть нарисован отдельно и показывается на короткий промежуток времени, именно это и создаёт иллюзию движения.

Для отображения анимации необходимо внутри цикла выполнить следующие действия [1]:

- очистить кадр;
- произвести рисование;
- показать кадр;
- произвести задержку экрана.

3. Общая часть лабораторной работы

Задача 1. Воспроизвести движение прямоугольника сверху экрана вниз.

Решение. Для начала необходимо инициализировать графический режим и создать окно:

```
SDL_Init(SDL_INIT_EVERYTHING);  
SDL_Window* window = SDL_CreateWindow("HI",  
    100, 100,  
    800, 600, SDL_WINDOW_SHOWN);  
SDL_Renderer* renderer = SDL_CreateRenderer(window, -1,  
0);
```

Отметим, что после выполнения программы память должна быть очищена:

```
SDL_DestroyRenderer(renderer);
```

```
SDL_DestroyWindow(window);
SDL_Quit();
```

Между двумя этими блоками будет отображаться цикл, реализующий анимацию. Нарисуем прямоугольник, размером 100 единиц.

Рисование будет производиться в цикле со счётчиком:

```
for (int i = -100; i <= 600; i += 10) { }
```

Такие параметры изменения счетчика заданы, исходя из размеров окна с учетом того, что прямоугольник будет появляться из-за границы отображаемой области.

Очистим содержимое окна. Выполним заливку черным цветом:

```
SDL_SetRenderDrawColor(renderer, 0, 0, 0, 0);
SDL_RenderClear(renderer);
```

Выполним рисование прямоугольника фиолетового цвета, изменяя его координату по вертикальной оси в соответствии со значением счетчика:

```
SDL_SetRenderDrawColor(renderer, 128, 0, 128, 0);
SDL_Rect target = { 300, i, 100, 100 };
SDL_RenderFillRect(renderer, &target);
```

Выведем полученное изображение на экран и выполним задержку экрана:

```
SDL_RenderPresent(renderer);
SDL_Delay(20);
```

Чем большее значение будет передано функции `SDL_Delay()`, тем на большее время будет осуществляться задержка экрана и, следовательно, тем медленнее будет происходить анимационное движение.

Полный код программы представлен на рис. 1, результат выполнения программы на рис. 2.

```
#include "SDL.h"

int main(int argc, char** argv)
{
    SDL_Init(SDL_INIT_EVERYTHING);
    SDL_Window* window = SDL_CreateWindow("HI",
        100, 100,
        800, 600, SDL_WINDOW_SHOWN);
    SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, 0);

    for (int i = -100; i <= 600; i += 10) {

        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 0);
        SDL_RenderClear(renderer);

        SDL_SetRenderDrawColor(renderer, 128, 0, 128, 0);
        SDL_Rect target = { 300, i, 100, 100 };
        SDL_RenderFillRect(renderer, &target);

        SDL_RenderPresent(renderer);
        SDL_Delay(20);
    }

    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}
```

Рис. 1. Код программы

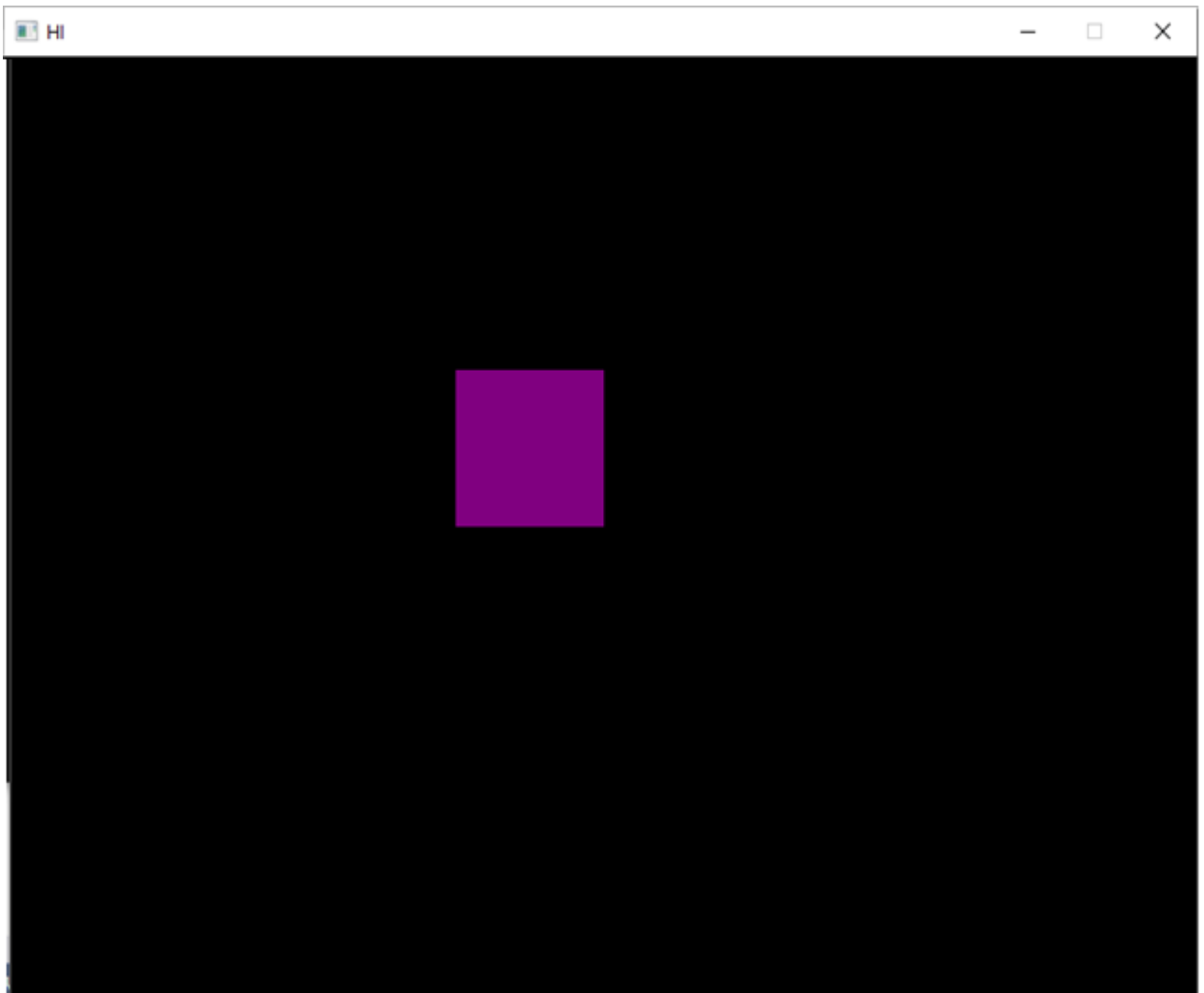


Рис. 2. Результат выполнения программы

Задача 2. Реализовать движение прямоугольника по кругу.

Решение. Получить координаты на окружности для заданного угла можно следующим образом:

$$x = r * \cos(u)$$

$$y = r * \sin(u)$$

Пусть прямоугольник совершает четыре полных оборота. Зададим параметрами счетчика градусную меру угла от 0 до 1440 ($360 * 4 = 1440$):

```
for (int i = 0; i <= 1440; i++) { }
```

Заливка экрана цветом и установка цвета рисования по сравнению с предыдущим примером не изменится. Для получения

значений синуса и косинуса потребуется подключить библиотеку `math.h`.

Заметим, что эти функции на вход получают значения угла в радианах. Определим вспомогательную переменную, в которой будет производиться пересчет значения угла в радианы:

```
double u=3.14/180*i;
```

Далее создадим структуру, описывающую прямоугольник, которую зададим следующим образом:

```
SDL_Rect target = { int (300+100*cos(u)), int  
(200+100*sin(u)), 100, 100 };
```

Здесь радиус окружности равен 100, кроме того, задано смещение относительно начала координат 300 по горизонтали и 200 по вертикали для того, чтобы рисунок не выходил за границу окна. Отметим, что результаты вычислений необходимо привести к целому типу. Полученный код программы приведен на рис. 3.

```
#include "SDL.h"
#include <math.h>
#include <stdio.h>

int main(int argc, char** argv)
{
    SDL_Init(SDL_INIT_EVERYTHING);
    SDL_Window* window = SDL_CreateWindow("HI",
        100, 100,
        800, 600, SDL_WINDOW_SHOWN);
    SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, 0);

    for (int i = 0; i <= 1440; i++) {
        SDL_SetRenderDrawColor(renderer, 0, 0, 0, 0);
        SDL_RenderClear(renderer);
        SDL_SetRenderDrawColor(renderer, 128, 0, 128, 0);
        double u=3.14/180*i;
        SDL_Rect target = { int (300+100*cos(u)), int (200+100*sin(u)), 100, 100 };
        SDL_RenderFillRect(renderer, &target);
        SDL_RenderPresent(renderer);
        SDL_Delay(10);
    }

    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}
```

Рис.3. Код программы

4. Задачи для самостоятельного решения

Реализовать в соответствии с вариантами (варианты выбрать $N_{1\text{вар}} = N_{\text{спис}} \% 6 + 1$, $N_{2\text{вар}} = (N_{\text{спис}} + 2) \% 6 + 1$) два из следующих анимационных сюжетов:

1. Падающий снег.
2. Ползущая змейка (змейка из 5 квадратов движется зигзагом слева направо).
3. Движение прямоугольника в случайном направлении с отскоком от стенок.
4. Равноускоренное движение окружности сверху вниз с отскоком от пола (симуляция мячика).
5. Движение трех точек по круговой орбите с различным радиусом орбит и скоростей.
6. Движение двух окружностей вокруг центра по эллиптической орбите.

5. Контрольные вопросы

1. Для чего предназначена библиотека SDL?
2. Из каких этапов состоит создание простейшего приложения с использованием графики?
3. Из каких этапов состоит отображение простейшей анимации средствами SDL?

Лабораторная работа №5. Интерактивная анимация

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков создания интерактивной анимации с помощью библиотеки SDL.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Подсистема обработки событий в библиотеке SDL является достаточно сложной, так как ее основная задача – объединить потребности разработчиков мультимедийных приложений (особенно игр) в быстром отклике на действия пользователя с возможностями современных многозадачных операционных систем по обработке событий от множества независимых источников, причем механизмы такой обработки могут существенно отличаться на различных платформах. При этом библиотека SDL должна еще сделать логически идентичную обработку событий на различных платформах существенно проще и более единообразной для программиста, чем непосредственное использование возможностей операционных систем. Для обеспечения поставленных задач библиотека SDL предоставляет механизмы по обработке событий, возникающих «извне» программы, таких как нажатие клавиш клавиатуры, перемещение мыши или нажатие клавиш на ней, изменение размера окна программы или его закрытие средствами операционной системы и т.д.

Для сохранения каждого полученного сообщения в общем виде и последующего анализа используется объединение типа `SDL_Event`.

Функция `SDL_PollEvent` вызывает функцию принудительного обновления очереди событий (однократный опрос возможных источников событий – функция `PumpEvents`, не имеющая аргументов и не возвращающая никакого значения), затем проверяет, имеется ли в очереди хотя бы одно событие любого типа, ожидающее обработки. Если очередь пуста, функция `SDL_PollEvent` возвращает 0. Если очередь не пуста и параметр `event` не равен `NULL`, то очередное (первое) событие извлекается из очереди и сохраняется в объединении `SDL_Event`.

Для символического представления констант, описывающих разные типы событий, в SDL используется перечисление `SDL_EventType`. Рассмотрим основные типы событий, обозначая их

здесь и далее соответствующими константами из этого перечисления (то есть, «событие SDL_KEYDOWN» – событие, при котором поле type объединения SDL_Event имеет значение, равное значению константы SDL_KEYDOWN) [4]:

- SDL_ACTIVEEVENT – приложение стало активным (с которым работает пользователь) или перестало быть активным;
- SDL_KEYDOWN – нажата клавиша на клавиатуре;
- SDL_KEYUP – отпущена клавиша на клавиатуре;
- SDL_MOUSEMOTION – перемещена мышь;
- SDL_MOUSEBUTTONDOWN – нажата клавиша мыши;
- SDL_MOUSEBUTTONUP – отпущена клавиша мыши;
- SDL_QUIT – запрос выхода из программы по действию пользователя (например, по нажатию мышью системной кнопки закрытия окна);
- SDL_VIDEORESIZE – пользователь изменил размер окна и требуется изменение видеорежима.

Полный список событий представлен на официальном сайте: https://wiki.libsdl.org/SDL_EventType.

Коды клавиш клавиатуры представлены по ссылке на официальный сайт SDL: https://wiki.libsdl.org/SDL_Scancode

3. Общая часть лабораторной работы

Задача 1. Написать программу, демонстрирующую смену фона окна при нажатии на кнопку.

Решение. Для начала необходимо инициализировать графический режим и создать окно, заливая его черным:

```
SDL_Init(SDL_INIT_EVERYTHING);
SDL_Window* window = SDL_CreateWindow("HI",
    100, 100,
    800, 600, SDL_WINDOW_SHOWN);
SDL_Renderer* renderer = SDL_CreateRenderer(window, -1,
0);

SDL_SetRenderDrawColor(renderer, 10, 10, 50, 0);
SDL_RenderClear(renderer); SDL_RenderPresent(renderer);
SDL_Delay(1000);
```

Отметим, что после выполнения программы память должна быть очищена:

```
SDL_DestroyRenderer(renderer);
SDL_DestroyWindow(window);
SDL_Quit();
```

Для начала упростим задачу и напишем программу, демонстрирующую заливку окна разными цветами при нажатии на левую и правую кнопку мыши.

Создадим переменную, в которую будут заноситься происходящие события, и логическую переменную, принимающую значение «истина» при необходимости выхода из программы.

```
SDL_Event event;
bool quit = false;
```

Создадим игровой цикл, который будет завершен, когда логическая переменная quit станет равна истине.

```
while(!quit)
{
}
```

Внутри цикла считаем текущее событие:

```
SDL_PollEvent(&event);
```

Обработаем событие нажатие на кнопку закрытия окна:

```
if(event.type == SDL_QUIT)
    quit=true;
```

Далее обработаем события нажатия на правую и левую кнопку мыши (изменение цвета заливки окна):

```
if(event.button.button == SDL_BUTTON_LEFT)
{
    SDL_SetRenderDrawColor(renderer, 77, 88, 90, 0);
    SDL_RenderClear(renderer);
    SDL_RenderPresent(renderer);
}
```

```

if(event.button.button == SDL_BUTTON_RIGHT)
{
    SDL_SetRenderDrawColor(renderer, 45, 55, 66, 0);
    SDL_RenderClear(renderer);
    SDL_RenderPresent(renderer);
}

```

Задача 2. Написать программу, переключения между различными цветами с помощью нескольких кнопок.

Решение. Модифицируем программу из предыдущей задачи.

Нарисуем три прямоугольника, которые будут выполнять функцию кнопок, а также обработаем событие нажатия левой кнопки мыши на кнопки. Обработать событие для правой кнопки мыши нам теперь не требуется.

Опишем структуру, задающую прямоугольник (координаты, размер и цвет):

```

struct Rect
{int x,y,dx,dy;
    int r,g,b;
};

```

Создадим функцию, рисующую массив из трех таких прямоугольников:

```

void DrawRect(Rect rect[3], SDL_Window* window,
SDL_Renderer* renderer)
{
    for (int i=0;i<3;i++)
    {
        SDL_SetRenderDrawColor(renderer, rect[i].r,
            rect[i].g, rect[i].b, 0);
        SDL_Rect rect2 = { rect[i].x, rect[i].y,
            rect[i].dx, rect[i].dy };
        SDL_RenderFillRect(renderer, &rect2);
    }
    SDL_RenderPresent(renderer);
}

```

В функции `main()` до игрового цикла создадим массив прямоугольников описанной структуры и инициализируем его

значениями. После этого вызовем функцию отрисовки прямоугольников.

```
Rect r[3]={0,0,50,50,50,50,100, 0, 60,50,50,50,100,50,
0,120,50,50,100,50,50};
DrawRect(r, window, renderer);
```

Проверку нажатия кнопки реализуем следующим образом. Если кнопка мыши нажата, в цикле осуществим проверку нахождения мыши внутри одного из трех квадратов. При попадании координат мыши в прямоугольник, зальем окно соответствующим цветом и повторно выведем на экран прямоугольники при помощи ранее описанной функции:

```
if(event.button.button == SDL_BUTTON_LEFT)
{
    for (int i=0;i<3;i++)
        if ((event.button.x >= r[i].x)&&(event.button.x
<= r[i].x+r[i].dx)&&(event.button.y >=
r[i].y)&&(event.button.y <= r[i].y+r[i].dy))
        {
            SDL_SetRenderDrawColor(renderer, r[i].r+30,
r[i].g+30, r[i].b+30, 0);
            SDL_RenderClear(renderer);
            DrawRect(r, window, renderer);
            SDL_RenderPresent(renderer);
        }
}
```

Получим фрагмент кода, представленный на рис. 1. На этом работа над программой завершена.

Задача 3. Написать программу, демонстрирующую управление положением, цветом и размером прямоугольника с помощью клавиатуры.

Решение. Инициализацию графического режима и создание окна произведем так же, как и в предыдущей задаче. Используем структуру из предыдущей задачи для описания прямоугольника.

Функцию рисования массива прямоугольников перепишем для рисования единственного прямоугольника.

```

void DrawRect(Rect rect, SDL_Window* window,
SDL_Renderer* renderer)
{
    SDL_SetRenderDrawColor(renderer, 10,10, 50, 0);
    SDL_RenderClear(renderer);
    SDL_SetRenderDrawColor(renderer, rect.r, rect.g,
        rect.b, 0);
    SDL_Rect rect2 = {rect.x, rect.y, rect.dx, rect.dy};
    SDL_RenderFillRect(renderer, &rect2);
    SDL_RenderPresent(renderer);
}

SDL_Event event ;
bool quit = false;
Rect r[3]={0,0,50,50,50,50,100, 0,60,50,50,50,100,50, 0,120,50,50,100,50,50};
DrawRect(r, window,  renderer);
while(!quit)
{
    SDL_PollEvent(&event);
    if(event.type == SDL_QUIT)
        quit=true;

    if(event.button.button == SDL_BUTTON_LEFT) {
        for (int i=0;i<3;i++)
            if ((event.button.x >= r[i].x)&&(event.button.x <= r[i].x+r[i].dx)
                &&(event.button.y >= r[i].y)&&(event.button.y <= r[i].y+r[i].dy))
            {
                SDL_SetRenderDrawColor(renderer, r[i].r+30, r[i].g+30, r[i].b+30, 0);
                SDL_RenderClear(renderer);
                DrawRect(r, window,  renderer);
                SDL_RenderPresent(renderer);
            }
    }
}

```

Рис. 1. Фрагмент кода программы задачи 2

Переменную, хранящую текущее событие, и логическую переменную завершения игрового процесса наведем аналогично предыдущей задаче:

```

SDL_Event event;
bool quit = false;

```

Инициализируем прямоугольник и вызовем функцию отображения прямоугольника до начала игрового цикла:

```
Rect r={100,100,50,50,50,50,100};
DrawRect(r, window, renderer);
```

Заведем переменные, задающие значение приращений координаты, размера цвета прямоугольника:

```
int d_move=1;
int d_size=1;
int d_color=10;
```

В программе управление положением прямоугольника будет производиться с помощью стрелок. Проверка условия нажатия кнопки «верх» производится следующим образом:

```
if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym
== SDLK_UP)) { }
```

При нажатии стрелки должен производиться пересчет координат с учетом размеров окна, затем должна быть вызвана функция отрисовки прямоугольника с учетом его новых параметров:

```
if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym
== SDLK_UP))
{
    if (r.y>=0) r.y-=d_move; DrawRect(r, window,
renderer);
}
if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym
== SDLK_DOWN))
{
    if (r.y<=600-r.dy) r.y+=d_move; DrawRect(r, window,
renderer);
}
if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym
== SDLK_LEFT))
{
    if (r.x>=0) r.x-=d_move; DrawRect(r, window,
renderer);
}
```

```

if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym
== SDLK_RIGHT))
{
    if (r.x<=800-r.dx) r.x+=d_move; DrawRect(r, window,
renderer);
}

```

Управление размерами прямоугольника будет производиться с помощью клавиш WASD. Необходимо ограничить размер прямоугольника границами окна и запретить назначение отрицательного приращения:

```

if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym
== SDLK_w))
{
    if (r.dy+r.y+d_size<800) r.dy+=d_size; DrawRect(r,
window, renderer);
}
if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym
== SDLK_s))
{
    if (r.dy>1+d_size) r.dy-=d_size; DrawRect(r, window,
renderer);
}
if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym
== SDLK_d))
{
    if (r.dx+r.x+d_size<800) r.dx+=d_size; DrawRect(r,
window, renderer);
}
if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym
== SDLK_a))
{
    if (r.dx>1+d_size) r.dx-=d_size; DrawRect(r, window,
renderer);
}

```

Аналогично организуем управление цветом, задавая приращение цветовых компонентов клавишами 1,2 и 3. Отметим, что значение цветового компонента не должно превышать 255.

```

if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym
== SDLK_1))

```



```

{
    r.r+=d_color%255; DrawRect(r, window, renderer);
}
if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym
== SDLK_2))
{
    r.g+=d_color%255; DrawRect(r, window, renderer);
}
if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym
== SDLK_3))
{
    r.b+=d_color%255; DrawRect(r, window, renderer);
}

```

Листинг игрового цикла примет вид, указанный на рис. 2.

```

while(!quit)
{
    SDL_PollEvent(&event);
    if(event.type == SDL_QUIT)
        quit=true;

    if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym == SDLK_UP)) {if (r.y==0) r.y-=d_move; DrawRect(r, window, renderer);}
    if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym == SDLK_DOWN)) {if (r.y<=600-r.dy) r.y+=d_move; DrawRect(r, window, renderer);}
    if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym == SDLK_LEFT)) {if (r.x==0) r.x-=d_move; DrawRect(r, window, renderer);}
    if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym == SDLK_RIGHT)) {if (r.x<=800-r.dx) r.x+=d_move; DrawRect(r, window, renderer);}

    if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym == SDLK_w)) {if (r.dy+r.y+d_size<800) r.dy+=d_size; DrawRect(r, window, renderer);}
    if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym == SDLK_s)) {if (r.dy>1+d_size) r.dy-=d_size; DrawRect(r, window, renderer);}
    if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym == SDLK_d)) {if (r.dx+r.x+d_size<800) r.dx+=d_size; DrawRect(r, window, renderer);}
    if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym == SDLK_a)) {if (r.dx>1+d_size) r.dx-=d_size; DrawRect(r, window, renderer);}

    if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym == SDLK_1)) {r.r+=d_color%255; DrawRect(r, window, renderer);}
    if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym == SDLK_2)) {r.g+=d_color%255; DrawRect(r, window, renderer);}
    if((event.type == SDL_KEYDOWN) && (event.key.keysym.sym == SDLK_3)) {r.b+=d_color%255; DrawRect(r, window, renderer);}
}

```

Рис. 2. Листинг игрового цикла

На этом разработка программы завершена.

4. Задачи для самостоятельного решения

Реализовать следующие интерактивные анимационные сюжеты в соответствии с вариантом ($N_{\text{вар}}=N_{\text{спис}} \% 6 + 1$):

1. Движение по экрану ромба с возможностью изменения цвета его заливки и размера (по отдельности по ширине и высоте).

2. Поиск пути в простейшем лабиринте (с проверкой столкновения со стенами), управление на клавишах-стрелках.

3. Появление кругов (не более 10) различного диаметра (от 5 до 50) в различных координатах окна (с проверкой отсутствия наложения кругов друг на друга) при нажатии на пробел. При левом щелчке мышью по кругу, происходит его удаление. Организовать контроль количества кругов на поле (не более 10 одновременно).

4. Имитация ловли падающих с верха экрана кругов прямоугольником. Изменение цвета прямоугольника при успешной поимке круга. Прямоугольник управляется мышью, клавишами можно менять скорость падения кругов.

5. Случайным образом на экране появляются и пропадают изображения цели. Если вовремя кликнуть на цель курсором, то цель меняет цвет и исчезает через секунду. Клавишами на клавиатуре можно изменять длительность нахождения целей на экране и их размер.

6. Управление фигурой на экране. С помощью клавиатуры можно менять форму фигуры (квадрат, окружность, снежинка), и ее позицию. По клавише «пробел» позиция и форма фигуры «запоминаются», а по клавише «таб» восстанавливаются запомненные.

5. Контрольные вопросы

1. Для чего предназначена библиотека SDL?
2. Из каких этапов состоит создание простейшего приложения с использованием графики?
3. Из каких этапов состоит отображение простейшей анимации средствами SDL?
4. В чем заключается принцип разработки интерактивной анимации средствами SDL?

Лабораторная работа №6. Интерактивная анимация нескольких объектов

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков создания интерактивной анимации нескольких объектов с помощью библиотеки SDL.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Для сохранения каждого полученного сообщения в общем виде и последующего анализа используется объединение типа `SDL_Event`.

Функция `SDL_PollEvent` вызывает функцию принудительного обновления очереди событий (однократный опрос возможных источников событий – функция `PumpEvents`, не имеющая аргументов и не возвращающая никакого значения), затем проверяет, имеется ли в очереди хотя бы одно событие любого типа, ожидающее обработки. Если очередь пуста, функция `SDL_PollEvent` возвращает 0. Если очередь не пуста и параметр `event` не равен `NULL`, то очередное (первое) событие извлекается из очереди и сохраняется в объединении `SDL_Event`.

Для символического представления констант, описывающих разные типы событий, в SDL используется перечисление `SDL_EventType`. Рассмотрим основные типы событий, обозначая их здесь и далее соответствующими константами из перечисления (то есть, «событие `SDL_KEYDOWN`» – событие, при котором поле `type` объединения `SDL_Event` имеет значение, равное значению константы `SDL_KEYDOWN`) [4]:

- `SDL_ACTIVEEVENT` – приложение стало активным (с которым работает пользователь) или перестало быть активным;
- `SDL_KEYDOWN` – нажата клавиша на клавиатуре;
- `SDL_KEYUP` – отпущена клавиша на клавиатуре;
- `SDL_MOUSEMOTION` – перемещена мышь;
- `SDL_MOUSEBUTTONDOWN` – нажата клавиша мыши;
- `SDL_MOUSEBUTTONUP` – отпущена клавиша мыши;
- `SDL_QUIT` – запрос выхода из программы по действию пользователя (например, по нажатию мышью системной кнопки закрытия окна);

- **SDL_VIDEORESIZE** – пользователь изменил размер окна и требуется изменение видеорежима.

Полный список событий представлен на официальном сайте: https://wiki.libsdl.org/SDL_EventType.

Коды клавиш клавиатуры представлены по ссылке на официальный сайт SDL: https://wiki.libsdl.org/SDL_Scancode

3. Общая часть лабораторной работы

Задача 1. Пусть в окне расположены несколько шаров. При нажатии по шару мышью, он исчезает. В правом верхнем углу отображается количество удаленных шаров.

Решение. Создадим пустой проект и добавим в него файл `Source.cpp`.

Подключим библиотеку **SDL**. Произведем щелчок правой кнопкой мыши по проекту в обозревателе решений и выберем «Управление проектами NuGet» (рис. 1).

Осуществим поиск, как показано на рис. 2.

Осуществим установку, как показано на рис. 3.

Перейдем к файлу с исходным кодом. Инициализируем графический режим, вызовем отображение окна. Для отображения русскоязычного заголовка перед текстом укажем «u8». Добавим обработчик события закрытия окна внутри игрового цикла и освобождение памяти.

Код программы приведен на рис. 4.

Напишем функцию отображения текста.

Пропишем подключение заголовочного файла:

```
#include <SDL_ttf.h>.
```

Подключим библиотеку, как это показано на рис. 5.

При возникновении ошибки добавим в папку `Debug` проекта библиотеку `freetype262d.dll`.

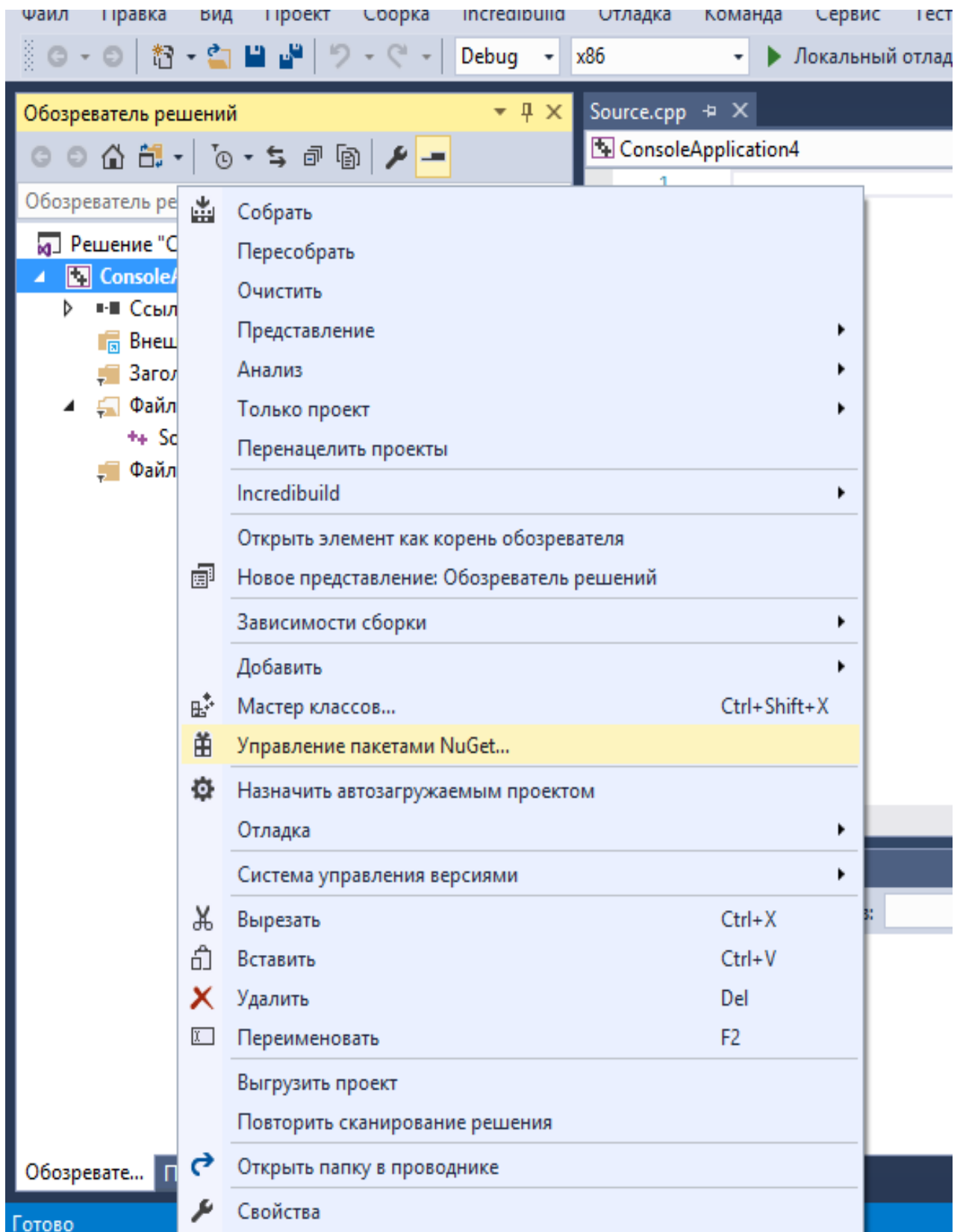


Рис. 1. Пункт меню «Управление проектами NuGet»

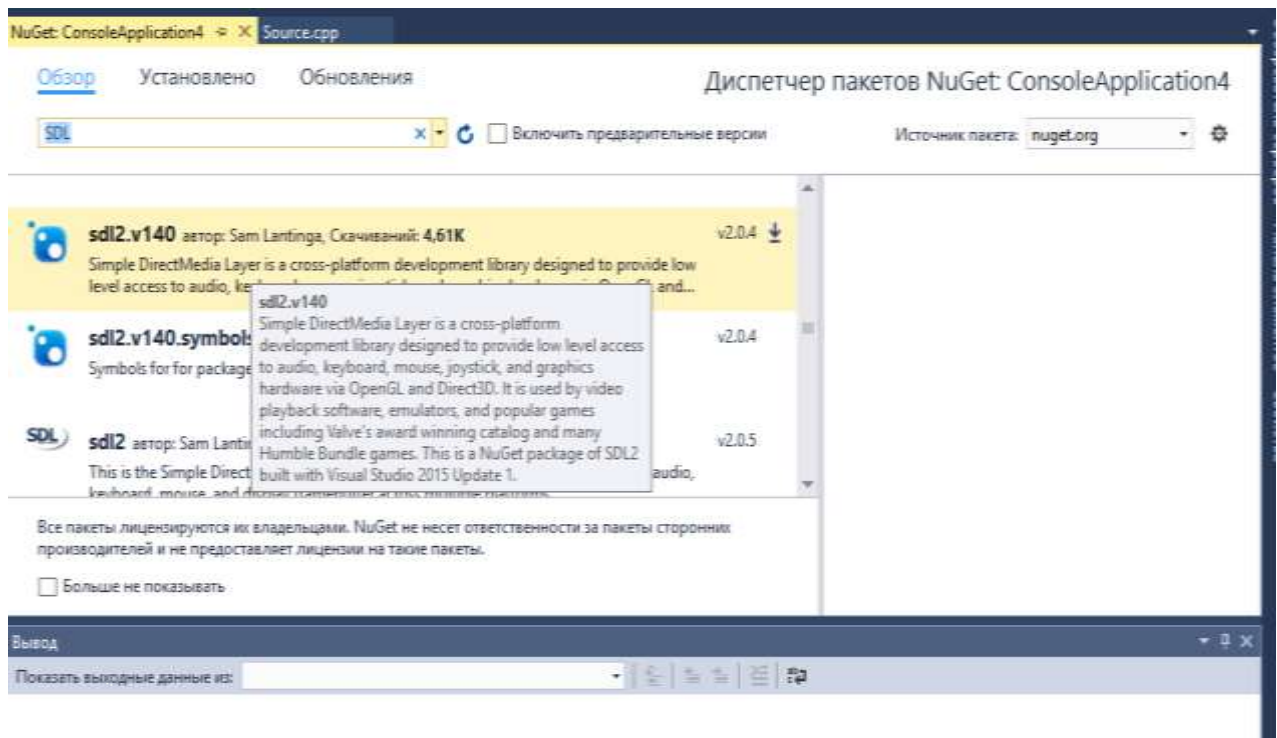


Рис. 2. Поиск библиотеки

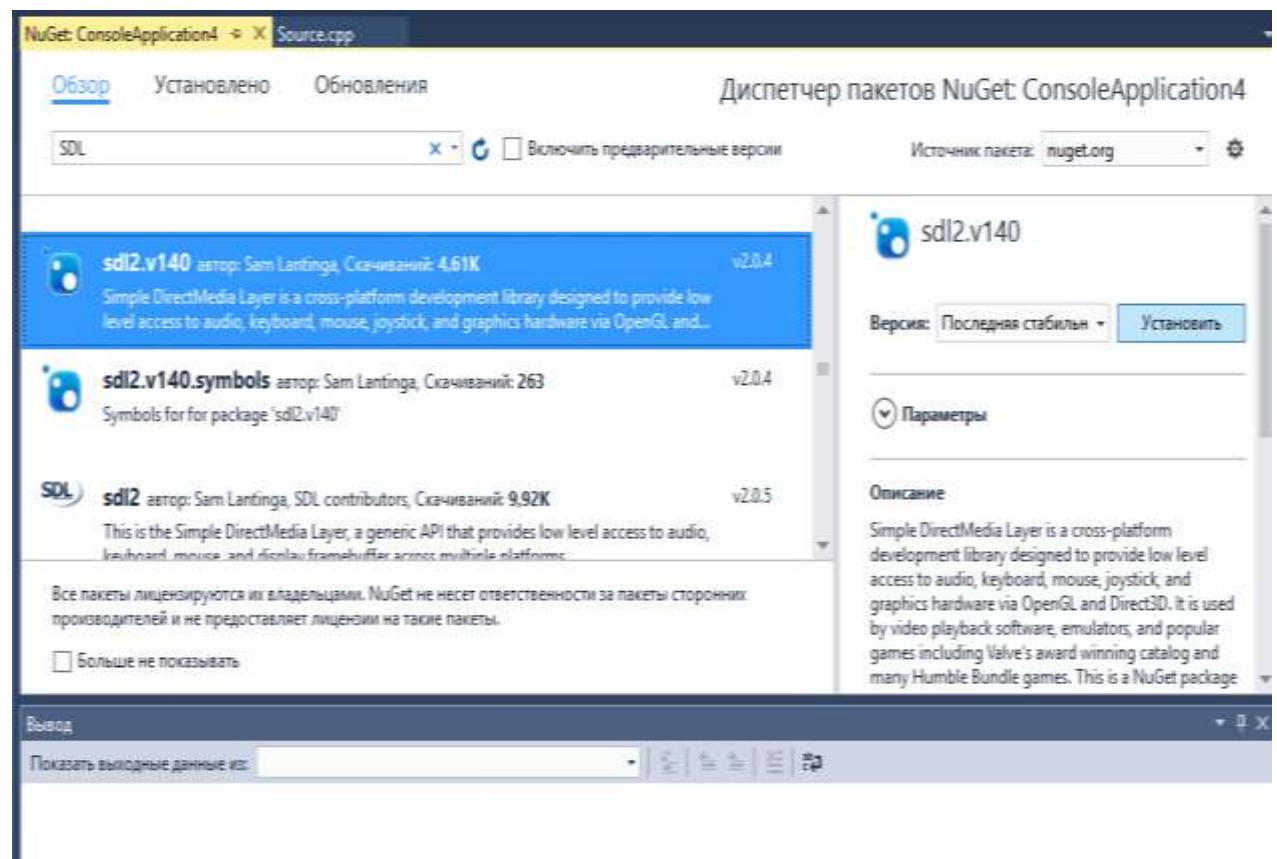


Рис. 3. Установка библиотеки

```
#include "SDL.h"
#include "math.h"
int main(int argc, char** argv)
{
    SDL_Init(SDL_INIT_EVERYTHING);
    SDL_Window* window = SDL_CreateWindow(u8"Окно",
        100, 100,
        800, 600, SDL_WINDOW_SHOWN);
    SDL_Renderer* renderer = SDL_CreateRenderer(window, -1, 0);

    SDL_SetRenderDrawColor(renderer, 10, 10, 50, 0);
    SDL_RenderClear(renderer); SDL_RenderPresent(renderer);
    SDL_Delay(1000);
    SDL_Event event;
    bool quit = false;

    while (!quit)
    {
        SDL_PollEvent(&event);
        if (event.type == SDL_QUIT)
            quit = true;

    }
    SDL_Delay(1000);
    SDL_DestroyRenderer(renderer);
    SDL_DestroyWindow(window);
    SDL_Quit();
    return 0;
}
```

Рис. 4. Код программы

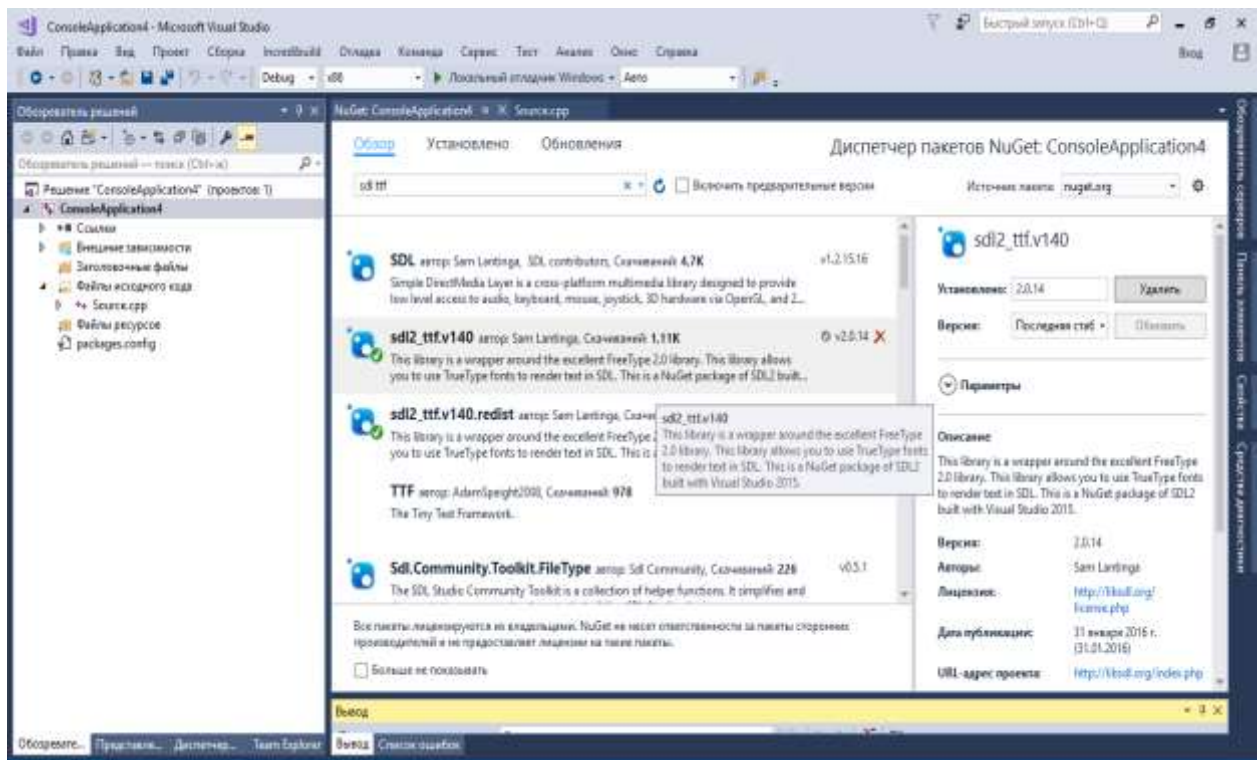


Рис. 5. Подключение библиотеки для работы со шрифтами

Создадим функцию, которой будем передавать окно, адрес рендера и текст:

```
void draw_text(SDL_Window* window, SDL_Renderer*
&renderer, char*text) {}
```

В ней разместим следующий код:

```
TTF_Init();
TTF_Font* my_font =TTF_OpenFont("Text.ttf", 100);
SDL_Color fore_color = { 130,140,50 };
SDL_Color back_color = { 188,155,166 };
SDL_Surface* textSurface = NULL;
textSurface = TTF_RenderText_Shaded(my_font, text,
fore_color, back_color);
```

Первая строка кода инициализирует режим работы с текстом. Далее загружается шрифт (файл должен быть добавлен в проект), задаются цвет шрифта и фона, создается рабочая поверхность, в которую заносится текст.

Далее задаются координаты текста, создается текстура, в которую преобразуется поверхность. Эта текстура передается в рендер для отрисовки. В заключение освобождается память.


```

SDL_Rect rect = { 0,0, 30, 30 };
SDL_Texture *texture =
SDL_CreateTextureFromSurface(renderer, textSurface);
SDL_RenderCopy(renderer, texture, NULL, &rect);

TTF_Quit();

```

Опишем функцию рисования шара:

```

void draw_ball(SDL_Renderer* &renderer, SDL_Rect rect)
{
    SDL_Surface *myImage = SDL_LoadBMP("1.bmp");
    SDL_SetColorKey(myImage, SDL_TRUE,
SDL_MapRGB(myImage->format, 255, 255, 255));

    SDL_Texture *texture =
SDL_CreateTextureFromSurface(renderer, myImage);
    SDL_RenderCopy(renderer, texture, NULL, &rect);
}

```

На вход будут передаваться координаты и размер прямоугольной области, в которой будет производиться рисование и рендер.

Внутри функции производится загрузка изображения, установка прозрачности, преобразование поверхности с изображением в текстуру и наложение текстуры на рендер.

Внутри главной функции программы создадим массив прямоугольников, задающих положение шаров, и циклически выведем изображения на экран. Цикл должен располагаться до игрового цикла:

```

SDL_Rect *ball = new SDL_Rect[5];
for (int i = 0; i < 5; i++)
{
    ball[i] = { i*50 + 110, i*50 + 10, 50+10*i, 50 + 10*i};
    draw_ball(renderer, ball[i]);
}
SDL_RenderPresent(renderer);

```

Теперь напишем код обработчика щелчка левой кнопкой мыши, в котором необходимо проверить попадание координат мыши в прямоугольник с шаром.

При выполнении условия размеры прямоугольника уменьшатся до нулей, экран очистится и прямоугольники перерисуются снова.

Этот код должен содержаться в игровом цикле:

```
if (event.button.button==SDL_BUTTON_LEFT)
    for (int i = 0; i < 5; i++)
    {
        if (event.button.x >= ball[i].x &&
event.button.x<=ball[i].x + ball[i].w &&
event.button.y>=ball[i].y && event.button.y <= ball[i].y
+ ball[i].h)
        {
            ball[i].w = 0;
            ball[i].h = 0;
            SDL_RenderClear(renderer);
            for (int j = 0; j < 5; j++)
            {
                draw_ball(renderer, ball[j]);
            }
            SDL_RenderPresent(renderer);
        }
    }
```

Осталось осуществить подсчет удаленных шаров. Внутри main() заведем целочисленную переменную `int k=0;` которую будем увеличивать каждый раз, когда будет выполняться условие нажатия на шарик. Также заведем строковую переменную `char *text=new char[10];`

До начала игрового цикла выведем количество очков, равное нулю. Преобразуем целочисленное значение переменной `k` в строку (потребуется `#include "stdlib.h"`):

```
itoa(k, text, 10);
```

Вызовем функцию отрисовки текста для полученного значения:

```
draw_text( window,  renderer, text);
SDL_RenderPresent(renderer);
```

При выполнении условия после увеличения счетчика аналогично выведем счет:

```
itoa(k, text, 10);  
draw_text(window, renderer, text);  
SDL_RenderPresent(renderer);
```

Результат запуска программы представлен на рис. 6.

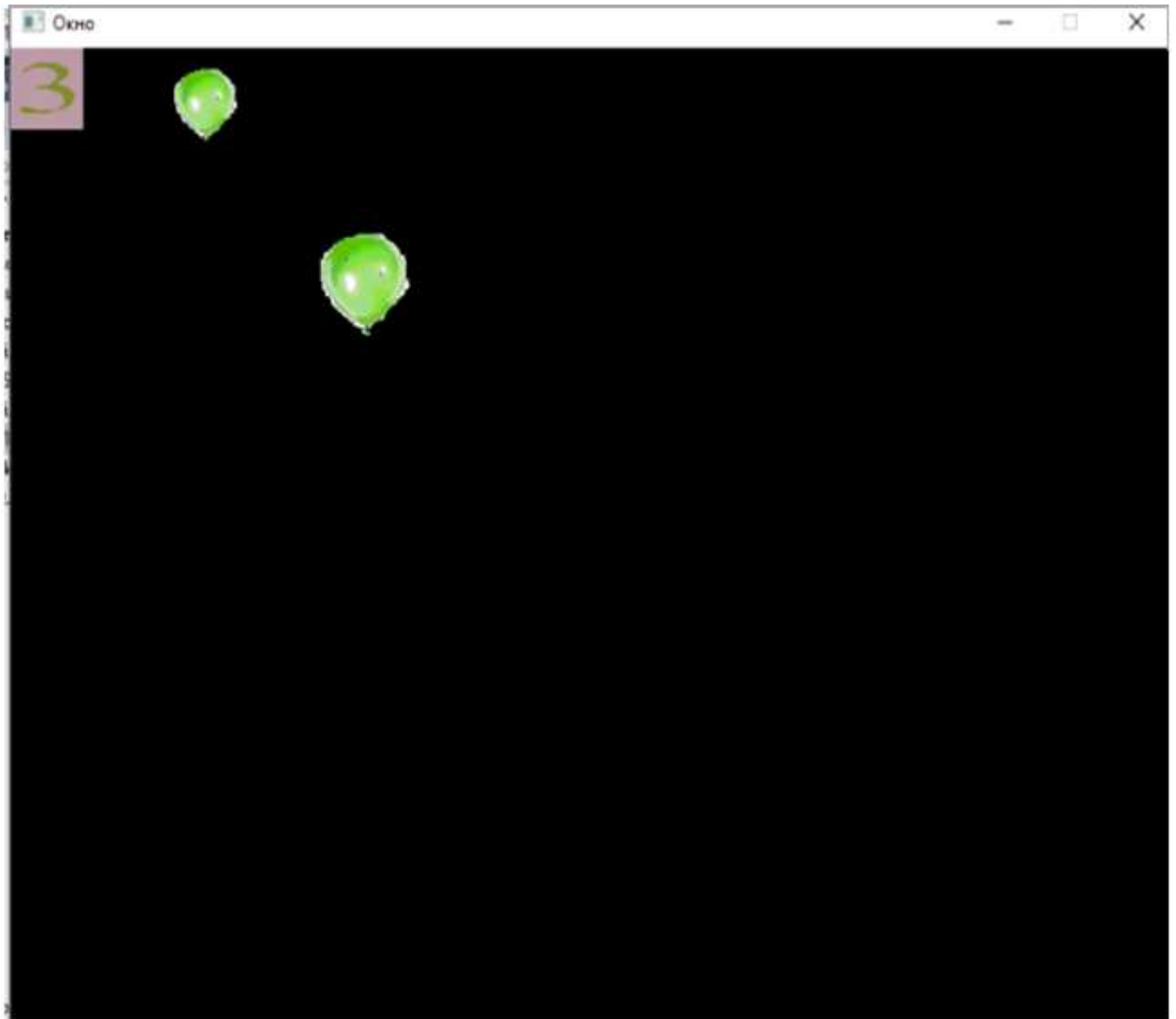


Рис. 6. Результат запуска программы

4. Задачи для самостоятельного решения

Задача 1. Модифицировать программу, описанную выше, таким образом, чтобы шары отображались в случайной позиции, а за их удаление назначалось различное количество очков. Предусмотреть вывод стоимости удаления на шаре, использовать собственные текстуры.

Задача 2. Реализовать следующие модификации программы в соответствии с вариантом ($N_{\text{вар}} = N_{\text{спис}} \% 7 + 1$):

1. Шары статичны, но щелчком мыши выбирается шар, который получает способность передвигаться при нажатии стрелок на клавиатуре; его цвет меняется; другие шары при столкновении с ним удаляются.

2. Шары движутся по синусоиде из левого края экрана в правый. Размер шаров постоянно меняется.

3. Шары статичны, но шар можно выделить щелчком мыши. Выделенный шар движется вслед за курсором мыши до следующего щелчка, отменяющего выделение. При столкновении с другими шарами выделенный шар удаляет их.

4. Шары статичны, но есть объект, передвигаемый клавиатурой по окну и удаляющий шар при столкновении. Предусмотреть возможность появления нового шара с другими характеристиками и внешним видом, но недоступного для удаления.

5. Шары движутся по круговой орбите с различным радиусом и центром. Размер шаров постоянно меняется.

6. Шары постоянно меняют свое местоположение случайным образом. Время смены местоположения шаров различно. Предусмотреть неналожение шаров друг на друга.

7. Шары не статичны, появляются по очереди и падают сверху вниз. Предусмотреть возможности ловить их полосой, без возможности поднимать ее с нижнего края экрана (как в арканоеде, но шары не отскакивают).

5. Контрольные вопросы

1. Для чего предназначена библиотека SDL?
2. Из каких этапов состоит создание простейшего приложения с использованием графики?
3. Из каких этапов состоит отображение простейшей анимации средствами SDL?
4. В чем заключается принцип разработки интерактивной анимации средствами SDL?

ГЛАВА 2. ДИНАМИЧЕСКИЕ МАССИВЫ

Лабораторная работа №7. Динамические двумерные массивы в непрерывной памяти (с постоянной длиной строки и с пересчётом индексов вручную)

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков использования динамических двумерных массивов в непрерывной памяти для решения задач.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Стандартные функции динамического выделения памяти

Функции динамического выделения памяти находят в оперативной памяти непрерывный участок требуемой длины и возвращают начальный адрес этого участка.

Функции динамического распределения памяти [1]:

```
void* malloc(РазмерМассиваВБайтах) ;  
void* calloc(ЧислоЭлементов, РазмерЭлементаВБайтах) ;
```

Для использования функций динамического распределения памяти необходимо подключение библиотеки <malloc.h>:

```
#include <malloc.h>
```

Обе представленные функции в качестве значения возвращают нетипизированный указатель `void *`, поэтому требуется явное приведение типа возвращаемого значения. Для определения размера массива в байтах, используемого в качестве аргумента функции `malloc()`, требуется количество элементов умножить на размер одного элемента. Не рекомендуется предполагать размер элемента. Для точного определения размера элемента в общем случае рекомендуется использовать операции `sizeof`, которая определяет количество байт, занимаемое элементом указанного типа.

Память, динамически выделенная с использованием функций `calloc()` или `malloc()`, после того, как станет не нужна программе,

может и должна быть освобождена с помощью функции `free(указатель)`. Если программист этого не сделал сам, то при завершении программы вся её динамическая память будет освобождена автоматически. Однако «правилом хорошего тона» в программировании является освобождение динамически выделенной памяти даже в самом конце программы.

Динамическое выделение памяти для двумерных массивов в виде одномерного массива

Двумерная матрица будет располагаться в оперативной памяти в форме ленты, состоящей из элементов строк. При этом индекс любого элемента двумерной матрицы можно получить по формуле

$$\text{index} = i \cdot m + j;$$

где i – номер текущей строки; j – номер текущего столбца.

Объем памяти, требуемый для размещения двумерного массива, определится как $n \cdot m \cdot (\text{размер элемента})$

Однако при таком объявлении компилятору явно не указывается количество элементов в строке и столбце двумерного массива, поэтому традиционное обращение к элементу путем указания индекса строки и индекса столбца является некорректным: `a[i][j]` – некорректно [1].

Правильное обращение к элементу с использованием указателя будет выглядеть как `*(p+i*m+j)`,

где

- p – указатель на массив,
- m – количество столбцов,
- i – индекс строки,
- j – индекс столбца.

Пример

Пусть необходимо организовать ввод и вывод значений динамического двумерного массива. Программа, реализующая данные действия, может иметь вид

```
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
int main()
```

```

{
    int *a;    // указатель на массив
    int i, j, n, m;

    printf("Введите количество строк: ");
    scanf("%d", &n);
    printf("Введите количество столбцов: ");
    scanf("%d", &m);

    // Выделение памяти
    a = (int*)malloc(n*m * sizeof(int));

    // Ввод элементов массива
    for (i = 0; i<n; i++)    // цикл по строкам
    {
        for (j = 0; j<m; j++)    // цикл по столбцам
        {
            printf("a[%d][%d] = ", i, j);
            scanf("%d", (a + i*m + j));
        }
    }

    // Вывод элементов массива
    for (i = 0; i<n; i++)    // цикл по строкам
    {
        for (j = 0; j<m; j++)    // цикл по столбцам
        {
            printf("%5d ", *(a + i*m + j)); // 5 знакомест под
элемент массива
        }
        printf("\n");
    }
    free(a);
    getchar();
    return 0;
}

```

Результат выполнения программы представлен на рис. 1.

```

C:\WINDOWS\system32\cmd.exe
Введите количество строк: 3
Введите количество столбцов: 4
a[0][0] = 1
a[0][1] = 2
a[0][2] = 3
a[0][3] = 4
a[1][0] = 5
a[1][1] = 6
a[1][2] = 7
a[1][3] = 6
a[2][0] = 5
a[2][1] = 4
a[2][2] = 5
a[2][3] = 4
  1   2   3   4
  5   6   7   6
  5   4   5   4
Для продолжения нажмите любую клавишу . . .

```

Рис. 1. Результат выполнения программы

Создание динамического массива с помощью массива указателей

Возможен также другой способ динамического выделения памяти под двумерный массив – с использованием массива указателей. Для этого необходимо [2]:

- выделить блок оперативной памяти под массив указателей;
- выделить блоки оперативной памяти под одномерные массивы, представляющие собой строки искомой матрицы;
- записать адреса строк в массив указателей.

При таком способе выделения памяти компилятору явно указано количество строк и количество столбцов в массиве.

Пример

Пусть необходимо организовать ввод и вывод значений динамического двумерного массива с использованием массива указателей. Программа, реализующая данные действия, может иметь вид

```

#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
int main()
{
    int **a; // указатель на указатель на строку элементов
    int i, j, n, m;
    printf("Введите количество строк: ");
    scanf("%d", &n);

```



```

printf("Введите количество столбцов: ");
scanf("%d", &m);

// Выделение памяти под указатели на строки
a = (int**)malloc(n * sizeof(int*));

// Ввод элементов массива
for (i = 0; i<n; i++) // цикл по строкам
{
    // Выделение памяти под хранение строк
    a[i] = (int*)malloc(m * sizeof(int));
    for (j = 0; j<m; j++) // цикл по столбцам
    {
        printf("a[%d][%d] = ", i, j);
        scanf("%d", &a[i][j]);
    }
}

// Вывод элементов массива
for (i = 0; i < n; i++) // цикл по строкам
{
    for (j = 0; j < m; j++) // цикл по столбцам
    {
        printf("%5d ", a[i][j]); // 5 знакомест под элемент
массива
    }
    printf("\n");
}

// Очистка памяти
for (i = 0; i < n; i++) // цикл по строкам
    free(a[i]); // освобождение памяти под строку
free(a);

getchar();
return 0;
}

```

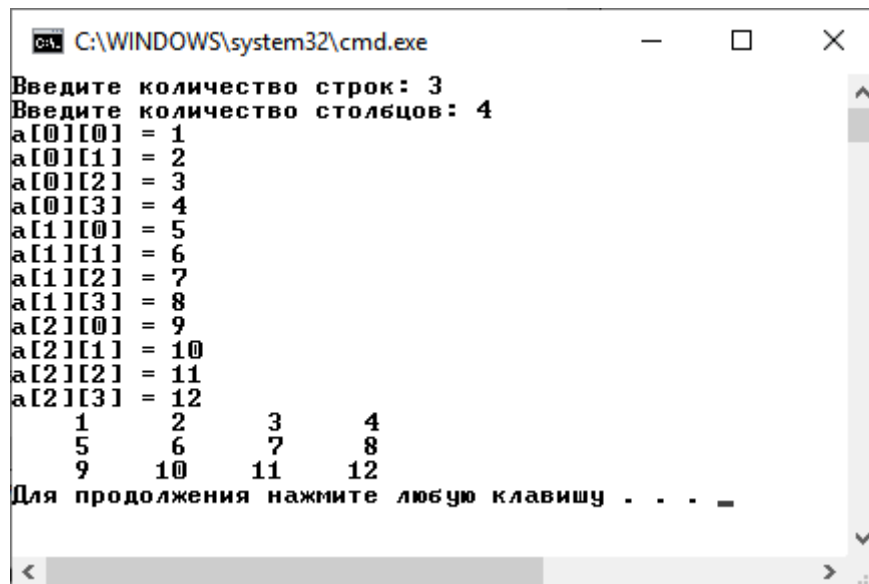
Результат выполнения программы представлен на рис. 2.

Перераспределение памяти

Если размер выделяемой памяти нельзя задать заранее, например при вводе последовательности значений до определенной

команды, то для увеличения размера массива при вводе следующего значения необходимо выполнить следующие действия [2]:

- Выделить блок памяти размерности $n+1$ (на 1 больше текущего размера массива).
- Скопировать все значения, хранящиеся в массиве во вновь выделенную область памяти.
- Освободить память, выделенную ранее для хранения массива.
- Переместить указатель начала массива на начало вновь выделенной области памяти.
- Дополнить массив последним введенным значением.



```

C:\WINDOWS\system32\cmd.exe
Введите количество строк: 3
Введите количество столбцов: 4
a[0][0] = 1
a[0][1] = 2
a[0][2] = 3
a[0][3] = 4
a[1][0] = 5
a[1][1] = 6
a[1][2] = 7
a[1][3] = 8
a[2][0] = 9
a[2][1] = 10
a[2][2] = 11
a[2][3] = 12
      1      2      3      4
      5      6      7      8
      9     10     11     12
Для продолжения нажмите любую клавишу . . . -
  
```

Рис. 2. Результат выполнения программы

Все перечисленные действия (кроме последнего) выполняет функция

```
void* realloc (void* ptr, size_t size);
```

где `ptr` – указатель на блок ранее выделенной памяти функциями `malloc()`, `calloc()` или `realloc()` для перемещения в новое место. Если этот параметр равен `NULL`, то выделяется новый блок, и функция возвращает на него указатель;

`size` – новый размер, в байтах, выделяемого блока памяти. Если `size = 0`, ранее выделенная память освобождается, и функция возвращает нулевой указатель, `ptr` устанавливается в `NULL`.

Размер блока памяти, на который ссылается параметр `ptr` изменяется на `size` байтов. Блок памяти может

уменьшаться или увеличиваться в размере. Содержимое блока памяти сохраняется, даже если новый блок имеет меньший размер, чем старый. Однако отбрасываются те данные, которые выходят за рамки нового блока памяти. Если новый блок памяти больше старого, то содержимое вновь выделенной памяти будет неопределенным.

«Свободные» массивы

С помощью динамического выделения памяти под указатели строк можно размещать свободные массивы. Свободным называется двухмерный массив (матрица), размер строк которого может быть различным.

Преимущество использования рваного массива заключается в том, что не требуется отводить память компьютера с запасом для размещения строки максимально возможной длины. Фактически рванный массив представляет собой одномерный массив указателей на одномерные массивы данных.

Для размещения в оперативной памяти матрицы со строками разной длины необходимо ввести дополнительный массив *m*, в котором будут храниться размеры строк (или осуществлять хранение размера в первом элементе массива).

3. Общая часть лабораторной работы

Задача 1. Выделить память для ввода массива целых чисел. После ввода каждого значения задавать вопрос о вводе следующего значения.

Решение. Для решения задачи при вводе каждого нового элемента массива будем производить перераспределение памяти с помощью функции `realloc()`.

Программа, реализующая решение этой задачи, может иметь вид

```
#include <stdio.h>
#include <malloc.h>
int main()
{
    int *a = NULL, i = 0, elem;
    char c;
    do {
        printf("a[%d]= ", i);
        scanf("%d", &elem);
        a = (int*)realloc(a, (i + 1) * sizeof(int));
```

```

    a[i] = elem;
    i++;
    getchar();
    printf("Next (y/n)? ");
    c = getchar();
} while (c == 'y');
for (int j = 0; j < i; j++)
    printf("%d ", a[j]);
getchar();
return 0;
}

```

Результат выполнения программы представлен на рис. 3.

```

C:\WINDOWS\system32\cmd.exe
a[0]= 1
Next (y/n)? y
a[1]= 4
Next (y/n)? y
a[2]= 6
Next (y/n)? y
a[3]= 8
Next (y/n)? y
a[4]= 8
Next (y/n)? y
a[5]= 5
Next (y/n)? y
a[6]= 2
Next (y/n)? y
a[7]= 33
Next (y/n)? y
a[8]= 2
Next (y/n)? y
a[9]= 6
Next (y/n)? n
1 4 6 8 8 5 2 33 2 6 Для продолжения нажмите любую клавишу

```

Рис. 3. Результат выполнения программы

Задача 2. Создать динамический массив с различным количеством элементов в строках.

Решение. Для решения задачи создадим массив указателей. Для каждого из элементов массива указателей выделим память в соответствии с количеством элементов в строке.

Программа, реализующая решение этой задачи, может иметь вид

```

int main()
{ int **a;
  int i, j, n, *m;
  system("chcp 1251");
  system("cls");

```

```

printf("Введите количество строк: ");
scanf("%d", &n);
a = (int**)malloc(n * sizeof(int*));
m = (int*)malloc(n * sizeof(int)); // массив кол-ва
элементов в строках массива a
// Ввод элементов массива
for (i = 0; i<n; i++)
{ printf("Введите количество столбцов строки %d: ", i);
  scanf("%d", &m[i]);
  a[i] = (int*)malloc(m[i] * sizeof(int));
  for (j = 0; j<m[i]; j++) {
    printf("a[%d][%d]= ", i, j);
    scanf("%d", &a[i][j]);
  }
}
// Вывод элементов массива
for (i = 0; i<n; i++)
{ for (j = 0; j<m[i]; j++)
  { printf("%3d ", a[i][j]);
  }
  printf("\n");
}
// Освобождение памяти
for (i = 0; i < n; i++)
{
  free(a[i]);
}
free(a);
free(m);
getchar(); getchar();
return 0;
}

```

Результат выполнения программы представлен на рис. 4.

4. Индивидуальная часть лабораторной работы

В индивидуальной части лабораторной работы написать и отладить программу на языке Си, которая должна решать задачу согласно варианту из списка задач для самостоятельного решения. Программа должна сформировать в памяти динамический двумерный массив (двумя возможными способами), заполнить его значениями с клавиатуры, вывести массив на экран, а также выполнить действия согласно варианту.

Также программа должна быть снабжена комментариями и отформатирована.

Для получения дополнительных навыков студентам предлагается по желанию решить задачи повышенной сложности.

Номер варианта задачи соответствует увеличенному на единицу остатку от деления номера студента в списке группы на 9 ($N_{\text{вар}}=1+N_{\text{студ}}\%9$).

```

C:\WINDOWS\system32\cmd.exe
Введите количество строк: 5
Введите количество столбцов строки 0: 6
a[0][0]= 1
a[0][1]= 2
a[0][2]= 3
a[0][3]= 4
a[0][4]= 5
a[0][5]= 6
Введите количество столбцов строки 1: 2
a[1][0]= 1
a[1][1]= 2
Введите количество столбцов строки 2: 1
a[2][0]= 1
Введите количество столбцов строки 3: 4
a[3][0]= 1
a[3][1]= 2
a[3][2]= 3
a[3][3]= 4
Введите количество столбцов строки 4: 3
a[4][0]= 1
a[4][1]= 2
a[4][2]= 3
  1  2  3  4  5  6
  1  2
  1
  1  2  3  4
  1  2  3
  
```

Рис. 4. Результат выполнения программы

5. Задачи для самостоятельного решения

1. Вычислить среднее арифметическое неотрицательных элементов матрицы, а также определить, сколько таких элементов в каждой отдельно взятой строке матрицы.

2. Вычислить среднее арифметическое k -й строки и сумму элементов k -го столбца.

3. Вычислить сумму наибольшего и наименьшего элементов для каждой строки.

4. Вычислить сумму каждого столбца. Среди полученных сумм найти максимальное значение.

5. Вычислить сумму каждой строки. Среди полученных сумм найти максимальное значение.

6. В полученной матрице вычислить среднее арифметическое элементов столбца, в котором находится максимальный элемент матрицы В.

7. Определить число и сумму элементов матрицы, лежащих вне интервала (А;В).

8. Определить количество отрицательных чисел в каждой строке.

9. Найти максимальный среди минимальных значений по строке.

Задачи повышенной сложности

1. Записать в массив последовательно вводимые оценки студентов по предмету «Программирование». Ввод прекратить при вводе нуля. Определить средний балл в группе по предмету, количество неудовлетворительных оценок и количество отличников.

2. В течение полугода в каждый из месяцев производится определенное количество замеров температуры воздуха (количество замеров известно заранее, но в каждом месяце не известно). Определить, в каком из месяцев разница между максимальной и минимальной температурой самая большая.

3. В группе определенное количество студентов. Каждый из студентов имеет оценки. Количество оценок заранее не известно, но ввод оценок студента завершается вводом нуля. Определить средний балл для каждого из студентов, средний балл в группе, количество студентов, имеющих менее трех оценок и количество студентов, у которых только хорошие оценки (4 и 5).

4. Пусть дан двумерный массив из n строк, длина строк переменная и вводится пользователем вручную. Поменять местами строки с наибольшим и наименьшим элементом массива.

6. Контрольные вопросы

1. Какие функции для динамического выделения памяти существуют?

2. Какие способы создания динамического массива существуют?

3. В чем заключается перераспределение памяти?

4. Что такое «свободные» массивы?

Лабораторная работа №8. Рваные массивы

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков работы с рваными массивами.

Для этого необходимо:

- изучить понятие рваного массива;
- изучить приёмы представления рваного массива в оперативной памяти ЭВМ;
- изучить различные способы хранения данных о количестве строк рваного массива и количестве элементов в его строках;

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Понятие рваного массива

По определению, массив – это упорядоченная совокупность элементов одного и того же типа, доступ к которым осуществляется по порядковому номеру (индексу). Во многих языках программирования (в том числе и в языке Си) элементы массива располагаются в непрерывной памяти ЭВМ. Это свойство массива позволяет вычислить адрес любого элемента, зная адрес начала массива (адрес его первого элемента) и размер каждого элемента (он постоянен по массиву).

Двумерный массив в языке Си представляется как массив массивов [1]. В памяти он хранится строчка за строчкой. Компилятор при обращении к элементу такого массива по двум индексам (номеру строки и номеру столбца) обеспечивает вычисление адреса искомого элемента, но это достигается только при условии, что на стадии компиляции известна длина строки массива и эта длина постоянна (то есть все строки имеют одинаковую длину).

Эти ограничения могут быть сняты путём использования *рваного массива*. Часто вместо слова «рваный» используют слово «*ступенчатый*» или «*зубчатый*».

Итак, *рванным* называют такой двумерный массив, строки которого не лежат друг за другом в непрерывной области памяти и/или строки которого могут отличаться друг от друга по длине [2].

Для удобной и эффективной работы с рваным массивом желательно обеспечить возможность доступа к элементам массива с помощью обычной двойной индексации.

Представление рваного массива в оперативной памяти ЭВМ

Можно выделить два основных способа представления рваного массива в оперативной памяти ЭВМ [1]:

- Память выделяется единым блоком для хранения элементов всех строк, а также всех вспомогательных конструкций.
- Память выделяется отдельно для каждой строки и для вспомогательных конструкций.

Независимо от способа выделения памяти обычно доступ к элементам массива обеспечивает вектор указателей на строки. Вспомогательные конструкции рваного массива можно для удобства объединить структурой – дескриптором рваного массива

Можно выделить следующие основные способы хранения данных о количестве строк рваного массива и количестве элементов в его строках [3]:

1. Хранить длину последовательности в элементе с индексом -1 самой этой последовательности (как вектора указателей на строки, так и самих строк).
2. Хранить отдельно количество строк и массив длин строк.
3. Вместо длины последовательности хранить признак её завершения, или так называемый терминальный символ.

Каждый из этих способов имеет свои преимущества и недостатки. С их учётом в зависимости от условий применения рваного массива можно выбирать и комбинировать различные способы хранения данных о количестве строк рваного массива и количестве элементов в его строках.

3. Индивидуальная часть лабораторной работы

В индивидуальной части лабораторной работы необходимо спроектировать рваный массив с учётом требований индивидуальной задачи и представить на проверку следующие результаты:

1. Диаграмму распределения памяти.
2. Описание необходимых структур данных.

3. Описание последовательности шагов для следующих действий:

- создание массива;
- доступ к элементам;
- доступ к размерам массива (количество строк и длина произвольной строки);
- сохранение в текстовый файл;
- сохранение в бинарный файл;
- создание из данных в текстовом файле;
- создание из данных в бинарном файле;
- уничтожение массива.

4. Описание преимуществ, недостатков и условий применения разработанного массива.

Номер варианта задачи соответствует увеличенному на единицу остатку от деления номера студента в списке группы на 8 ($N_{\text{вар}}=1+N_{\text{студ}}\%8$).

4. Задачи для самостоятельного решения

0. [Не соответствует ни одному номеру в списке группы, предназначен для разбора в качестве примера] Массив представляется указателем на вектор указателей на строки. Количество строк хранится в -1 элементе вектора указателей. Количество элементов строки хранится в -1 элементе строки. Память выделяется для каждой конструкции отдельно.

1. Массив представляется указателем на вектор указателей на строки. Количество строк хранится в -1 элементе вектора указателей. Количество элементов строки хранится в -1 элементе строки. Память выделяется одним блоком.

2. Массив представляется указателем на вектор указателей на строки. Количество строк хранится в -1 элементе вектора указателей. Количество элементов строки определяется терминальным символом. Память выделяется одним блоком.

3. Массив представляется указателем на вектор указателей на строки. Количество строк определяется терминальным символом. Количество элементов строки определяется терминальным символом. Память выделяется одним блоком.

4. Массив представляется указателем на вектор указателей на строки. Количество строк хранится в -1 элементе вектора указателей. Количество элементов строки определяется терминальным символом. Память выделяется для каждой конструкции отдельно.

5. Массив представляется указателем на вектор указателей на строки. Количество строк определяется терминальным символом. Количество элементов строки определяется терминальным символом. Память выделяется для каждой конструкции отдельно.

6. Массив представлен дескриптором. Количество строк определяется полем дескриптора. Количество элементов строк определяется динамическим массивом размеров.

7. Массив представлен дескриптором. Количество строк определяется полем дескриптора. Количество элементов строки определяется терминальным символом.

8. Массив представлен дескриптором. Количество строк определяется полем дескриптора. Количество элементов строки хранится в -1 элементе строки.

5. Контрольные вопросы

1. Что такое рваный массив?
2. Перечислите основные способы представления рваного массива в оперативной памяти ЭВМ.
3. Каковы их преимущества и недостатки?
4. Перечислите основные способы хранения данных о количестве строк рваного массива и количестве элементов в его строках. Каковы их преимущества и недостатки?

Лабораторная работа №9. Организация библиотеки для работы с «рванными» массивами

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков выполнения программной реализации библиотеки для работы с рванным массивом. Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Многофайловый проект

Большинство современных программных продуктов невозможно разработать и отладить, размещая все функции проекта в одном файле. Ранние версии компиляторов даже имели ограничение на размер файла исходного текста в 64 килобайта. Вследствие этого компиляторы всех фирм без исключения имеют средства модульного программирования, позволяющие разрабатывать, тестировать и отлаживать проект по частям, разбивая его на модули. Модуль представляет собой наименьшую неделимую часть программы. Модуль только целиком может быть добавлен к программе и только целиком из нее удален, даже когда он содержит функции, к которым нет обращений [2].

Каждый файл исходного текста программы на языке Си после компиляции образует отдельный модуль. Программа собирается из отдельных модулей, находящихся на диске, и библиотечных модулей с помощью специальной программы, называемой компоновщиком или редактором связей. Компоновщик можно вызвать явно из командной строки или неявно из интегрированной среды разработки программ. Имена всех модулей и библиотек функций, используемых в проекте, заносятся в файл проекта с помощью средств конкретной системы программирования и, в дальнейшем, среда разработки программ автоматически отслеживает все изменения в исходных файлах проекта, автоматически компилирует и собирая новые версии программы.

Библиотеки функций

При разработке программных проектов, состоящих из большого числа модулей, целесообразно пользоваться библиотеками модулей.

Каждая такая библиотека обычно имеет расширение *.lib и может быть добавлена в проект с помощью средств поддержки проекта.

Особенностью использования библиотек является то, что компоновщик будет обращаться к библиотеке и выбирать оттуда необходимые модули только по мере необходимости, то есть если имеются ссылки к функциям библиотечного модуля.

Необходимо помнить, что модуль будет выбран из библиотеки и подключен к программе всегда целиком, даже если программе требуется всего только одна функция.

В лабораторной работе под библиотекой будем понимать совокупность связанных между собой типов данных и обрабатывающих их функций.

В языке Си обычно выделяют интерфейс библиотеки. Он содержит описание пользовательских типов данных, объявление глобальных переменных модуля в качестве внешних (*extern*) переменных и прототипы функций модуля. Для исключения повторного описания типов используют директивы условной компиляции. Интерфейс библиотеки помещают в заголовочном файле (*.h).

Реализацию библиотеки помещают в файл с таким же именем, но расширением «*.c» или «*.cpp». Реализация библиотеки содержит описание глобальных переменных модуля и описание функций[2].

3. Общая часть лабораторной работы

Пример

Пусть необходимо реализовать библиотеку для работы с рваным массивом.

Иногда бывает с некоторой совокупностью однотипных данных удобно работать как с двумерным массивом, но количество элементов в разных его строках различно. Тогда этот массив можно организовать как «рванный» массив.

Создадим в проекте заголовочный файл DynArray.h и разместим там код, представленный в следующем листинге.

```
#ifndef DYNARRAY_H
#define DYNARRAY_H
// Прототипы своих функций
void PrintArray(double **p);
double ** ConstructArray(int m, int *n);
void DestructArray(double **p);
```

```
#endif
```

Интерфейс библиотеки представим в виде ряда функций: ConstructArray (создание массива и ввод его параметров и значений), DestructArray (освобождение памяти), PrintArray (выводит массив на экран).

Функция PrintArray выводит массив на экран, получая указатель на него в списке аргументов. Вся дополнительная информация (количество строк, количество элементов в каждой строке) извлекается также по этому указателю.

Создадим в проекте файл исходного кода DynArray.cpp и разместим там код, представленный в следующем листинге.

```
#include "DynArray.h"
#include <stdlib.h>
#include "stdio.h"
#include <conio.h>

//*****
// Функция запрашивает необходимую информацию и строит
// массив.
// Возвращает указатель на него. Количество строк
// хранится в
// «минус первом» элементе вектора указателей на строки,
// а в
// «минус первом» элементе каждой строки – количество.
// Элементов
// в строке
double ** ConstructArray(int m, int *n)
{
    double ** p=NULL;
    // выделяем память для вектора указателей
    p=(double **)malloc( sizeof(double *)*m + sizeof(int));
    // запоминаем количество строк
    int *p1=(int*) p;
    *p1=m;
    // сдвигаем указатель, теперь количество строк
    // оказывается в «минус первом» // элементе вектора
    // указателя на строки
    p1++
    p=(double **) p1;
```

```

// Организуем построчное выделение памяти
for(int i=0; i<m; i++)
{
    // выделяем память
    p[i]=(double *)malloc( n[i]*sizeof(double) +
sizeof(int));
    // запоминаем количество элементов в данной строке
    p1=(int*)p[i];
    *p1=n[i];
    // сдвигаем указатель, теперь количество элементов в
данной строке
    // оказывается в «минус первом» элементе данной строки
    p1++;
    p[i]=(double *)p1;

    // заполним нулями
    for(int j=0; j<n[i]; j++) // вводим элементы
    {
        p[i][j]=0;
    }
}
return p; // вернуть указатель на сконструированный
массив
}

void DestructArray(double **p)
{
    if(!p) return;// проверка корректности
    int m; // количество строк
    int *p1;
    p1=(int*)p;
    m=p1[-1];
    for(int i=0; i<m; i++)
    {
        int *p2=(int*) p[i];
        p2--;
        free(p2)]; // освобождаем память от текущей строки
    }
    p1--;
    free(p1); // освобождаем память от вектора указателей
на строки
}

//*****
*****

```

```
// Функция выводит на экран рванный массив
void PrintArray(double **p)
{
    int n, m; // количество строк, столбцов
    int *p1;
    p1=(int*)p;
    m=p1[-1];
    for(int i=0; i<m; i++)
    {
        int *p2=(int *) p[i];
        n=p2[-1];
        for(int j=0; j<n; j++) printf("%-6.2lf", p[i][j]);
        printf("\n");
    }
}
```

Для удобства отладки и проверки количество строк и столбцов, а также значение элемента массива должны лежать в пределах от 1 до 10. Необходимо иметь в виду, что при небольших размерах массива накладные расходы и трудоемкость организации «рванных» массивов, наверняка, превзойдут положительный эффект от его использования. Положительный эффект становится ощутимым при больших размерах массива.

Код демонстрации библиотеки разместим в главной функции программы, как показано в следующем листинге.

```
#include"DynArray.h"
#include<stdlib.h>
#include"stdio.h"
#include<conio.h>
//*****
int void main(void)
{
    // указатель на вектор указателей на строки
    double **pA;
    int m=5, n[5]=[1,2,3,4,5];
    // построит массив и вернет указатель на него
    pA=ConstructArray(5, n);
    // выведет массив на экран
    PrintArray(pA);
    // освободит память
    DestructArray(pA);
    return 0;
}
```


4. Индивидуальная часть лабораторной работы

В индивидуальной части лабораторной работы выполнить программную реализацию на языке Си библиотеки для работы с вьюемым массивом, спроектированным в лабораторной работе №8. Кроме самой библиотеки разработать демонстрационную программу, с помощью которой пользователь может убедиться в работоспособности и корректной работе библиотеки. Индивидуальная задача определяется задачей лабораторной работы №8.

5. Контрольные вопросы

1. Что такое модуль в языке Си?
2. Что такое библиотека в языке Си?
3. Что содержит интерфейс библиотеки?
4. Что содержит реализация библиотеки?
5. Какое расширение файла обычно используется для обозначения файла с интерфейсом библиотеки?
6. Какое расширение файла обычно используется для обозначения файла с реализацией библиотеки?

ГЛАВА 3. РАБОТА СО СТРОКАМИ

Лабораторная работа №10. Работа со строками

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков работы со строками как с массивами символов.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

В компьютере все значения хранятся в виде чисел. И строки тоже, там нет никаких символов и букв. Строка представляет собой массив чисел. Каждое число соответствует определённому символу, который берётся из таблицы кодировки. При выводе на экран символ отображается определённым образом.

Для хранения строк используются массивы типа *char*. Тип *char* хранит один байт данных. В соответствии с таблицей кодировки каждое из этих чисел связано с символом. И в обратную сторону – каждый символ определяется своим порядковым номером в таблице кодировки.

Строка в Си – это массив типа *char*, последний элемент которого хранит терминальный символ '\0'. Числовое значение этого символа 0, поэтому можно считать, что массив оканчивается нулём [2].

Для ввода строки используется классификатор *%s*. При этом строка вводится до первого терминального символа, в том числе пробела.

Для вывода строки используется классификатор *%s*. При этом строка выводится до первого терминального символа [1].

Функции *gets()* и *puts()* позволяют читать и выводить строки на консоль.

Функция *gets()* читает строку символов, введенных с клавиатуры, и помещает их по адресу, указанному в аргументе. Можно набирать символы, пока не будет нажат ввод. Символ, соответствующий клавише ввод (возврат каретки), не станет частью строки. Вместо этого в конце строки появится нулевой символ, и *gets()* закончит работу.

Функция *gets()* имеет прототип
`char *gets(char *str);`

где `str` – это массив символов. Функция `gets()` возвращает указатель на `str`.

Следующая программа осуществляет чтение строки в массив `str`:

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char str[80];
    gets(str);
    return 0;
}
```

Функция ***puts()*** выводит передаваемый ей аргумент на экран, завершая вывод переходом на новую строку. Она имеет следующий прототип:

```
int puts(const char *str);
```

3. Общая часть лабораторной работы

Задача 1. Составить функцию для вычисления длины строки. На вход функция должна получать указатель на строку. В результате выполнения функции должно быть возвращено целое число.

Решение. Для решения задачи в теле функции будем последовательно сравнивать каждый символ с символом конца строки. Если конец строки на очередном символе не достигнут, длина строки будет увеличиваться на один.

```
int len(char *s) {
    l = 0;
    while (s[l] != '\0') {
        l++;
    }
    return l;
}
```

Задача 2. Составить программу для определения количества букв «а» во введенном тексте.

Решение. Программа, реализующая решение этой задачи, может иметь вид

```
#include <conio.h>
#include <stdio.h>
int len(char *s)
{
    int l = 0;
    while (s[l] != '\0') {
        l++;
    }
    return l;
}

void main()
{
    char s[256];    //Первое слово
    gets(s);
    int k=0;

    for (int i = 0; i < len(s); i++)
        if (s[i] =='a') k++;

    printf("%d", k);
    getch();
}
```

Результаты выполнения программы представлены на рис. 1 и рис. 2.

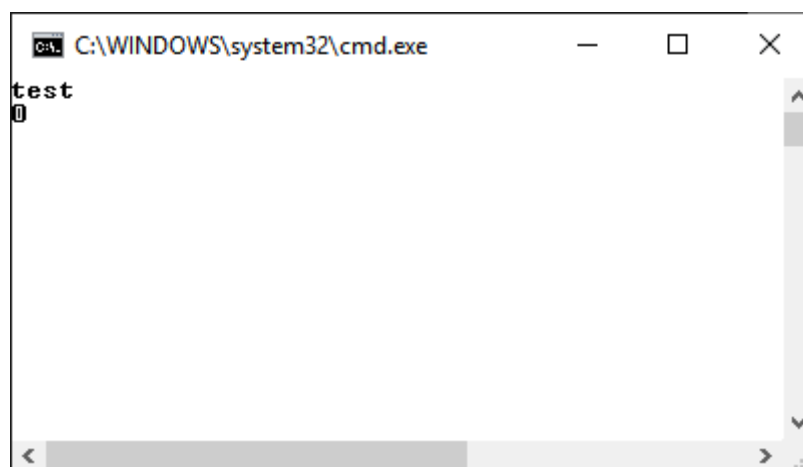


Рис. 1. Результат выполнения программы

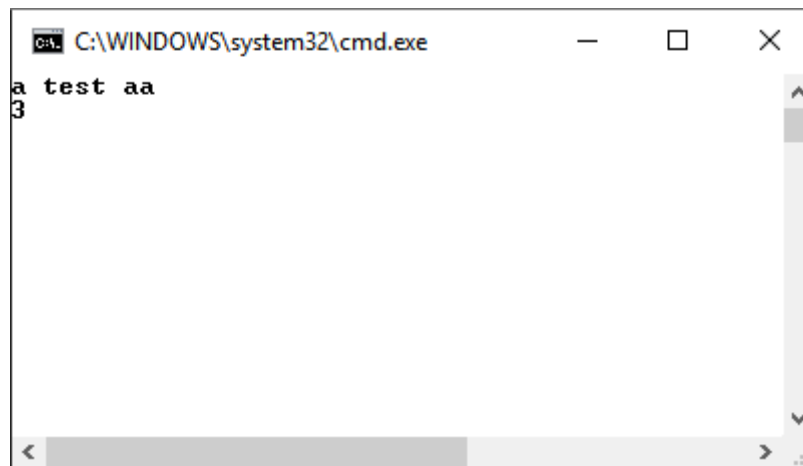


Рис. 2. Результат выполнения программы

4. Индивидуальная часть лабораторной работы

В индивидуальной части лабораторной работы написать и отладить три программы на языке Си, каждая из которых должна решать задачу согласно варианту из списка задач для самостоятельного решения. Варианты задач определяются следующим образом: N , $N+9$ и $N+18$, где $N_{\text{вар}} = 1 + N_{\text{студ}} \% 9$.

Каждая программа должна запросить у пользователя все необходимые исходные данные и вывести результаты на экран.

Программы должны быть снабжены комментариями и отформатированы.

5. Задачи для самостоятельного решения

1. Определить, сколько раз в тексте встречается заданная буква.
2. Определить количество слов в тексте и вывести каждое слово на новой строке.
3. Определить, сколько слов в тексте начинается на букву К или к.
4. Определить, сколько раз в тексте встречается заданное слово.
5. Удалить из текста все слова, в которых встречается не более двух различных букв.
6. Удалить из текста все слова, которые являются палиндромами.
7. Удалить из текста все цифры. Определить количество удаленных цифр.
8. В заданном тексте заменить заданную последовательность символов на другую заданную последовательность символов.
9. В заданном тексте удалить все части текста, заключенные в скобки (вместе со скобками).

10. Выяснить, верно ли, что каждое слово, не являющееся палиндромом, имеет четную длину.

11. В текст, содержащий меньше 50 символов, равномерно вставить пробелы между словами, чтобы его длина составляла ровно 50 символов.

12. Найти все прописные русские гласные буквы, которые содержатся в заданном тексте.

13. Найти все прописные русские буквы, которых не содержится в заданном тексте.

14. Дана строка. Определить общее количество символов '+' и '-' в ней, а также количество таких символов, после которых следует цифра ноль.

15. Дана строка. Определить, какой символ в ней встречается раньше: 'x' или 'w'. Если какого-то из символов нет, вывести сообщение об этом.

16. Удалить в строке все лишние пробелы, то есть серии подряд идущих пробелов заменить на одиночные пробелы. Крайние пробелы в строке удалить.

17. Дан текст. Вычислить сумму имеющихся в нем цифр.

18. Дан текст. Определить наибольшее количество подряд идущих пробелов в нем.

19. Даны два слова. Найти только те символы слов, которые встречаются в обоих словах только один раз.

20. В заданной строке вставить после каждого символа 'a' символ 'b'.

21. Даны две строки. Удалить в первой строке первое вхождение второй строки.

22. Строка состоит из слов, разделенных одним или несколькими пробелами. Определить слово наибольшей длины.

23. Дан email в строке. Определить, является ли он корректным (наличие символа '@' и точки, наличие не менее двух символов после последней точки и т.д.).

24. Дана строка. Вставить после каждого символа два случайных символа.

25. Даны два предложения. Для каждого слова первого предложения определить количество его вхождений во второе предложение.

26. Исключить из строки группы символов, расположенные между символами «/*», «*/» включая границы . Предполагается, что нет вложенных скобок.

27. Исключить из строки группы символов, расположенные между символами «/*», «*/» включая границы . Предполагается, что нет вложенных скобок.

6. Контрольные вопросы

1. Что такое строка?
2. Чем оканчивается строка?
3. Какие функции существуют для ввода и вывода строк?
4. Какая есть взаимосвязь между строкой и массивом?

Лабораторная работа №11. Стандартные функции для работы со строками

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков работы со строками с помощью стандартных функций.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

В компьютере все значения хранятся в виде чисел. И строки тоже, там нет никаких символов и букв. Строка представляет собой массив чисел. Каждое число соответствует определённому символу, который берётся из таблицы кодировки. При выводе на экран символ отображается определённым образом.

Для хранения строк используются массивы типа *char*. Тип *char* хранит один байт данных. В соответствии с таблицей кодировки каждое из этих чисел связано с символом. И в обратную сторону – каждый символ определяется своим порядковым номером в таблице кодировки.

Строка в Си – это массив типа *char*, последний элемент которого хранит терминальный символ '\0'. Числовое значение этого символа 0, поэтому можно считать, что массив оканчивается нулём [2].

Для ввода строки используется классификатор *%s*. При этом строка вводится до первого терминального символа, в том числе пробела.

Для вывода строки используется классификатор *%s*. При этом строка выводится до первого терминального символа [1].

Функции *gets()* и *puts()* позволяют читать и выводить строки на консоль.

Основные стандартные функции для работы со строками и символами приведены в табл. 1 [3].

Функции для работы со строками и символами

Функция	Пояснение
<u>strlen</u> (имя_строки)	Определяет длину указанной строки без учёта нуля-символа
Копирование строк	
<u>strcpy</u> (s1,s2)	Выполняет побайтное копирование символов из строки s2 в строку s1
<u>strncpy</u> (s1,s2, n)	Выполняет побайтное копирование n символов из строки s2 в строку s1. возвращает значение s1
Конкатенация строк	
<u>strcat</u> (s1,s2)	Объединяет строку s2 со строкой s1. Результат сохраняется в s1
<u>strncat</u> (s1,s2,n)	Объединяет n символов строки s2 со строкой s1. Результат сохраняется в s1
Сравнение строк	
<u>strcmp</u> (s1,s2)	Сравнивает строку s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2, <0 – если s1>s2 С учётом регистра
<u>strncmp</u> (s1,s2,n)	Сравнивает n символов строки s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2, <0 – если s1>s2 С учётом регистра
<u>stricmp</u> (s1,s2)	Сравнивает строку s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2, <0 – если s1>s2 Без учёта регистра
<u>strnicmp</u> (s1,s2,n)	Сравнивает n символов строки s1 со строкой s2 и возвращает результат типа int: 0 –если строки эквивалентны, >0 – если s1<s2, <0 – если s1>s2 Без учёта регистра

Функция	Пояснение
Обработка символов	
<u>isalnum(c)</u>	Возвращает значение true, если с является буквой или цифрой, и false в других случаях
<u>isalpha(c)</u>	Возвращает значение true, если с является буквой, и false в других случаях
<u>isdigit(c)</u>	Возвращает значение true, если с является цифрой, и false в других случаях
<u>islower(c)</u>	Возвращает значение true, если с является буквой нижнего регистра, и false в других случаях
<u>isupper(c)</u>	Возвращает значение true, если с является буквой верхнего регистра, и false в других случаях
<u>isspace(c)</u>	Возвращает значение true, если с является пробелом, и false в других случаях
<u>toupper(c)</u>	Если символ с, является символом нижнего регистра, то функция возвращает преобразованный символ с в верхнем регистре, иначе символ возвращается без изменений
Функции преобразования	
<u>atof(s1)</u>	Преобразует строку s1 в тип double
<u>atoi(s1)</u>	Преобразует строку s1 в тип int
<u>atol(s1)</u>	Преобразует строку s1 в тип long int
Функции стандартной библиотеки ввода/вывода <stdio>	
<u>getchar(c)</u>	Считывает символ с со стандартного потока ввода, возвращает символ в формате int
<u>gets(s)</u>	Считывает поток символов со стандартного устройства ввода в строку s до тех пор, пока не будет нажата клавиша ENTER

Функция	Пояснение
Функции поиска	
<u>strchr(s,c)</u>	Поиск первого вхождения символа с в строке s. При удачном поиске возвращает указатель на место первого вхождения символа с. Если символ не найден, то возвращается ноль
<u>strcspn(s1,s2)</u>	Определяет длину начального сегмента строки s1, содержащего те символы, которые не входят в строку s2
<u>strspn(s1,s2)</u>	Возвращает длину начального сегмента строки s1, содержащего только те символы, которые входят в строку s2
strprbk(s1,s2)	Возвращает указатель первого вхождения любого символа строки s2 в строке s1

3. Общая часть лабораторной работы

Разработать несколько программ, используя функции для работы со строками и символами.

Копирование строк

Задача 1. Составить программу для копирования строк.

Решение. Для решения задачи используются функции `strcpy` и `strncpy`. Программа, реализующая решение этой задачи, может иметь следующий вид:

```
#include "stdafx.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    char s2[27] = "Текст";           // инициализация
    строки s2
```

```

        char s1[27];                                // резервируем строку
для функции strcpy()
        cout << "strcpy(s1,s2) = " << strcpy(s1,s2) << endl;
// содержимое строки s2 скопировалось в строку s1,
возвращается указатель на s1
        cout << "s1=  " << s1      << endl; // вывод
содержимого строки s1
        char s3[7];                                // резервируем строку для
следующей функции
        cout << strncpy(s3, s2, 7)<< endl; // копируем 7
символов из строки s2 в строку s3
        system("pause");
        return 0;
}

```

Конкатенация строк

Задача 2. Составить программу для конкатенации строк.

Решение. Для решения задачи будем использовать функции `strcpy` и `strncpy`. Программа, реализующая решение этой задачи, может иметь вид

// str_cpy.cpp: определяет точку входа для консольного приложения.

```

#include "stdafx.h"
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
    char s2[27] = "Text";// инициализация строки s2
    char s1[27];    // резервируем строку для функции
strcpy()
    cout << "strcpy(s1,s2) = " << strcpy(s1,s2) << endl;
// содержимое строки s2 скопировалось в строку s1,
возвращается указатель на s1
    cout << "s1=  " << s1<< endl; // вывод содержимого
строки s1
    char s3[7];    // резервируем строку для
следующей функции
    cout << strncpy(s3, s2, 7 << endl; // копируем 7
символов из строки s2 в строку s3
    system("pause");
}

```

```
    return 0;  
}
```

4. Индивидуальная часть лабораторной работы

Взяв за основу свои программы, написанные для решения задач лабораторной работы №10, переработать их таким образом, чтобы решение задач происходило с использованием стандартных функций для работы со строками.

Каждая программа должна запросить у пользователя все необходимые исходные данные (с проверкой их корректности) и вывести результаты на экран.

Программы должны быть снабжены комментариями и отформатированы.

5. Контрольные вопросы

1. Что такое строка?
2. Чем оканчивается строка?
3. Какие существуют функции для ввода и вывода строк?
4. Какие существуют встроенные функции для работы со строками?
5. Какова связь между строкой и массивом?

ГЛАВА 4. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Лабораторная работа №12. Особенности передачи аргументов и возврата значений

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков программной реализации библиотеки для работы с рваным массивом.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Передача параметров в функцию

В правильно организованной функции использование списка аргументов (параметров) и возвращаемого значения является единственным способом связи этой функции с остальными.

При вызове функции в стеке создаются локальные переменные, соответствующие по типу формальным параметрам функции. Затем они инициализируются значениями фактических параметров, задаваемых при вызове функции в строгом соответствии с последовательностью формальных параметров. При необходимости производится допустимое преобразование типов. Соответствие фактических и формальных параметров устанавливается не по именам параметров, а по их местоположению в списках параметров [3].

Рассмотренный алгоритм вызова функции гарантирует сохранение значений фактических параметров независимо от того, что делала функция с соответствующими формальными параметрами.

Пример 1

Рассмотрим функцию Sum(), вычисляющую значение суммы элементов массива:

```
#include <stdio.h>
double Sum(double A[], int nA)
{   double s = 0;
    while(nA) s += A[--nA];
    return s;
}
void main (void)
```

```

{
    double B[] = { 1, 2, 3, 4, 5 };
    int nB = sizeof(B)/sizeof(B[0]);
    printf("Сумма = %lf\n", Sum(B,nB));
    printf("nB = %d\n", nB);
}

```

Функция при своей работе изменяет значение формального параметра `nA`, в котором передается размер массива. Однако если запустить программу, то можно увидеть, что значение соответствующего фактического параметра `nB` осталось неизменным вследствие того, что функция работает с копиями параметров.

При фактическом параметре-массиве в функцию передается значение его имени, то есть адрес его первого элемента, для которого создается копия в стеке, не являющаяся уже адресом-константой и допускающая изменение своего значения внутри функции. В связи с этим функцию `Sum` можно реализовать и следующим образом:

```

double Sum(double A[], int nA)
{
    double s = 0, *Aend = A + nA;
    while( A < Aend ) s += *(A++);
    return s;
}

```

Соответствующий фактический параметр – константный адрес массива `B` в вызывающей функции не меняет своего значения, то есть никаких противоречий в предыдущей функции нет.

Для формальных параметров-массивов описания в виде `A[]` и `*A` совершенно идентичны и обозначают локальную копию адреса соответствующего типа. Какое из этих описаний использовать, зависит от смысла параметра [3]. Если это массив, то более наглядно использовать описание вида `A[]`.

Возврат значений

Возврат результата работы функции в вызывающую программу в виде единственного значения можно осуществить с помощью оператора `return`. При этом результат возвратится как значение самой функции и должен иметь соответствующий тип. Типы возвращаемых значений могут быть любыми, кроме массивов. Тип `void` означает,

что функция не возвращает никакого значения. Тип `void*` означает, что функция возвращает указатель на произвольный тип данных [3].

Если требуется изменить содержимое массива, его адрес необходимо передать в функцию и обычным способом, с помощью операции индексации, изменить необходимые элементы массива.

Пример 2

В следующем примере функция `FillArray()` заполняет массив указанным значением:

```
void FillArray(double A[], int nA, double val)
{
    int i;
    for (i=0; i<nA; i++) A[i] = val;
}

void main (void)
{
    double B[100];
    FillArray(B, 40, 35.4);
    /* ... */
    FillArray(&B[60], 20, 15.4);
    /* ... */
}
```

Первый вызов `FillArray()` заполняет 40 первых элементов массива `B` значением 35.4, второй вызов заполняет 20 элементов массива `B`, начиная с элемента `B[60]`, значением 15.4. При возврате из функции массив будет изменен, так как занесение значения `val` происходит непосредственно по необходимому адресу.

Эту же функцию можно использовать для заполнения строк двумерного массива:

```
void main (void)
{
    double a[10][20];
    int n = sizeof(a) / sizeof(a[0]);
    int m = sizeof(a[0]) / sizeof(a[0][0]);
    int i;
    /* ... */
    for(i=0; i<n; i++ )
        FillArray(a[i], m, 14.6);
    /* ... */
}
```


В примере необходимо обратить внимание на соответствие типов передаваемых параметров и на способ вычисления количества строк и количества столбцов двумерного массива.

Возврат из функции нескольких значений, которые не являются элементами массива, можно организовать, используя указатели.

Пример 3

В следующем примере функция Decart() осуществляет перевод пары полярных координат в декартовые:

```
void Decart(double *px, double *py, double r,
double f)
{
    (*px) = r * cos(f);
    (*py) = r * sin(f);
}
```

При обращении к этой функции для параметров *px* и *py* необходимо передавать адреса:

```
void main(void)
{
    double x, y, r=5, f=0.5;
    /* ... */
    Decart( &x, &y, r, f );
    /* ... */
}
```

В рассмотренном примере при вызове функции создаются локальные копии адресов переменных *x* и *y*, а внутри функции происходит обращение к переменным *x* и *y* через их адреса (как и в случае массивов), поэтому значения *x* и *y* после вызова функции будут изменены.

3. Индивидуальная часть лабораторной работы

В индивидуальной части лабораторной работы необходимо добавить в библиотеку, разработанную в результате выполнения лабораторных работ №8–9, новую функцию AddLine, которая добавляет одну строку в конец рваного массива, полученного через список аргументов, и возвращает новое количество строк в нём.

Необходимо иметь в виду, что для изменения значения параметра, полученного через список аргументов, этот параметр должен быть передан как указатель на искомый объект.

Кроме самой библиотеки необходимо разработать демонстрационную программу, с помощью которой пользователь может убедиться в корректной работе библиотеки.

4. Контрольные вопросы

1. Приведите пример кода с некорректной попыткой изменения значения аргумента.

2. Приведите пример кода с корректной попыткой изменения значения аргумента.

3. Какие способы сообщить из пользовательской функции 2 и более значений вы знаете? Приведите код.

Лабораторная работа №13. Динамическая структура данных «стек». Реализация в динамическом массиве

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков реализации динамической структуры «стек» на динамическом массиве.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Динамические структуры данных – это структуры данных, память под которые выделяется и освобождается не на этапе компиляции, а в процессе работы программы.

Динамический элемент – это элемент динамической структуры, который в конкретный момент выполнения программы может либо существовать, либо отсутствовать в памяти.

Динамическое распределение памяти – это выделение памяти под отдельные элементы динамической структуры в тот момент, когда они «начинают существовать» в процессе выполнения программы.

Информационное поле (поле данных) – это поле структуры, в котором содержатся непосредственно обрабатываемые данные.

Связное представление данных – это установление связи между элементами динамической структуры через указатели.

Можно выделить следующие виды динамических структур данных (ДСД) [3]:

По способу обхода (доступа):

- список;
- стек;
- очередь;
- очередь с приоритетом;
- кольцевая очередь;
- ассоциативный массив.

По количеству связей элементов с «соседними» элементами:

- односвязные;
- двухсвязные;
- многосвязные.

По характеру связи:

- Линейные. В линейной динамической структуре данные связываются в цепочку. К линейным структурам относятся списки (односвязные, двухсвязные, кольцевые), стеки, очереди (односторонние, двухсторонние, очереди с приоритетами).
- Нелинейные (деревья, графы).

Из всего многообразия ДСД рассмотрим только наиболее распространённые из них.

Деревья

Каждая вершина дерева представляет собой структуру, имеющую информационное поле и указатели поддеревья, исходящие из этой вершины (рис. 1.). Максимальное количество поддеревьев, сходящихся в одной вершине, называется *порядком дерева*. Если порядок дерева равен двум, то дерево называется *бинарным* [3].

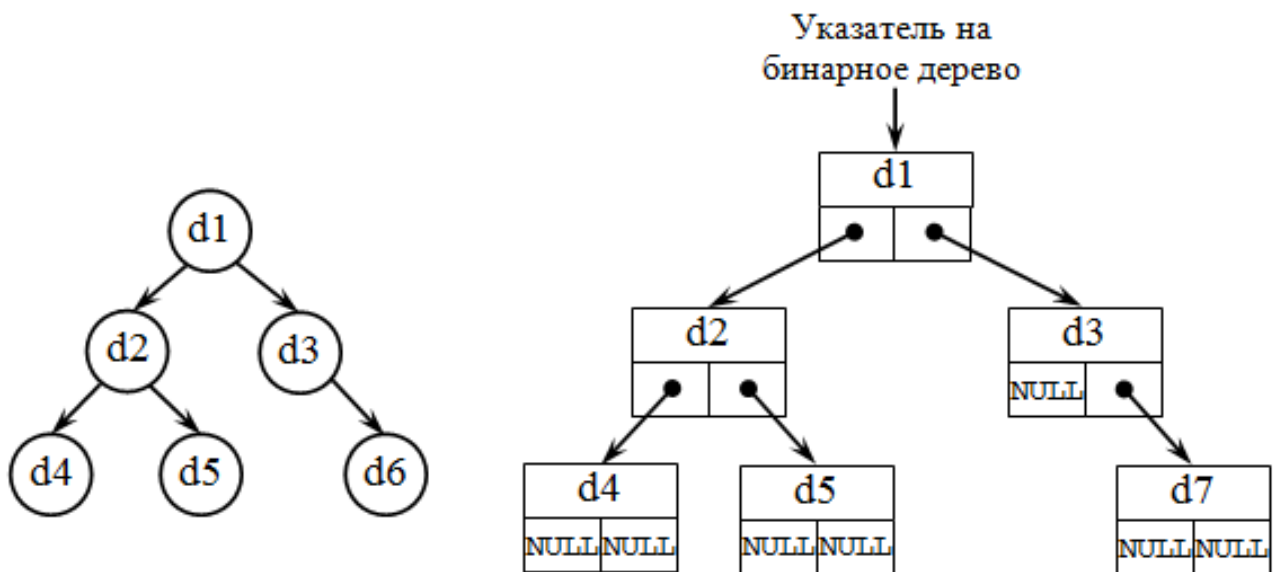


Рис. 1. Динамическая структура данных «Дерево»

Стек (LIFO) – такой последовательный список с переменной длиной, включение и исключение элементов из которого выполняются только с одной стороны списка, называемого вершиной стека (рис. 2) [3].

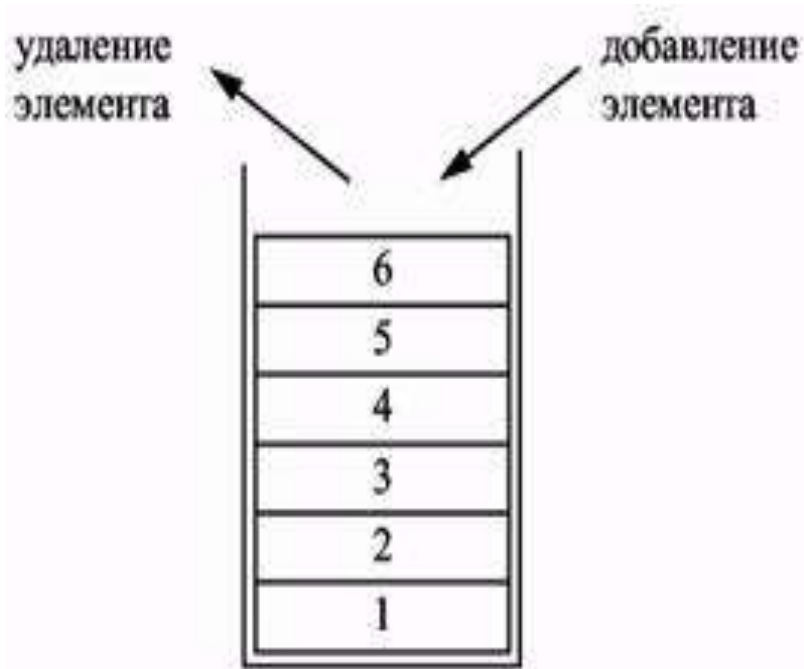


Рис. 2. Динамическая структура «Стек»

Основные операции над стеком – включение нового элемента в стек (английское название push – заталкивать) и исключение элемента из стека (англ. pop – выскакивать).

Полезными могут быть также вспомогательные операции: определение текущего количества элементов в стеке; очистка стека; неразрушающее чтение элемента из вершины стека, которое может быть реализовано, как комбинация основных операций.

3. Общая часть лабораторной работы

Задача 1. Разработать интерфейс ДСД «Стек» на основе динамических массивов.

Решение. Как было показано, основными операциями со стеком являются вставка в стек и извлечение из него. Вспомогательными являются следующие операции [3]:

- определение текущего количества элементов в стеке или проверка наличия элементов в стеке;
- очистка стека и неразрушающее чтение элемента из вершины стека;
- печать стека.

С учётом необходимости управления динамической памятью добавляют следующие служебные операции: инициализация и уничтожение стека.

Схематично работа со стеком представлена на рис. 3.

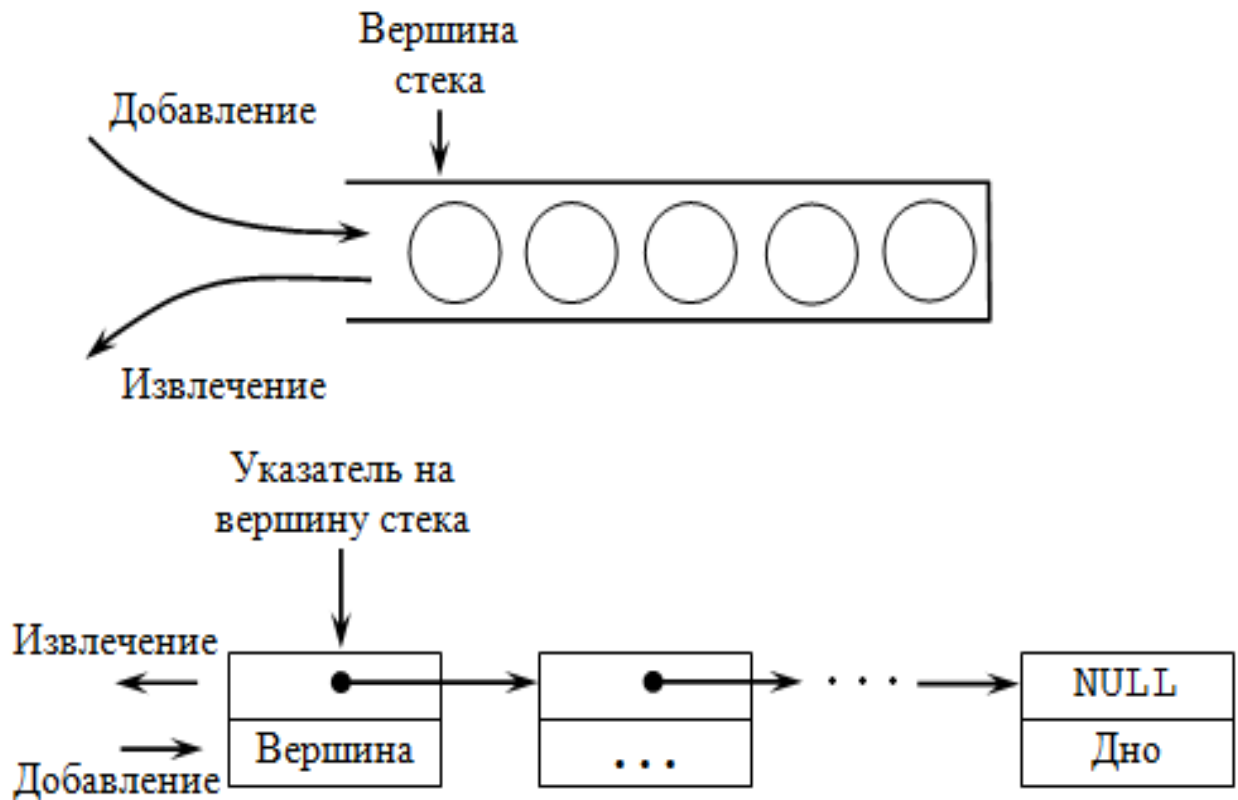


Рис. 3. Работа со структурой «Стек»

Необходимо так проектировать интерфейс динамической структуры данных (как, впрочем, и других типов), чтобы избавить пользователя от ненужных подробностей её внутреннего устройства.

Рассмотрим описание интерфейса стека на примере стека для хранения вещественных чисел. Тип хранимых элементов в дальнейшем может быть адаптирован для других практических задач.

Шаг 1. Поместим описание интерфейса стека в заголовочный файл «StackA.h»:

```
#ifndef _STACK_A_H
#define _STACK_A_H

typedef struct
{
    double *A;
    int l;
    int top;
} StackA;

int Push(StackA *s, double v);
double Pop(StackA *s);
double Peek(StackA const *s);
```

```

int IsEmptyStack(StackA const *s);
void Clear(StackA *s);
void InitStack(StackA *s, int l);
void PrintStack(StackA const *s);
void Destruct(StackA *s);

#endif

```

Для предотвращения ошибки повторного определения типов при многократном (прямом или опосредованном) включении заголовочного файла обраним его содержимое директивами условной компиляции.

Шаг 2. Создадим файл «StackA.c», в котором опишем функции для работы с стеком. Подключим в файле заголовочный файл «stackA.h».

Опишем функции для работы со стеком:

```

void InitStack(StackA *s, int l) // инициализация стека
из l элементов
{
    if(s->A=(double *)malloc(sizeof(double)*l))
    {
        s->l=l;
    }
    else s->l=-1;
    s->top=-1;
}
void PrintStack(StackA const *s) // вывод элементов стека

{
    int i;
    for(i=0; i<= s->top; i++)
        printf ("%7.3lf ", (s->A)[i]);
}
void Destruct(StackA *s) // уничтожение стека

{
    if(s->A) free(s->A);
    s->l=-1;
    s->top=-1;
}
int Push(StackA *s, double v) // добавление элемента в
стек

```

```

{
    if(s->top>=s->l-1) return 0;
    s->top++;
    s->A[s->top]=v;
    return 1;
}

double Pop(StackA *s)// достать элемент
{
    if(s->top>=0)
        return s->A[s->top--];
    else return 1e300;
}

double Peek(StackA const *s)// посмотреть верхний
элемент без удаления
{
    if(s->top>=0)
        return s->A[s->top];
    else return 1e300;
}

void Clear(StackA *s) //очистка стека
{
    s->top=-1;
}

int IsEmptyStack(StackA const *s)// проверка стека на
наличие элементов
{
    if(s->top <0) return 1;
    else return 0;
}

```

Полученные функции могут быть применены для задач, решаемых с помощью стека.

4. Индивидуальная часть лабораторной работы

В индивидуальной части лабораторной работы написать и отладить программу, реализующую работу с динамической структурой данных «Стек» на основе динамических массивов. Программа должна запросить у пользователя все необходимые исходные данные и вывести результаты решения задачи на экран.

Также программа должна быть снабжена комментариями и отформатирована.

5. Задачи для самостоятельного решения

1. Ввести некоторое число и записать его цифры в стек. Вывести число, у которого цифры идут в обратном порядке. Цифры из записи числа считывать поочередно (например, с помощью `getch()`).
2. Ввести некоторое число и записать его цифры в стек. Цифры из записи числа считывать поочередно (например, с помощью `getch()`). Вычислить сумму и произведение его цифр.
3. Ввести некоторое число и записать его цифры в стек. Вывести число, у которого цифры идут в обратном порядке. Цифры из записи числа считывать поочередно (например, с помощью `getch()`). Определить, верно ли, что цифры, стоящие на нечетных позициях, являются четными, а на четных позициях – нечетными.
4. Ввести некоторое число и записать его цифры в стек. Вывести число, у которого цифры идут в обратном порядке. Цифры из записи числа считывать поочередно (например, с помощью `getch()`). Вывести все цифры, стоящие на нечетных позициях.
5. Ввести некоторое число и записать его цифры в стек. Вывести число, у которого цифры идут в обратном порядке. Цифры из записи числа считывать поочередно (например, с помощью `getch()`). Вычислить наименьшую и наибольшую цифры в записи числа.
6. Ввести некоторое число и записать его цифры в стек. Вывести число, у которого цифры идут в обратном порядке. Цифры из записи числа считывать поочередно (например, с помощью `getch()`). Вывести число, которое получится, если поменять цифры стоящие на четных и нечетных позициях местами.

6. Контрольные вопросы

1. Что такое динамические структуры данных?
2. Какие виды динамических структур данных существуют?
3. Что такое стек?
4. Какие операции для работы со стеком необходимы?

Лабораторная работа №14. Динамическая структура данных «стек». Реализация в связном списке

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков реализации динамической структуры «стек» с помощью связанного списка.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Динамические структуры данных – это структуры данных, память под которые выделяется и освобождается не на этапе компиляции, а в процессе работы программы.

Динамический элемент – это элемент динамической структуры, который в конкретный момент выполнения программы может либо существовать, либо отсутствовать в памяти.

Динамическое распределение памяти – это выделение памяти под отдельные элементы динамической структуры в тот момент, когда они «начинают существовать» в процессе выполнения программы.

Информационное поле (поле данных) – это поле структуры, в котором содержатся непосредственно обрабатываемые данные.

Связное представление данных – это установление связи между элементами динамической структуры через указатели.

Можно выделить следующие виды динамических структур данных (ДСД) [3]:

По способу обхода (доступа):

- список;
- стек;
- очередь;
- очередь с приоритетом;
- кольцевая очередь;
- ассоциативный массив.

По количеству связей элементов с «соседними» элементами:

- односвязные;
- двухсвязные;
- многосвязные.

По характеру связи:

- **Линейные.** В линейной динамической структуре данные связываются в цепочку. К линейным структурам относятся списки (односвязные, двухсвязные, кольцевые), стеки, очереди (односторонние, двухсторонние, очереди с приоритетами).
- **Нелинейные (деревья, графы).**

Из всего многообразия ДСД рассмотрим только наиболее распространённые из них.

Стек – такой последовательный список с переменной длиной, включение элементов в который и исключение элементов из которого выполняются только с одной стороны списка, называемого вершиной стека (рис. 1) [3].

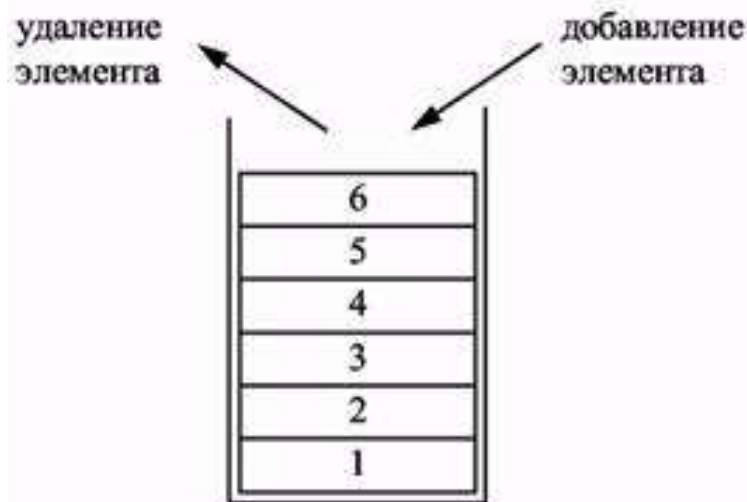


Рис. 1. Динамическая структура «Стек»

Основные операции над стеком – включение нового элемента в стек (английское название *push* – заталкивать) и исключение элемента из стека (англ. *pop* – выскакивать).

Полезными могут быть также вспомогательные операции:

- определение текущего количества элементов в стеке;
- очистка стека;
- неразрушающее чтение элемента из вершины стека, которое может быть реализовано как комбинация основных операций.

3. Общая часть лабораторной работы

Задача 1. Разработать интерфейс ДСД «Стек» на основе связанных списков.

Решение. Решение задачи представим в виде библиотеки для работы со стеком и демонстрационной программы.

Шаг 1. Поместим описание интерфейса стека в заголовочный файл «StackL.h»:

```
#ifndef _STACK_L_H
#define _STACK_L_H
struct LE
{
    double v;
    struct LE *next;
};

typedef struct
{
    struct LE *top;
} StackL;

int Push(StackL *s, double v);
double Pop(StackL *s);
double Peek(StackL const *s);
int IsEmptyStack(StackL const *s);
void Clear(StackL *s);
void InitStack(StackL *s);
void PrintStack(StackL const *s);
void Destruct(StackL *s);

#endif
```

Шаг 2. Реализацию библиотеки поместим в файл «StackL.c»:

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

void InitStack(StackL *s)
{
    s->top=NULL;
}

int Push(StackL *s, double v)
{
    struct LE *p=(struct LE *)malloc(sizeof(struct LE));
```

```

    if(!p) return 0;
    p->next=s->top;
    p->v=v;
    s->top=p;
    return 1;
}

double Pop(StackL *s)
{
    struct LE *p;
    double x=1e300;

    if(s->top)
    {
        p=s->top;
        s->top=s->top->next;
        x=p->v;
        free(p);
    }
    return x;
}

double Peek(StackL const *s)
{
    if(s->top)
        return s->top->v;
}

int IsEmptyStack(StackL const *s)
{
    return s->top==NULL;
}

void Clear(StackL *s)
{
    while(!IsEmptyStack(s)) Pop(s);
}

void PrintStack(StackL const *s)
{
    struct LE *p;
    for(p=s->top; p; p=p->next)
        printf("%-7.3lf ", p->v);
}

```

```
void Destruct(StackL *s)
{
    Clear(s);
}
```

Шаг 3. Опишем вызов функций для работы со стеком:

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

#include "stackL.h"

int main(int argc, char *argv[])
{
    StackL a, b;
    int i;
    double x;

    InitStack(&a);
    Push(&a, 1.1);
    Push(&a, 2.2);

    printf ("Enter value to push: ");
    scanf("%lf", &x);
    Push(&a, x);
    PrintStack(&a);

    for (i=0; i<5; i++) printf("\n Extracted %-7.3lf\n",
Pop(&a));

    PrintStack(&a);
    Destruct(&a);

    system("PAUSE");
    return 0;
}
```

Полученные функции могут быть использованы для задач, решаемых с помощью стека.

4. Индивидуальная часть лабораторной работы

В индивидуальной части лабораторной работы написать и отладить программу, реализующую работу с динамической структурой данных «Стек» на основе связанных списков. Программа

должна запросить у пользователя все необходимые исходные данные и вывести результаты решения задачи на экран.

Программа должна быть снабжена комментариями и отформатирована.

5. Задачи для самостоятельного решения

1. Ввести некоторое число и записать его цифры в стек. Вывести число, у которого цифры идут в обратном порядке. Цифры из записи числа считывать поочередно (например, с помощью `getch()`).
2. Ввести некоторое число и записать его цифры в стек. Цифры из записи числа считывать поочередно (например, с помощью `getch()`). Вычислить сумму и произведение его цифр.
3. Ввести некоторое число и записать его цифры в стек. Вывести число, у которого цифры идут в обратном порядке. Цифры из записи числа считывать поочередно (например, с помощью `getch()`). Определить, верно ли, что цифры, стоящие на нечетных позициях, являются четными, а на четных позициях – нечетными.
4. Ввести некоторое число и записать его цифры в стек. Вывести число, у которого цифры идут в обратном порядке. Цифры из записи числа считывать поочередно (например, с помощью `getch()`). Вывести все цифры, стоящие на нечетных позициях.
5. Ввести некоторое число и записать его цифры в стек. Вывести число, у которого цифры идут в обратном порядке. Цифры из записи числа считывать поочередно (например, с помощью `getch()`). Вычислить наименьшую и наибольшую цифры в записи числа.
6. Ввести некоторое число и записать его цифры в стек. Вывести число, у которого цифры идут в обратном порядке. Цифры из записи числа считывать поочередно (например, с помощью `getch()`). Вывести число, которое получится, если поменять цифры, стоящие на четных и нечетных позициях местами.

6. Контрольные вопросы

1. Что такое динамические структуры данных?
2. Какие виды динамических структур данных существуют?
3. Что такое стек?
4. Какие операции для работы со стеком необходимы?

Лабораторная работа №15. Динамическая структура данных «Очередь»

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков реализации динамической структуры «очередь» и применения её для решения задач.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Динамические структуры данных – это структуры данных, память под которые выделяется и освобождается не на этапе компиляции, а в процессе работы программы.

Динамический элемент – это элемент динамической структуры, который в конкретный момент выполнения программы может либо существовать, либо отсутствовать в памяти.

Динамическое распределение памяти – это выделение памяти под отдельные элементы динамической структуры в тот момент, когда они «начинают существовать» в процессе выполнения программы.

Информационное поле (поле данных) – это поле структуры, в котором содержатся непосредственно обрабатываемые данные.

Связное представление данных – это установление связи между элементами динамической структуры через указатели.

Можно выделить следующие виды динамических структур данных (ДСД) [3]:

По способу обхода (доступа):

- список;
- стек;
- очередь;
- очередь с приоритетом;
- кольцевая очередь;
- ассоциативный массив.

По количеству связей элементов с «соседними» элементами:

- односвязные;
- двухсвязные;
- многосвязные.

По характеру связи:

- Линейные. В линейной динамической структуре данные связываются в цепочку. К линейным структурам относятся списки (односвязные, двухсвязные, кольцевые), стеки, очереди (односторонние, двухсторонние, очереди с приоритетами).
- Нелинейные (деревья, графы).

Из всего многообразия ДСД рассмотрим только наиболее распространённые из них.

Очередью (FIFO, First In – First Out, «первым пришел – первым исключается») называется такой последовательный список с переменной длиной, в котором включение элементов выполняется только с одной стороны списка (эту сторону часто называют **концом** или **хвостом очереди**), а исключение – с другой стороны (называемой **началом** или **головой очереди**) (рис. 1) [3].

Основные операции над очередью – те же, что и над стеком: включение, исключение, определение размера, очистка, неразрушающее чтение.

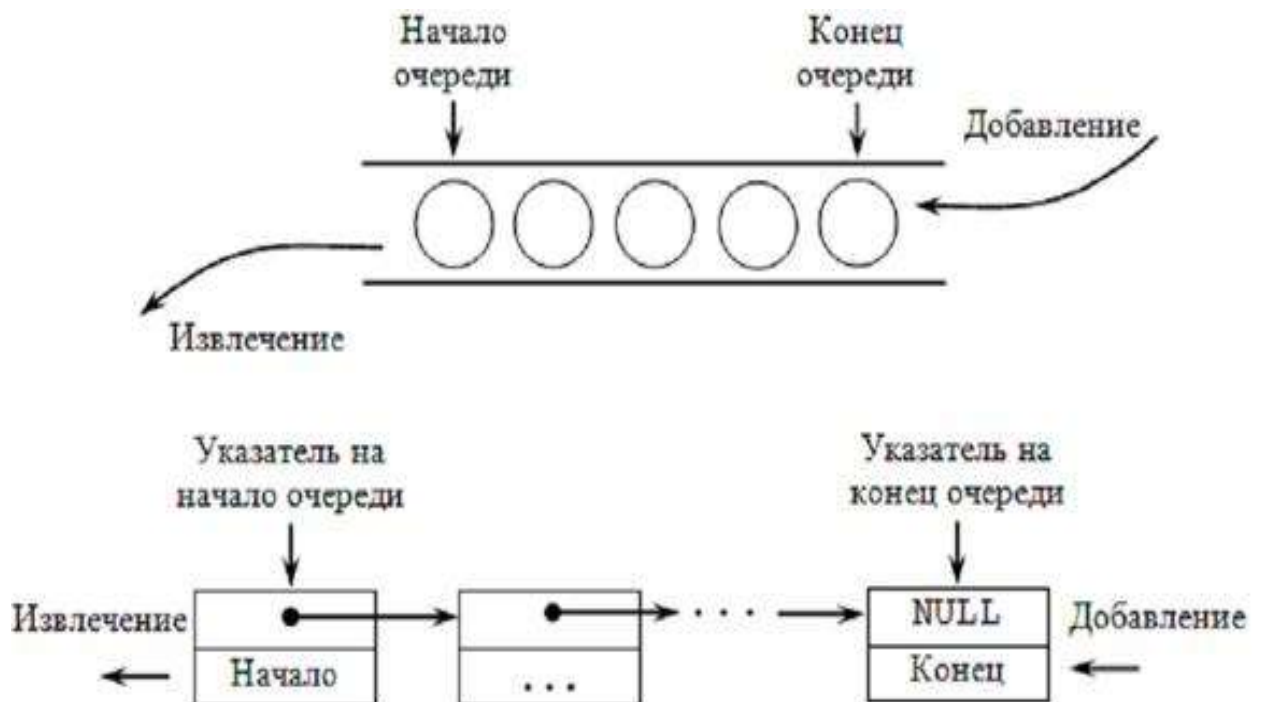


Рис. 1. Динамическая структура «Очередь»

Дек (от англ. *deq* – double ended queue, т.е очередь с двумя концами) – это такой последовательный список, в котором как

включение, так и исключение элементов может осуществляться с любого из двух концов списка (рис. 2).



Рис. 2. Динамическая структура «Дек»

3. Общая часть лабораторной работы

Задача 1. Разработать интерфейс ДСД «Очередь» на основе связанных списков.

Решение. Описание очереди выглядит следующим образом:

```
struct имя_типа {
    информационное_поле;
    адресное_поле1;
    адресное_поле2;
};
```

Здесь `информационное_поле` – это поле любого, ранее объявленного или стандартного, типа; `адресное_поле1`, `адресное_поле2` – это указатели на объекты того же типа, что и определяемая структура, в них записываются адреса первого и следующего элементов очереди. Адресное поле может ссылаться на ранее объявленную структуру:

```
struct list1 {
    type pole1;
    list1 *pole2;
}
struct ch3 {
    list1 *beg, *next;
}
```

Очередь как динамическую структуру данных легко организовать на основе линейного списка. При этом предпочтительно будет использовать линейный двунаправленный список, так как работа идет с обоими концами очереди. Хотя для работы с таким списком достаточно иметь один указатель на любой элемент списка, здесь целесообразно хранить два указателя – один на начало списка (откуда извлекаем элементы) и один на конец списка (куда добавляем элементы). Если очередь пуста, то списка не существует, и указатели принимают значение NULL.

Описание элементов очереди аналогично описанию элементов линейного двунаправленного списка, поэтому объявим очередь через объявление линейного двунаправленного списка:

```
struct Queue {
    Double_List *Begin; //начало очереди
    Double_List *End;  //конец очереди
};

Queue *My_Queue; //указатель на очередь
```

Основные операции, производимые с очередью:

- создание очереди;
- печать (просмотр) очереди;
- добавление элемента в конец очереди;
- извлечение элемента из начала очереди;
- проверка пустоты очереди;
- очистка очереди.

Реализацию этих операций приведем в виде соответствующих функций, которые в свою очередь используют функции операций с линейным двунаправленным списком.

```
//создание очереди
void Make_Queue(int n, Queue* End_Queue){
    Make_Double_List(n, &(End_Queue->Begin), NULL);
    Double_List *ptr; //вспомогательный указатель
    ptr = End_Queue->Begin;
    while (ptr->Next != NULL)
        ptr = ptr->Next;
    End_Queue->End = ptr;
}
```

```

//печать очереди
void Print_Queue(Queue* Begin_Queue){
    Print_Double_List(Begin_Queue->Begin);
}

//добавление элемента в конец очереди
void Add_Item_Queue(int NewElem, Queue* End_Queue){
    End_Queue->End = Insert_Item_Double_List(End_Queue->End, 0, NewElem)->Next;
}

//извлечение элемента из начала очереди
int Extract_Item_Queue(Queue* Begin_Queue){
    int NewElem = NULL;
    if (Begin_Queue->Begin != NULL) {
        NewElem = Begin_Queue->Begin->Data;
        Begin_Queue->Begin=Delete_Item_Double_List(Begin_Queue->Begin,0);
        //удаляем вершину
    }
    return NewElem;
}

//проверка пустоты очереди
bool Empty_Queue(Queue* Begin_Queue){
    return Empty_Double_List(Begin_Queue->Begin);
}

//очистка очереди
void Clear_Queue(Queue* Begin_Queue){
    return Delete_Double_List(Begin_Queue->Begin);
}

```

На основе полученных функций необходимо сформировать библиотеку.

4. Индивидуальная часть лабораторной работы

В индивидуальной части лабораторной работы написать и отладить программу, реализующую работу с динамической структурой данных «Очередь». Программа должна запросить у пользователя все необходимые исходные данные и вывести результаты решения задачи на экран.

Программа должна быть снабжена комментариями и отформатирована.

5. Задачи для самостоятельного решения

1. Используя динамическую структуру «очередь», зашифровать содержимое текста: каждый символ заменить его кодом+1.

2. Сформировать очередь из элементов целого типа. Четные элементы возвести в квадрат. Распечатать исходную и результирующую очереди.

3. Используя динамическую структуру «очередь», перевести введенную последовательность цифр в слово, состоящее из кодов ASCII.

4. Сформировать динамическую структуру «очередь», элементами которой являются цифры. Извлекая элементы из очереди, напечатать их двоичные эквиваленты.

5. Дан текстовый файл. Используя очередь, переписать содержимое его строк в новый текстовый файл, перенося при этом в конец каждой строки все входящие в нее цифры, сохраняя исходный порядок следования среди цифр и среди остальных символов строки.

6. Дан файл из целых чисел. Используя очередь, за один просмотр файла напечатать сначала все отрицательные числа, затем все положительные числа, сохраняя исходный порядок в каждой группе.

6. Контрольные вопросы

1. Что такое динамические структуры данных?
2. Какие виды динамических структур данных существуют?
3. Что такое очередь?
4. Какие операции для работы с очередью необходимы?

Лабораторная работа №16. Динамическая структура данных «Список с произвольным доступом»

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков реализации динамической структуры «список с произвольным доступом» и применения её для решения задач.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Динамические структуры данных – это структуры данных, память под которые выделяется и освобождается не на этапе компиляции, а в процессе работы программы.

Динамический элемент – это элемент динамической структуры, который в конкретный момент выполнения программы может либо существовать, либо отсутствовать в памяти.

Динамическое распределение памяти – это выделение памяти под отдельные элементы динамической структуры в тот момент, когда они «начинают существовать» в процессе выполнения программы.

Информационное поле (поле данных) – это поле структуры, в котором содержатся непосредственно обрабатываемые данные.

Связное представление данных – это установление связи между элементами динамической структуры через указатели.

Можно выделить следующие виды динамических структур данных (ДСД) [3]:

По способу обхода (доступа):

- список;
- стек;
- очередь;
- очередь с приоритетом;
- кольцевая очередь;
- ассоциативный массив.

По количеству связей элементов с «соседними» элементами:

- односвязные;
- двухсвязные;
- многосвязные.

По характеру связи:

- **Линейные.** В линейной динамической структуре данные связываются в цепочку. К линейным структурам относятся списки (односвязные, двухсвязные, кольцевые), стеки, очереди (односторонние, двухсторонние, очереди с приоритетами).
- **Нелинейные** (деревья, графы).

Из всего многообразия ДСД рассмотрим только наиболее распространённые из них.

Списком называется упорядоченное множество, состоящее из переменного количества элементов, к которым применимы операции включения, исключения. Список, отражающий отношения соседства между элементами, называется **линейным**. **Длина списка** равна количеству элементов, содержащихся в списке, список нулевой длины называется пустым списком. Линейные связные списки являются простейшими динамическими структурами данных [3].

Графически связи в списках удобно изображать с помощью стрелок. Если элемент не связан ни с каким другим, то в поле указателя записывают значение, не указывающее ни на какой элемент. Такая ссылка обозначается специальным именем – NULL (nul, nil) (рис.1.).

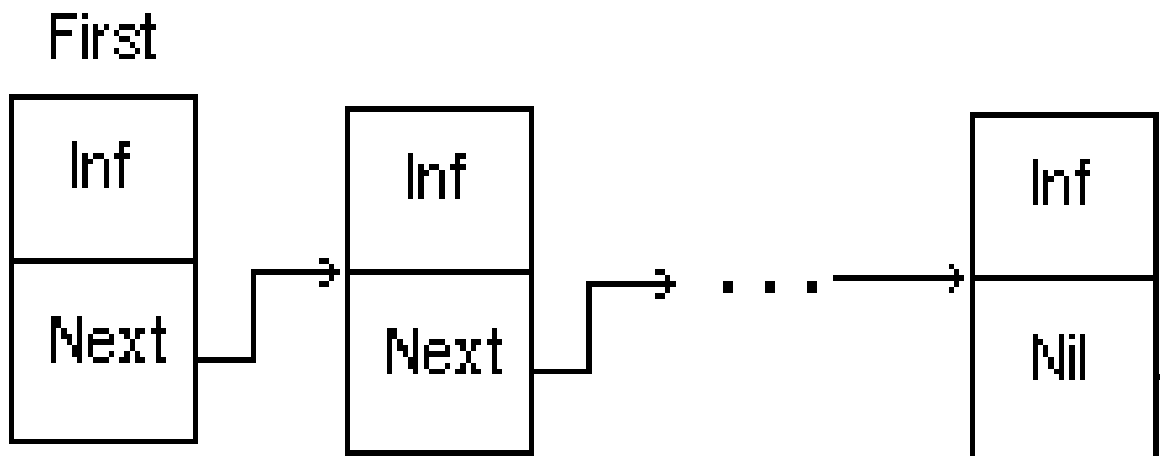


Рис. 1. Динамическая структура данных «Список»

Здесь Inf – информационная часть звена списка (величина любого простого или структурированного типа, кроме файлового), Next – указатель на следующее звено списка; First – указатель на заглавное звено списка.

Двусвязный список характеризуется наличием пары указателей в каждом элементе списка: на предыдущий элемент и на следующий (рис.2.). Очевидный плюс тут в том, что от элемента структуры можно пойти в обе стороны. Таким образом упрощаются многие операции. Однако на указатели тратится дополнительная память.

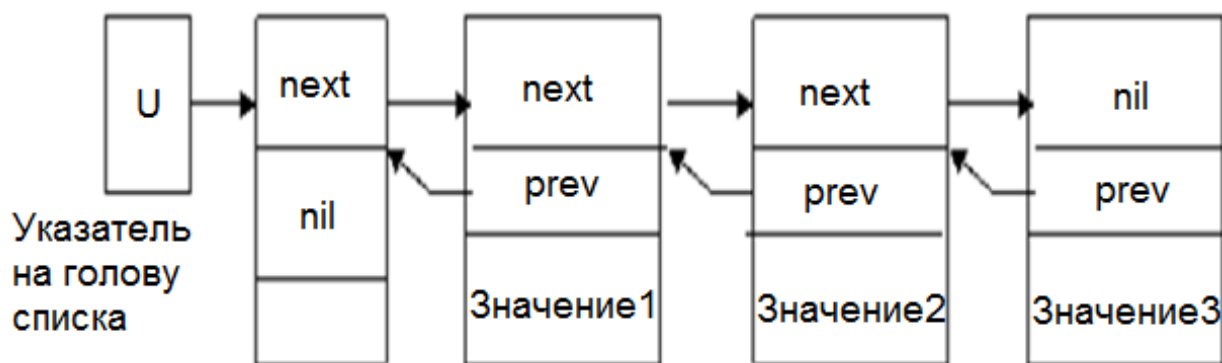


Рис. 2. Динамическая структура данных «Двусвязный список»

Одной из разновидностей линейных списков является **кольцевой список** (рис. 3.), который может быть организован на основе как односвязного, так и двухсвязного списков. При этом в односвязном списке указатель последнего элемента должен указывать на первый элемент; в двухсвязном списке в первом и последнем элементах соответствующие указатели переопределяются.

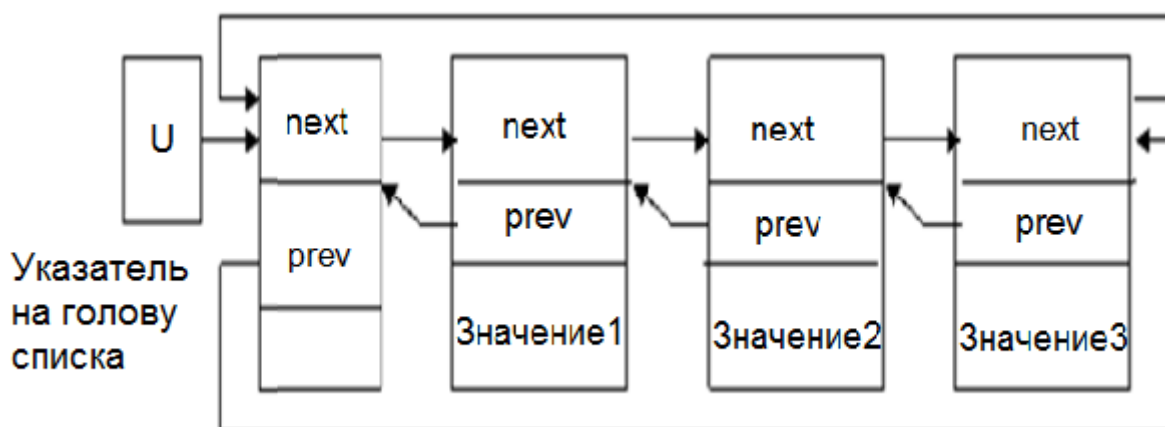


Рис. 3. Динамическая структура данных «Кольцевой список»

3. Общая часть лабораторной работы

Задача 1. Разработать интерфейс ДСД «Одномвязанный список»

Решение. Описание простейшего элемента такого списка в общем виде выглядит следующим образом [2]:

```
struct имя_типа {информационное_поле; адресное_поле;};
```


В этом примере информационное_поле – это поле любого, ранее объявленного или стандартного, типа; адресное_поле – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента списка.

Пример описания элемента списка приведён в следующем листинге:

```
struct Node {
    int key;//информационное поле
    Node*next;//адресное поле
};
```

Информационных полей может быть несколько:

```
struct point {
    char*name;//информационное поле
    int age;//информационное поле
    point*next;//адресное поле
};
```

Каждый элемент списка содержит ключ, который идентифицирует этот элемент. Ключ обычно бывает либо целым числом, либо строкой.

Основными операциями, осуществляемыми с однонаправленными списками, являются [3]:

- создание списка;
- печать (просмотр) списка;
- вставка элемента в список;
- удаление элемента из списка;
- поиск элемента в списке;
- проверка пустоты списка;
- удаление списка.

Отметим, что при выполнении любых операций с линейным однонаправленным списком необходимо обеспечивать позиционирование какого-либо указателя на первый элемент. В противном случае часть или весь список будет недоступен.

Для описания алгоритмов основных операций используется следующее объявление:

```

struct Single_List { //структура данных
    int Data; //информационное поле
    Single_List *Next; //адресное поле
};

. . . . .
Single_List *Head; //указатель на первый элемент списка
. . . . .
Single_List *Current;
//указатель на текущий элемент списка (при необходимости)

```

Создание однонаправленного списка

Чтобы создать список, необходимо сначала создать первый элемент списка, а затем с помощью функции добавить к нему остальные элементы.

При относительно небольших размерах списка удобно использовать рекурсивную функцию.

Добавление может выполняться как в начало, так и в конец списка.

```

//создание однонаправленного списка (добавления в конец)
void Make_Single_List(int n, Single_List** Head) {

    if (n > 0)
    {
        (*Head) = new Single_List();
        //выделяем память под новый элемент

        cout << "Введите значение ";
        cin >> (*Head)->Data;
        //вводим значение информационного поля

        (*Head)->Next=NULL; //обнуление адресного поля
        Make_Single_List(n-1, &((*Head)->Next));
    }
}

```

Печать (просмотр) однонаправленного списка

Операция печати списка заключается в последовательном просмотре всех элементов списка и выводе их значений на экран.

Для обработки списка организуется функция, в которой необходимо переставлять указатель на следующий элемент списка до тех пор, пока указатель не станет равен NULL, то есть будет достигнут конец списка.

Реализуем эту функцию рекурсивно.

```
//печать однонаправленного списка
void Print_Single_List(Single_List* Head)
{
    if (Head != NULL)
    {
        cout << Head->Data << "\t";
        Print_Single_List(Head->Next);
        //переход к следующему элементу
    }

    else cout << "\n";
}
```

Вставка элемента в однонаправленный список

В динамические структуры легко добавлять элементы, так как для этого достаточно изменить значения адресных полей.

Вставка первого и последующих элементов списка отличаются друг от друга, поэтому в функции, реализующей операцию, сначала осуществляется проверка, на какое место вставляется элемент.

Далее реализуется соответствующий алгоритм добавления (рис. 4).

```
/*вставка элемента с заданным номером в однонаправленный
список*/
Single_List* Insert_Item_Single_List(Single_List* Head,
    int Number, int DataItem){
    Number--;
    Single_List *NewItem=new(Single_List);
    NewItem->Data=DataItem;
    NewItem->Next = NULL;
```

```

if (Head == NULL) { //список пуст
    Head = NewItem; //создаем первый элемент списка
}
else { //список не пуст
    Single_List *Current=Head;
    for(int i=1; i < Number && Current->Next!=NULL; i++)
        Current=Current->Next;
    if (Number == 0){
        //вставляем новый элемент на первое место
        NewItem->Next = Head;
        Head = NewItem;
    }
    else { //вставляем новый элемент на непервое место
        if (Current->Next != NULL)
            NewItem->Next = Current->Next;
        Current->Next = NewItem;
    }
}
return Head;
}

```

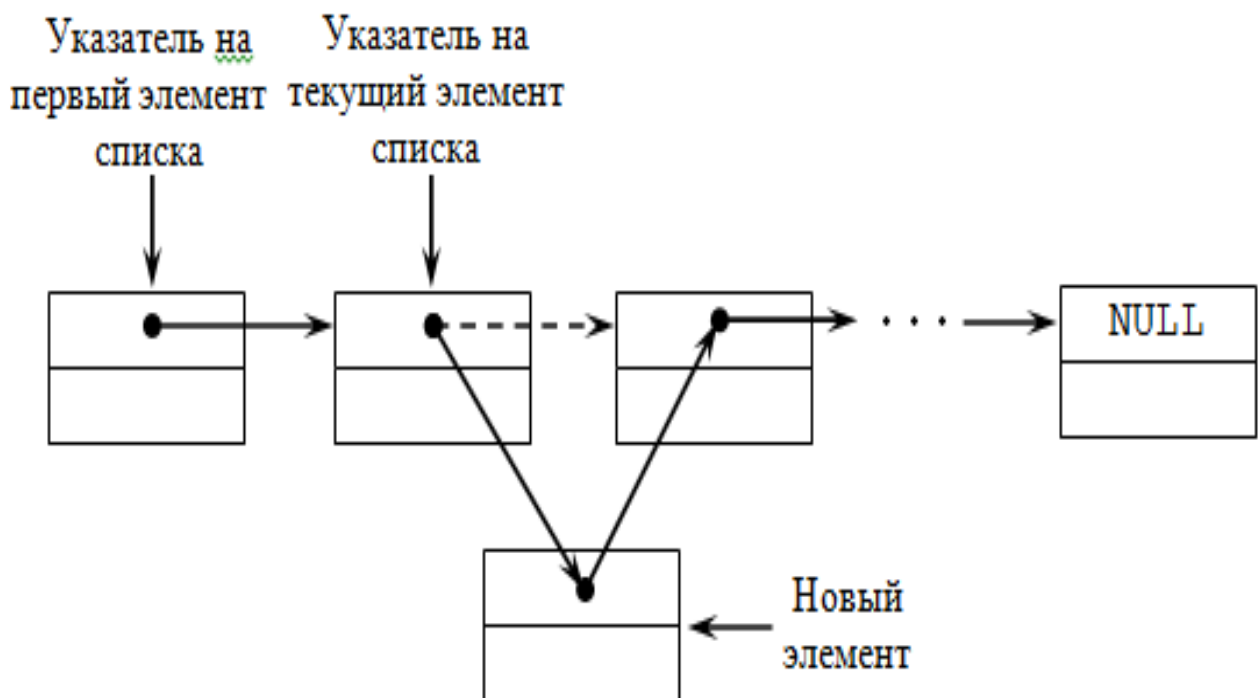


Рис. 4. Вставка элемента в однонаправленный список

Удаление элемента из однонаправленного списка

Из динамических структур можно удалять элементы, так как для этого достаточно изменить значения адресных полей. Операция

удаления элемента однонаправленного списка осуществляет удаление элемента, на который установлен указатель текущего элемента. После удаления указатель текущего элемента устанавливается на предшествующий элемент списка или на новое начало списка, если удаляется первый.

Алгоритмы удаления первого и последующих элементов списка отличаются друг от друга. Вследствие этого в функции, реализующей операцию удаления, осуществляется проверка, какой элемент удаляется. Далее реализуется соответствующий алгоритм удаления (рис. 5.).

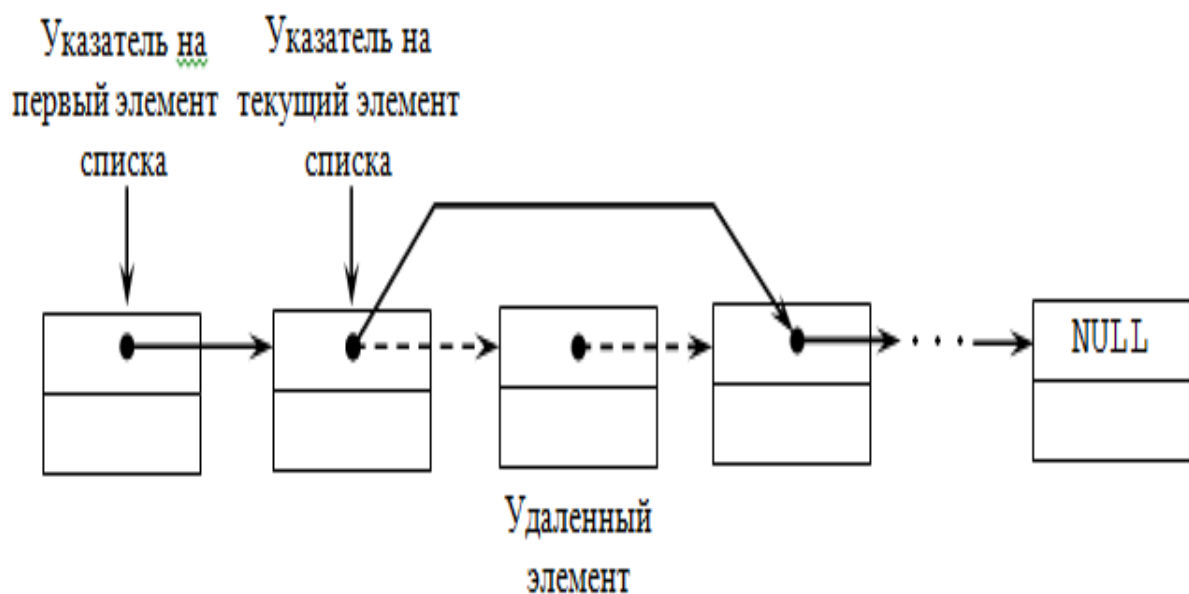


Рис. 5. Удаление элемента из однонаправленного списка

```
/*удаление элемента с заданным номером из
однонаправленного списка*/
Single_List* Delete_Item_Single_List(Single_List* Head,
    int Number){
    Single_List *ptr;//вспомогательный указатель
    Single_List *Current = Head;
    for (int i = 1; i < Number && Current != NULL; i++)
        Current = Current->Next;
    if (Current != NULL){//проверка на корректность
        if (Current == Head){//удаляем первый элемент
            Head = Head->Next;
            delete(Current);
            Current = Head;
        }
    }
```

```

else { //удаляем непервый элемент
    ptr = Head;
    while (ptr->Next != Current)
        ptr = ptr->Next;
    ptr->Next = Current->Next;
    delete(Current);
    Current=ptr;
}
}
return Head;
}

```

Поиск элемента в однонаправленном списке

Операция поиска элемента в списке заключается в последовательном просмотре всех элементов списка до тех пор, пока текущий элемент не будет содержать заданное значение или пока не будет достигнут конец списка.

В последнем случае фиксируется отсутствие искомого элемента в списке (функция принимает значение false).

```

/поиск элемента в однонаправленном списке
bool Find_Item_Single_List(Single_List* Head, int
DataItem){
    Single_List *ptr; //вспомогательным указатель
    ptr = Head;
    while (ptr != NULL){ //пока не конец списка
        if (DataItem == ptr->Data) return true;
        else ptr = ptr->Next;
    }
    return false;
}

```

Удаление однонаправленного списка

Операция удаления списка заключается в освобождении динамической памяти. Для этой операции организуется функция, в которой необходимо переставлять указатель на следующий элемент списка до тех пор, пока указатель не станет равен NULL, то есть не будет достигнут конец списка. Реализуем рекурсивную функцию.

```

/*освобождение памяти, выделенной под однонаправленный
список*/
void Delete_Single_List(Single_List* Head){
    if (Head != NULL){
        Delete_Single_List(Head->Next);
        delete Head;
    }
}

```

Таким образом, однонаправленный список имеет только один указатель в каждом элементе. Это позволяет минимизировать расход памяти на организацию такого списка. Одновременно это позволяет осуществлять переходы между элементами только в одном направлении, что зачастую увеличивает время, затрачиваемое на обработку списка. Так, для перехода к предыдущему элементу необходимо осуществить просмотр списка с начала до элемента, указатель которого установлен на текущий элемент.

4. Индивидуальная часть лабораторной работы

В индивидуальной части лабораторной работы написать и отладить программу, реализующую работу с динамической структурой данных «Односвязанный список». Программа должна запросить у пользователя все необходимые исходные данные и вывести результаты решения задачи на экран.

Программа должна быть снабжена комментариями и отформатирована.

5. Задачи для самостоятельного решения

1. Сформировать динамический список из элементов целого типа, проверить, является ли он упорядоченным набором чисел. Удалить из него отрицательные элементы.
2. Сформировать список динамических элементов, упорядоченный по возрастанию. Включить в список новый элемент, сохранив свойство упорядоченности. Из каждой группы подряд идущих одинаковых элементов оставить один. Распечатать исходный и результирующий списки.
3. Дан список L. Удалить из него повторяющиеся элементы. По списку L построить два новых списка L1 и L2: первый из элементов с положительными значениями, а второй из остальных

элементов исходного списка. Вывести исходный и полученные списки.

4. Дан список студентов. Элемент списка содержит фамилию, курс, номер группы, оценки по пяти предметам. Найдите средний балл каждой группы по каждому предмету. Определите самого старшего студента и самого младшего студентов. Определите самую большую группу.

6. Контрольные вопросы

1. Что такое динамические структуры данных?
2. Какие виды динамических структур данных существуют?
3. Что такое список?
4. Какие операции для работы со списком необходимы?

Лабораторная работа №17. Динамическая структура данных «Кольцо»

1. Цель лабораторной работы

Цель лабораторной работы – получение навыков реализации динамической структуры «Кольцо» для решения задач.

Продолжительность лабораторной работы – 2 часа.

2. Краткие теоретические сведения

Динамические структуры данных – это структуры данных, память под которые выделяется и освобождается не на этапе компиляции, а в процессе работы программы.

Динамический элемент – это элемент динамической структуры, который в конкретный момент выполнения программы может либо существовать, либо отсутствовать в памяти.

Динамическое распределение памяти – это выделение памяти под отдельные элементы динамической структуры в тот момент, когда они «начинают существовать» в процессе выполнения программы.

Информационное поле (поле данных) – это поле структуры, в котором содержатся непосредственно обрабатываемые данные.

Связное представление данных – это установление связи между элементами динамической структуры через указатели.

Можно выделить следующие виды динамических структур данных (ДСД) [3]:

По способу обхода (доступа):

- список;
- стек;
- очередь;
- очередь с приоритетом;
- кольцевая очередь;
- ассоциативный массив.

По количеству связей элементов с «соседними» элементами:

- односвязные;
- двухсвязные;
- многосвязные.

По характеру связи:

- **Линейные.** В линейной динамической структуре данные связываются в цепочку. К линейным структурам относятся списки (односвязные, двухсвязные, кольцевые), стеки, очереди (односторонние, двухсторонние, очереди с приоритетами).
- **Нелинейные** (деревья, графы).

Из всего многообразия ДСД рассмотрим только наиболее распространённые из них.

Списком называется упорядоченное множество, состоящее из переменного количества элементов, к которым применимы операции включения, исключения.

Список, отражающий отношения соседства между элементами, называется **линейным**.

Длина списка равна количеству элементов, содержащихся в списке, список нулевой длины называется пустым списком. Линейные связные списки являются простейшими динамическими структурами данных.

Графически связи в списках удобно изображать с помощью стрелок. Если компонент не связан ни с каким другим, то в поле указателя записывают значение, не указывающее ни на какой элемент. Такая ссылка обозначается специальным именем – NULL (nul, nil) (рис.1.).

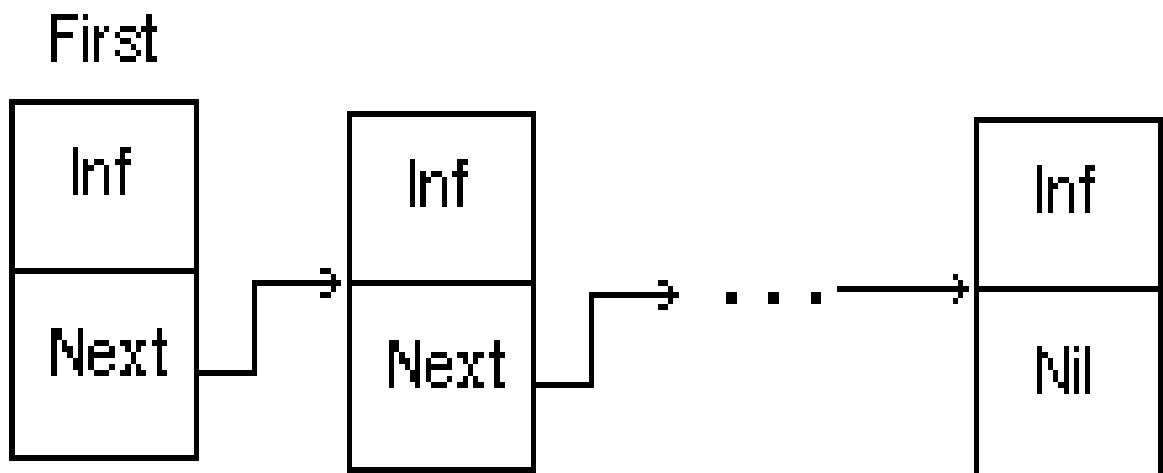


Рис. 1. Динамическая структура данных «Список»

Здесь Inf – информационная часть звена списка (величина любого простого или структурированного типа, кроме файлового),

Next – указатель на следующее звено списка; First – указатель на заглавное звено списка.

Двусвязный список характеризуется наличием пары указателей в каждом элементе списка: на предыдущий элемент и на следующий (рис.2.). Очевидный плюс тут в том, что от элемента структуры можно пойти в обе стороны. Таким образом упрощаются многие операции. Однако на указатели тратится дополнительная память.

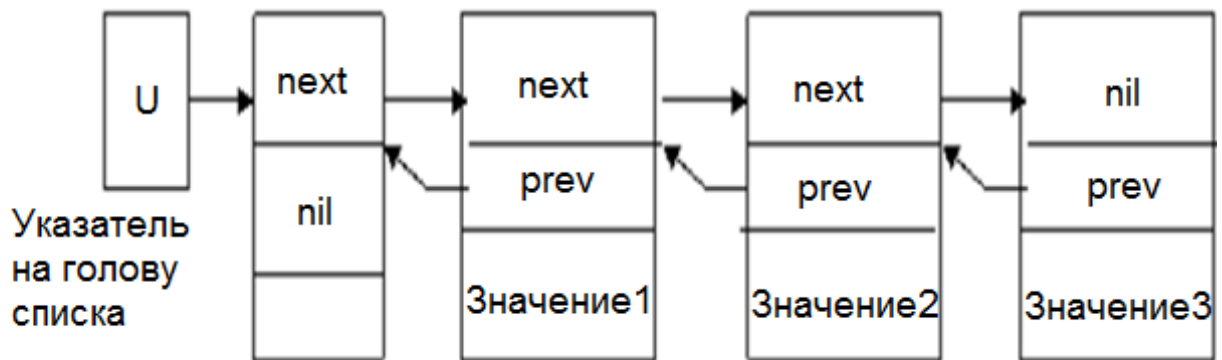


Рис. 2. Динамическая структура данных «Двусвязный список»

Одной из разновидностей линейных списков является *кольцевой список* (рис. 3), который может быть организован на основе как односвязного, так и двухсвязного списков. При этом в односвязном списке указатель последнего элемента должен указывать на первый элемент; в двухсвязном списке в первом и последнем элементах соответствующие указатели переопределяются.

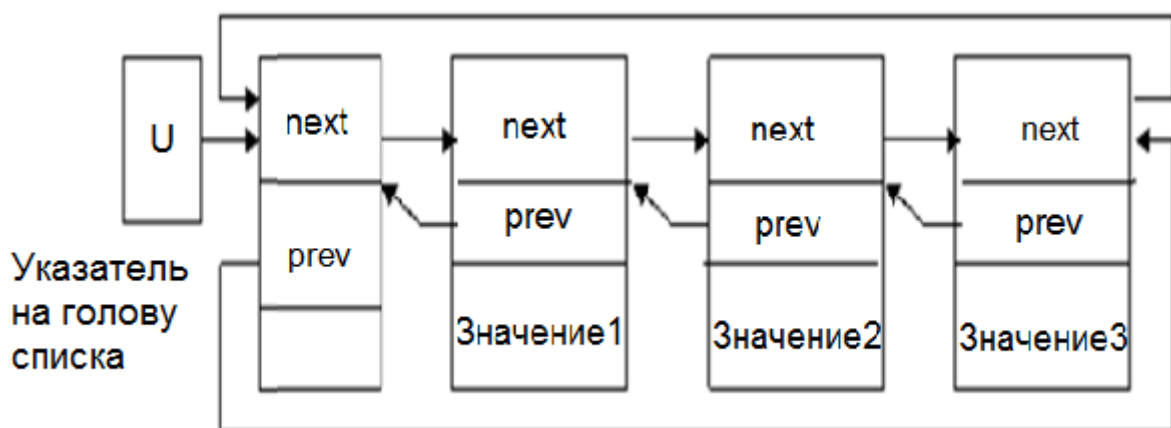


Рис. 3. Динамическая структура данных «Кольцевой список»

3. Общая часть лабораторной работы

Задача 1. Разработать интерфейс ДСД «Кольцо».

Решение. Опишем типы данных и структуры:

```

typedef int T;

typedef struct _Node {
    T data;
    struct _Node *next;
    struct _Node *prev;
} Node;

typedef struct _Ring {
    size_t size;
    Node *current;
} Ring;

Ring* createRing() {
    Ring *tmp = (Ring*)malloc(sizeof(Ring));

    tmp->size = 0;
    tmp->current = NULL;

    return tmp;
}

```

Реализация двунаправленного кольца очень похожа на реализацию двусвязного списка. Функции добавления и извлечения с последующим удалением элемента:

```

void addElement(Ring *ring, T value) {
    Node *prev = NULL;
    Node *tmp = (Node*)malloc(sizeof(Node));

    tmp->data = value;
    //Если в кольце нет элементов
    if (ring->current == NULL) {
        ring->current = tmp;
        tmp->next = tmp->prev = tmp;
    }

    else {
        prev = ring->current->next->prev;
        tmp->next = ring->current->next;
        tmp->prev = ring->current;
        prev->prev = tmp;
        ring->current->next = tmp;
    }
}

```

```

        ring->current = tmp;
    }
    ring->size++;
}

T removeElement(Ring *ring) {
    Node *afterTarget = NULL;
    T retVal;

    if (ring->current == NULL) {
        exit(1);
    }

    //Если в кольце последний элемент
    if (ring->current->next == ring->current) {
        retVal = ring->current->data;
        free(ring->current);
        ring->current = NULL;
    }

    else {
        afterTarget = ring->current->next;
        ring->current->prev->next = afterTarget;
        afterTarget->prev = ring->current->prev;
        retVal = ring->current->data;
        free(ring->current);
        ring->current = afterTarget;
    }

    ring->size--;
    return retVal;
}

```

Отметим, что при добавлении нового элемента указатель на элемент кольца сдвигается на вновь добавленный элемент. При удалении указатель `current` сдвигается на следующий элемент, после удаляемого. Функции `prev` и `next` позволяют двигаться по кольцевому списку. Функция `next` сдвигает указатель `current` вперёд, функция `prev` назад.

```

Node* next(Ring *ring) {
    Node* retVal = NULL;
    if (ring->current) {
        ring->current = ring->current->next;
        retVal = ring->current;
    }
    return retVal;
}

Node* prev(Ring *ring) {
    Node* retVal = NULL;

    if (ring->current) {
        ring->current = ring->current->prev;
        retVal = ring->current;
    }
    return retVal;
}

```

4. Индивидуальная часть лабораторной работы

В индивидуальной части лабораторной работы написать и отладить программу, реализующую работу с динамической структурой данных «Кольцо». Программа должна запросить у пользователя все необходимые исходные данные и вывести результаты решения задачи на экран.

Задача 1. Сформировать библиотеку для работы с кольцевым списком. Реализовать следующие функции:

- создание списка;
- печать (просмотр) списка;
- вставка элемента в список;
- удаление элемента из списка;
- поиск элемента в списке
- проверка пустоты списка;
- вычисление количества элементов в списке;
- удаление списка.

Используя полученную библиотеку, решить задачу согласно варианту.

5. Задачи для самостоятельного решения

1. N ребят располагаются по кругу. Начав отсчет от первого, удаляют каждого k -го, смыкая при этом круг. Определить порядок удаления ребят из круга.

2. По кругу выложены карточки с цифрами. Слева от карточек с цифрами, кратными пяти, положили карту со случайной цифрой, а саму карточку убрали. Действие выполнили некоторое количество раз. Вывести на экран исходный и полученный набор карточек.

3. По кругу выложены карточки с цифрами. Слева от карточек с цифрами, кратными пяти, положили карту со случайной цифрой, а находящуюся справа карту заменили на карту с цифрой, на три меньше текущей. Действие выполнили некоторое количество раз. Вывести на экран исходный и полученный набор карточек.

4. Задача носит имя Иосифа Флавия – известного историка I века. Существует легенда, что Иосиф выжил и стал известным благодаря своему математическому дару. В ходе иудейской войны он, будучи в составе отряда из 41 иудейского воина, был загнан римлянами в пещеру. Воины отряда обладали невероятно сильным боевым духом, не желали сдаваться в плен и предпочли самоубийство. Они встали в круг и последовательно начали убивать каждого третьего из живых до тех пор, пока не останется ни одного воина. Иосиф же не разделял чаяний других воинов. Чтобы не быть убитым, Иосиф вычислил «спасительные места» на которые поставил себя и своего товарища. В этой задаче предлагается вычислить, какой (какие) элементы останутся последними в списке, если из списка последовательно будет удаляться каждый третий элемент.

6. Контрольные вопросы

1. Что такое динамические структуры данных?
2. Какие виды динамических структур данных существуют?
3. Что такое «Кольцо»?
4. Какие операции для работы с кольцом необходимы?

СПИСОК ИСПОЛЬЗОВАННОЙ И РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ

Основная литература

1. Костюкова, Н.И. Программирование на языке Си [Электронный ресурс]: методические рекомендации и задачи по программированию / Н.И. Костюкова. – Электрон. текстовые данные. – Новосибирск: Сибирское университетское издательство, 2017. – 160 с. – 978-5-379-02016-3. – Режим доступа: <http://www.iprbookshop.ru/65289.html>. – ЭБС «IPRbooks»
2. Поляков, А.Ю. Программирование [Электронный ресурс] : практикум / А.Ю. Поляков, А.Ю. Полякова, Е.Н. Перышкова. – Электрон. текстовые данные. – Новосибирск: Сибирский государственный университет телекоммуникаций и информатики, 2015. – 55 с. – 2227-8397. – Режим доступа: <http://www.iprbookshop.ru/55494.html>. – ЭБС «IPRbooks»
3. Иванова, Г.С. Программирование: основы алгоритмизации и процедурное программирование, объектно-ориентированное программирование: учебник для вузов. – 2-е изд., стер. – М.: Кнорус, 2014. – 425 с. – (Бакалавриат). 588 с.
4. Керниган, Б. Язык программирования Си / Б. Керниган, Д. Ритчи. – М.: Вильямс, 2017. – 288 с.

Дополнительная литература

5. Шилдт, Г. Полный справочник по Си / Г. Шилдт. – М.: Вильямс, 2009. – 704 с.
6. Подбельский, В.В. Программирование на языке Си / В.В. Подбельский, С.С. Фомин. – М.: Финансы и статистика, 2009. – 600 с.
7. Голицына, О.Л. Основы алгоритмизации и программирования: учеб. пособие/ О.Л. Голицына, И.И. Попов. – М.: Форум, 2010. – 432 с.
8. Дейтел, Х. Как программировать на С: [пер. с англ.]/ Х. Дейтел, П. Дейтел. – М.: БИНОМ-ПРЕСС, 2010. – 1456 с.
9. Булатицкий, Д.И. Алгоритмические языки и программирование: учеб. пособие/ Д.И. Булатицкий. – Брянск: БГТУ, 2005.– 76с.
10. Уэйт, М. Язык Си. Руководство для начинающих: [пер. с англ.]/М. Уэйт, С. Прайта, Д. Мартин. – М.: Мир, 1988. – 512 с.

11. Давыдов, В.Г. Программирование и основы алгоритмизации: учеб. пособие для вузов/В.Г. Давыдов. – М.: Высш. шк., 2003. – 448 с.
12. Березин, Б.И. Начальный курс С и С++. /Б.И Березин, С.Б. Березин. – М.: Диалог-МИФИ, 2005. – 288с.
13. Павловская, Т.А. С/С++. Программирование на языке высокого уровня: учебник для вузов / Т.А Павловская. – СПб: Питер, 2007. – 460 с.
14. 10. Павловская, Т.А. С/С++. Структурное и объектно-ориентированное программирование: практикум. / Т.А. Павловская, Ю. А. Щупак – СПб.: Питер, 2011. – 352 с.
15. Керниган, Б. Элементы стиля программирования: [пер. с англ.]/Б. Керниган, Ф. Плотджер. – М.: Радио и связь 1984.
16. Ван Тассел, Д. Стиль, разработка, разработка, эффективность, отладка и испытания программ: [пер. с англ.]/Д. Ван Тассел. – М.: Мир, 1981

Ресурсы Интернет

17. Форум программистов «Stackoverflow». – Режим доступа: <http://stackoverflow.com/>
18. Раздел сайта CITFORUM, посвященный курсу «Программирование на языке СИ». – Режим доступа: <http://citforum.ru/programming/c/dir.shtml>
19. Раздел сайта ИНТУИТ, посвященный курсу «Основы программирования». – Режим доступа: <http://www.intuit.ru/studies/courses/2193/67/info>
20. Раздел сайта ИНТУИТ, посвященный курсу «Основы программирования на языке С». – Режим доступа: <http://www.intuit.ru/studies/courses/43/43/info>
21. Раздел сайта ИНТУИТ, посвященный курсу «Стили и методы программирования». – Режим доступа: <http://www.intuit.ru/studies/courses/40/40/info>
22. Раздел сайта KPOLYAKOV, посвященный курсу «Язык программирования Си». – Режим доступа: <http://kpolyakov.spb.ru/school/c.htm> –

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ	3
ГЛАВА 1. УКАЗАТЕЛИ. ГРАФИКА И АНИМАЦИЯ	5
Лабораторная работа №1. Работа с указателями	5
Лабораторная работа №2. Использование графических примитивов для создания простых изображений	13
Лабораторная работа №3. Формирование сложных параметризованных изображений с помощью пользовательских функций	31
Лабораторная работа №4. Простейшая анимация	43
Лабораторная работа №5. Интерактивная анимация	49
Лабораторная работа №6. Интерактивная анимация нескольких объектов	59
ГЛАВА 2. ДИНАМИЧЕСКИЕ МАССИВЫ	69
Лабораторная работа №7. Динамические двумерные массивы в непрерывной памяти (с постоянной длиной строки и с пересчётом индексов вручную)	69
Лабораторная работа №8. Рваные массивы	80
Лабораторная работа №9. Организация библиотеки для работы с «рванными» массивами	84
ГЛАВА 3. РАБОТА СО СТРОКАМИ	90
Лабораторная работа №10. Работа со строками	90
Лабораторная работа №11. Стандартные функции для работы со строками	96
ГЛАВА 4. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ	102
Лабораторная работа №12. Особенности передачи аргументов и возврата значений	102
Лабораторная работа №13. Динамическая структура данных «стек». Реализация в динамическом массиве	107
Лабораторная работа №14. Динамическая структура данных «стек». Реализация в связном списке	114
Лабораторная работа №15. Динамическая структура данных «Очередь»	120

Лабораторная работа №16. Динамическая структура данных «Список с произвольным доступом»	126
Лабораторная работа №17. Динамическая структура данных «Кольцо».....	137
СПИСОК ИСПОЛЬЗОВАННОЙ И РЕКОМЕНДУЕМОЙ ЛИТЕРАТУРЫ.....	144

Учебное издание

**Дмитрий Иванович Булатицкий
Елизавета Викторовна Коптенок
Руслан Александрович Исаев
Алексей Олегович Радченко**

**ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ СИ:
УПРАВЛЕНИЕ ПАМЯТЬЮ, ГРАФИКА
И ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ**

Редактор издательства
Компьютерный набор

Л.Н. Мажугина
Е.В. Коптенок

Темплан 2018г., п. 20

Подписано в печать 19.02.2019 Формат 60х84 1/16. Бумага офсетная. Офсетная
печать. Усл. печ.л. 8,95. Уч.-изд.л. 8,95. Тираж 300 экз. Заказ .

Издательство Брянского государственного технического университета
241035, г. Брянск, бульвар 50 лет Октября, 7, БГТУ, тел. 58-82-49
Лаборатория оперативной полиграфии БГТУ, ул. Институтская, 16