МИНОБРНАУКИ РОССИИ

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

Г. В. РАЗУМОВСКИЙ, И. А. ХАХАЕВ

ОБРАБОТКА СТРУКТУР И СПИСКОВ НА ЯЗЫКЕ СИ

Учебно-методическое пособие

Санкт-Петербург
Издательство СПбГЭТУ «ЛЭТИ»
2020

УДК:004.43(07)
ББК 9 973.2-018.197
Р 17:

Разумовский Г. В., Хахаев И. А.

Обработка структур и списков на языке Си: учеб.-метод. пособие к практическим занятиям. СПб.: Изд-во СПбГЭТУ «ЛЭТИ», 2020.48 с.

ISBN 978-5-7629-

Содержит описания практических занятий, связанных с изучением в компьютерном классе основ обработки пользовательских типов данных и динамических структур данных на языке программирования Си. Представлены краткие сведения, примеры и задания по использованию данных структурного типа, массивов структур, линейных и кольцевых списков, стеков и очередей.

Предназначено для студентов бакалавриата направления 09.03.01 «Информатика и вычислительная техника».

Рецензент:.

УДК:004.43(07)
ББК 9 973.2-018.197

Утверждено

редакционно-издательским советом университета
в качестве учебно-методического пособия

ISBN 978-5-7629-

ВВЕДЕНИЕ

Практические занятия являются одним из элементов учебного процесса, связанного с изучением языка программирования С (Си). На них студенты знакомятся с особенностями обработки пользовательских типов данных и динамических информационных структур, отлаживают программы в соответствии с указанным заданием. Занятия разбиты на темы, каждая из которых посвящена изучению одного из вариантов информационных структур. В каждой теме содержатся краткие теоретические сведения, примеры обработки информационных структур и задания для написания программ. При составлении заданий частично использованы задачи из учебника В. В. Подбельского и С. С. Фомина [1].

Подготовка к практическому занятию заключается в изучении лекционного материала, в котором изложена данная тема, в проработке примеров, представленных в пособии, и в решении задачи в соответствии вариантом, указанным преподавателем для данной темы. Результатом такой подготовки должен быть электронный отчет, в котором представлена схема алгоритма и текст программы для выбранной задачи. На практическом занятии студенты должны дополнительно решить не менее двух задач и по каждой из них представить отчет в электронном виде.

При решении задач обработки динамических информационных структур (списков) рекомендуется графически изображать элементы списка и связи между ними, а также последовательность переназначения связей. Этот прием помогает понять суть производимых операций и избежать ошибок времени выполнения.

Для написания и отладки программы студенты на своих компьютерах могут использовать любые операционные системы, редакторы и среды разработки. Однако для совместимости с программным обеспечением компьютерного класса рекомендуется использовать свободно распространяемую среду разработки Code::Blocks. Она может быть установлена с сайта *www.codeblocks.org/download/*. Версия для Windows, не требующая установки (-mingw-nosetup), распаковывается в специально созданную папку, а ее запуск обеспечивается файлом CbLauncher. Для упрощения работы с Code::Blocks можно в меню Панели инструментов отключить (снять «галочки») для всех панелей инструментов, кроме трех: `Главная`, `Compiler` и `CodeCompletion`. Далее в пункте главного меню `Настройки` в диалоге `Компилятор...` на вкладке `Флаги компилятора` следует включить два режима – `Создать отладочную информацию` и `Enable all common compiler warnings` (обязательно).

СТРУКТУРЫ, МАССИВЫ СТРУКТУР

Для описания (моделирования) объектов реального мира рассмотренные ранее базовые и пользовательские типы данных в Си (символы, числа, массивы, строки) недостаточны. Объекты реального мира делятся на категории (типы, классы), у каждой такой категории есть название и набор свойств. Объекты одной категории имеют одинаковые наборы свойств, но отличаются значениями этих свойств.

Для моделирования объектов реального мира в Си существует возможность создать новый пользовательский тип данных, называемый «структура». Структура в Си – объединение в единое целое множества именованных элементов (компонентов) данных [1]. В отличие от массива, состоящего из однотипных элементов, элементы структуры могут быть разных типов и должны иметь различные имена. Каждый такой элемент (компонент) называется полем структуры.

При использовании структур в программах структурные типы данных в зависимости от назначения могут быть определены как глобальные (вне всех функций модуля), так и как локальные в коде какой-то функции. Глобальный структурный тип доступен всем функциям модуля.

Для описания структурного нужно определить имя типа, а также имена и типы полей структуры.

В качестве примера рассмотрим структурный тип, предназначенный для хранения информации о студентах [2]. Пусть для описания студента используются его фамилия и имя, пол, год рождения, номер (код) специальности, год обучения (курс), номер группы и средний балл за указанный год обучения. Тогда возможно следующее описание структурного типа.

```
struct student{
    char name[MAX_NAME_LEN];
    char gender; /* 'm' or 'f' */
    int year_of_birth;
    char spec[MAX_GROUP_LEN];
    int year;
    char ngroup[MAX_GROUP_LEN];
    float average;
};
```

Здесь именем структурного типа является `student`, а ключевое слово `struct` является спецификатором структурного типа. В фигурных скобках описываются типы и имена полей, разделенные символом ”;”. Длины текстовых полей в данном примере задаются именованными константами, которые могут быть определены с использованием директивы препроцессора `#define`. При использовании динамических массивов после объявления переменной структурного типа необходимо для таких полей выделять память с помощью функций `calloc()`, `malloc()` или

realloc() с проверкой успешности выполнения этой операции. Следует обратить внимание, что при описании структурного типа задание каких-либо значений полей недопустимо. Инициализировать поля можно только во время объявления переменной структурного типа. Например,

```
struct student st {”Петров Иван”,'m',2003,”09.03.01”,1, ”1305”,
4.5};
```

Чтобы задать синоним структурного типа и обеспечить возможность его модификации (включая изменение имени), используется директива typedef. Она начинается с ключевого слова typedef, за которым идет спецификатор типа, и заканчивается идентификатором, который становится синонимом для указанного типа, например:

```
typedef struct student studs;
```

Теперь при объявлении переменных можно использовать более короткий вариант studs st;

В дальнейшем вместо термина «имя переменной структурного типа» будем использовать термин «имя структуры», а вместо термина «переменная структурного типа» – термин «структура».

Поля структуры являются обычными переменными. Для доступа к полю нужно использовать составное имя <имя_структуры>.<имя_поля>. В этом случае, если поля различных структур имеют одинаковые имена, никаких конфликтов не возникает. Для описанной выше структуры вывод имени студента будет выглядеть как printf("%s\n",st.name);, а ввод года рождения – как scanf("%d", &st.year);.

Если поле переменной структурного типа является массивом, то для доступа к элементу такого массива нужно использовать синтаксис <имя_структуры>.<имя_поля>[индекс], например:

```
printf("%c",st.spec[2]);.
```

Важной особенностью работы со структурами является возможность присваивания однотипных структур. При этом все поля структуры в левой части оператора присваивания получают значения соответствующих полей структуры в правой части оператора присваивания. Так, если определены переменные

```
studs st0, st1;
```

и заданы значения полей структуры st0, то разрешена операция

```
st1 = st0;
```

Массив структур определяется аналогично массиву переменных любого скалярного типа, например

```
studs stud_array[MAX_SIZE];
```

Доступ к отдельной структуре как к элементу такого массива осуществляется с использованием индекса (например, st1=stud_array[3];), а доступ к полю

элемента массива структур обеспечивается аналогично доступу к полю отдельной структуры (start=stud_array[3].year;).

Рассмотрим пример формирования и вывода элементов массива структур, в котором задано количество элементов массива, а значения полей структур вводятся с клавиатуры.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX_NAME_LEN 64
#define MAX_GROUP_LEN 10
#define N 4
#define OS_TYPE linux
#ifdef OS_TYPE
    #define CLS system("clear")
#else
    #define CLS system("cls")
#endif // OS_TYPE
//Определение структурного типа
struct student{
    char name[MAX_NAME_LEN];
    char gender; //'m' or 'f'
    int year_of_birth;
    char spec[MAX_GROUP_LEN];
    int year;
    char group[MAX_GROUP_LEN];
    float average;
};
//Определение пользовательского типа
typedef struct student studs;

int main()
{
    studs st1[N];
    int slen, i;
    char s1[MAX_NAME_LEN];
//Чтение данных с клавиатуры и заполнение структуры
    for(i=0;i<N;i++)
    {
        CLS;
        printf("Element: %d, estimated: %d\n\n", i+1, N-i-1);
        printf("Enter lastname firstname: ");
        fgets(s1,MAX_NAME_LEN,stdin);
        slen=strlen(s1);
        s1[slen-1]='\0';
        strcpy(st1[i].name,s1);
        printf("Enter gender status (m or f): ");
        st1[i].gender=getchar();
        printf("Enter year of birth (YYYY): ");
        scanf("%d",&st1[i].year_of_birth);
```

```
        getchar();
        printf("Enter speciality code: ");
        fgets(s1,MAX_GROUP_LEN,stdin);
        slen=strlen(s1);
        s1[slen-1]='\0';
        strcpy(st1[i].spec,s1);
        printf("Enter year of education (1-5): ");
        scanf("%d",&st1[i].year);
        getchar();
        printf("Enter group code: ");
        fgets(s1,MAX_GROUP_LEN,stdin);
        slen=strlen(s1);
        s1[slen-1]='\0';
        strcpy(st1[i].group,s1);
        printf("Enter average rating (0.00-5.00): ");
        scanf("%f",&st1[i].average);
        getchar();
    }
    puts("Array is ready!");
//Форматированный вывод полей структуры
    printf("|%20s |%6s|%6s |%10s |%6s |%6s |%5s|\n",
    "fullname","gender","year","code","course","group","ball");
    for(i=0;i<N;i++)
    printf("|%20s |   %c  |%6d |%10s |%6d |%6s |%5.3f|\n",
        st1[i].name,st1[i].gender,st1[i].year_of_birth,st1[i].spec,
        st1[i].year,st1[i].group,st1[i].average);
    return 0;
}
```

В данном примере предельные длины строк и размер массива структур определены в директивах `#define`. Для очистки экрана после ввода значений полей для очередного элемента массива структур создан макрос `CLS`, который изменяется в зависимости от значения переменной `OS_TYPE`. Если для переменной `OS_TYPE` установлено значение `linux`, то для очистки экрана вызывается функция `system()` (описанная в `stdlib.h`) с аргументом `clear`, что обеспечивает очистку экрана в операционных системах семейства Linux и Mac OS X. Если значение `OS_TYPE` не установлено, функция `system()` вызывается с аргументом `cls`, что обеспечивает очистку экрана в операционных системах семейства Windows.

На каждом шаге цикла выводится номер текущего элемента массива структур и количество оставшихся элементов.

В зависимости от типа поля структуры для получения значения используются функции ввода символа, (`getchar()`), ввода числа (`scanf()`) или ввода строки (в данном случае – `fgets()`).

После заполнения массив структур выводится в виде таблицы со строкой, содержащей названия полей.

В сформированном массиве структур можно выполнять поиск элементов по значению какого-либо поля, перестановку и сортировку элементов массива.

Если размер массива структур до выполнения программы является неизвестным, то следует использовать операции работы с динамической памятью.

В этом случае массив структур объявляется как указатель на переменную структурного типа (первый элемент массива)

```
studs *st1;
```

Если массив структур формируется при вводе значений полей с клавиатуры, то очевидным способом решения данной задачи является выделение памяти для массива из одного элемента (`st1=(studs*)malloc(sizeof(studs))`) с последующим добавлением памяти для каждого следующего элемента массива с помощью функции `realloc()`. Однако такой подход требует постоянной проверки успешности выполнения каждой попытки выделения памяти, а также приводит к нежелательному эффекту, известному как «фрагментация кучи».

Другим вариантом может быть выделение памяти под заведомо большое (несколько десятков) элементов массива структур, подсчет количества реально введенных элементов и выполнение `realloc()` в сторону уменьшения объема выделенной памяти. В любом из приведенных выше вариантов требуется предусмотреть признак окончания ввода элементов массива.

Можно частично решить проблемы, связанные с формированием массивов структур в динамической памяти, если данные для полей элементов массива структур считывать из файла. Для этого используется файл с известной структурой (форматом), в котором каждая строка соответствует одному элементу массива структур. Данные для полей разделяются каким-либо символом-разделителем (не пробелом). Такой формат текстового файла называется CSV – «Comma separated values». Возможными разделителями могут быть любые символы, не являющиеся буквами и цифрами и не встречающиеся в содержании полей структуры. Чаще всего в качества разделителя используется точка с запятой “`;`”. Такие файлы легко создавать с помощью электронной таблицы или в простом текстовом редакторе.

При использовании файла для формирования массива структур для заполнения полей каждого элемента массива требуется прочитать строку, разделить ее на элементы массива строк, а затем элементы массива строк преобразовать в значения полей элемента массива структур.

Если количество строк в файле заранее неизвестно, то перед преобразованием строк файла в поля элементов массива структур можно подсчитать количество строк и соответственно определить количество элементов массива структур.

Пусть в текстовом файле формата CSV (с именем data01.txt) записаны модельные данные о студентах в соответствии с ранее определенным структурным типом.

```
Last James;m;2000;380401;2;g7307;4.53
Mae Vanessa;f;1990;240502;1;g1208;4.83
Chang Jeckie;m;1999;131313;4;g0228;3.75
Stone Sharonne;f;2001;441500;3;o5301;3.00
McCartney Pol;m;1999;123456;2;o5500;4.15
Howston Witney;f;1987;654321;1;g3322;4.00
```

В примере, показанном далее, определяется количество строк в исходном файле, формируется массив структур в динамической памяти, и выводится список студентов, чей средний балл превышает заданный. Выводимый список отсортирован по убыванию года рождения.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Предельные длины строк заданы явно
#define MAXLEN 128
#define MAX_NAME_LEN 64
#define MAX_GROUP_LEN 10
// Определение структурного типа
struct student{
    char name[MAX_NAME_LEN];
    char gender; //'m' or 'f'
    int year_of_birth;
    char spec[MAX_GROUP_LEN];
    int year;
    char group[MAX_GROUP_LEN];
    float average;
};
// Определение пользовательского типа
typedef struct student studs;
// Функция очистки памяти для динамического массива строк
void clear_strarray(char **str, int n);
// Функция разделения строки по заданному разделителю
char **simple_split(char *str, int length, char sep);
// Функция сортировки массива структур по числовому полю
void sort_structs(studs *sarray, int n);
// Функция вычисления общего среднего балла
double avg_ball(studs *sarray, int n);

int main()
{
    studs *st1;
    int slen,i,n,len1,len2,len3;
    double ball;
    char **s2=NULL;
    char s1[MAXLEN];
```

```
    char sep;
    FILE *df;

    n=0;
    df=fopen("data01.txt","r");
    if(df!=NULL)
    { // Подсчет строк в файле
        while(fgets(s1,MAXLEN,df)!=NULL) n++;
        rewind(df); // Возврат в начало файла
        st1=(studs*)malloc(n*sizeof(studs));
        if(st1)
        {
//Чтение данных из файла и заполнение полей структур
            sep=';';
            for(i=0;i<n;i++)
            {
                fgets(s1,MAXLEN,df);
                slen=strlen(s1);
                s1[slen-1]='\0';
                slen=strlen(s1);

                s2=simple_split(s1,slen,sep);
                if(s2!=NULL)
                {
                    len1=strlen(s2[0]);
                    len2=strlen(s2[3]);
                    len3=strlen(s2[5]);

                    strncpy(st1[i].name,s2[0],len1);
                    st1[i].gender=s2[1][0];
                    st1[i].year_of_birth=atoi(s2[2]);
                    strncpy(st1[i].spec,s2[3],len2);
                    st1[i].year=atoi(s2[4]);
                    strncpy(st1[i].group,s2[5],len3);
                    st1[i].average=atof(s2[6]);
                    clear_strarray(s2,7);
                }
                else
                {
                    i=n;
                    puts("Row data not available!");
                }
            }
        }
        else puts("Out of memory!");
        fclose(df);
    }
    else perror("Data error!");
// Обработка массива структур
    if(st1&&n)
```

```
    {
        sort_structs(st1,n);
        ball=avg_ball(st1,n);
        printf("|%20s |%6s|%6s |%10s |%6s |%6s |%5s|\n",
       "fullname","gender","year","code","course","group","ball");
        for(i=0;i<n;i++)
        {
            if(st1[i].average>ball)
            printf("|%20s |   %c  |%6d |%10s |%6d |%6s |%5.3f|\n",
            st1[i].name,st1[i].gender,st1[i].year_of_birth,
            st1[i].spec,st1[i].year, st1[i].group,st1[i].average);
        }
        free(st1);
        st1=NULL;
    }
    else puts("No data found!");
    return 0;
}
```

В случае удачного открытия файла подсчитывается количество строк в этом файле, после чего выделяется нужный объем памяти для массива структур. При успешном выделении памяти строки из файла считываются по одной в переменную `s1`, каждая строка файла разделяется на элементы промежуточного массива строк `s2` по разделителям с помощью функции `simple_split()`, и в зависимости от типа поля элемента массива структур выполняется преобразование элемента массива строк в поле отдельной структуры. После заполнения полей элемента массива структур промежуточный массив строк очищается с помощью функции `clear_strarray()`. Функции `simple_split()` и `clear_strarray()` студенты могут реализовать самостоятельно на основе ранее изученного материала. По завершению формирования массива структур файл с входными данными закрывается.

Сортировка массива структур по убыванию года рождения выполняется в функции `sort_structs()`, реализующей метод вставок. Код этой функции описан ниже.

```
void sort_structs(studs *sarray, int n)
{
    studs tmp;
    int i,j;
    for(i=0;i<n;i++)
    {
        tmp=sarray[i];
 for(j=i-1;(j>=0)&&(sarray[j].year_of_birth<tmp.year_of_birth);j--)
 sarray[j+1]=sarray[j];
        sarray[j+1]=tmp;
    }
}
```

После сортировки выводятся только те элементы массива, в котором значения поля, содержащего средний балл студента, превышают значение общего среднего балла. Вычисление общего среднего балла выполняется в функции `avg_ball()`, код которой приведен далее.

```
double avg_ball(studs *sarray, int n)
{
    double avgb;
    int i;
    avgb=0.0;
    for(i=0;i<n;i++) avgb=avgb+sarray[i].average;
    return avgb/n;
}
```

При заданных входных данных приведенная программа формирует следующую таблицу:

```
|        fullname |gender|  year |      code |course | group | ball|
|      Last James |   m  |  2000 |    380401 |     2 | g7307 |4.530|
|   McCartney Pol |   m  |  1999 |    123456 |     2 | o5500 |4.150|
|     Mae Vanessa |   f  |  1990 |    240502 |     1 | g1208 |4.830|
```

Программа завершает свою работу, освобождая память, выделенную для массива структур, с использованием функции `free()`.

Задания по теме.

Перед выполнением задания по данной теме следует выбрать предметную область и спроектировать структурный тип для описания объектов предметной области. Структурный тип должен иметь не менее пяти полей хотя бы трех различных типов (все варианты чисел считаются как один тип – числовой). Каждый студент должен написать программу и выполнить по одному заданию из пунктов 1-4 и 5-10.

Для описанной структуры необходимо разработать:

1. Функцию построения одномерного массива структур с заданным количеством элементов и вводом данных с клавиатуры.

2. Функцию формирования одномерного массива структур до тех пор, пока при вводе данных с клавиатуры не появится заданное значение для некоторого поля структуры.

3. Функцию формирования одномерного массива структур с организацией диалога с пользователем, в котором фиксируется момент окончания ввода данных.

4. Функцию построения массива структур из файла с помощью функции блочного чтения `fread()`.

5. Функцию выгрузки массива структур в файл с использованием функции блочной записи `fwrite()`.

6. Функцию выгрузки массива структур в файл формата CSV с заданным разделителем.

7. Функцию дополнения уже существующего массива структур новыми структурами;

8. Функцию поиска структур с заданным значением выбранного элемента.

9. Функцию вывода на экран содержимого массива структур блоками по N элементов.

10. Функцию вывода элементов массива структур с заданными признаками (диапазон значений полей, наличие заданной подстроки и т.п.).

УКАЗАТЕЛИ НА СТРУКТУРЫ

При использовании указателей на структуры появляется возможность размещать структуры в динамической памяти («куче») поскольку размещение структур в стеке быстро приводит к его переполнению вследствие большого объема памяти, требуемой для каждой структуры. Кроме того, применение указателей на структуры целесообразно при передаче структур в качестве параметров функций, поскольку при передаче структуры в функцию по значению в стеке создается копия структуры. При передаче структуры в функцию через указатель (по ссылке) ее копия не создается, структура обрабатывается функцией по тому же адресу, по которому она была размещена в динамической памяти.

Память для структуры, размещаемой по указателю, выделяется обычным образом с помощью вызова функции `malloc()` и последующей проверкой результата.

Указатель на структуру объявляется с помощью звездочки (*), которую помещают перед именем переменной структурного типа. Для структурного типа, заданного в предыдущем разделе, можно определить указатель как

```
studs *stud0;
```

и выделить память для структуры в операторе

```
stud0=(studs*)malloc(sizeof(studs);
```

После окончания работы следует очистить выделенную память `free(stud0);`.

При обращении к полям структуры, размещенной в динамической памяти, используется либо синтаксис `(*stud0).name`, либо упрощенный вариант `stud0->name` (следует помнить, что конструкция `*stud0.name` является указателем на поле структуры `stud0`).

Размещаемый в динамической памяти массив структур, также размещенных в динамической памяти, задается как указатель на указатель (например, `studs **stud0;`).

В примере далее приведена программа для вывода списка студентов, имеющих средний балл выше, чем общий средний балл, при этом список отсортирован по убыванию года рождения. В отличие от предыдущего примера, в полях структуры используются динамические массивы. Все структуры и массив структур размещается в динамической памяти, а обработка и вывод результатов реализованы в виде функций.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXLEN 128
// Определение структурного типа
struct student{
        char *name;
        char gender; // 'm' or 'f'
        int year_of_birth;
        char *spec;
        int year;
        char *group;
        float average;
};
// Определение пользовательского типа
typedef struct student studs;
// Функция очистки памяти для динамического массива строк
void clear_strarray(char **str, int n);
// Функция разделения строки по заданному разделителю
char **simple_split(char *str, int length, char sep);
// Функция печати заголовка таблицы результатов
void print_header();
// Функция заполнения полей структуры из массива строк
studs *fill_struct(char **str);
// Функция сортировки массива структур по числовому полю
void sort_structs(studs **sarray, int n);
// Функция вычисления общего среднего балла
double avg_ball(studs **sarray, int n);
// Функция очистки памяти для структуры
void clear_struct(studs *str0);
// Функция вывода элемента массива структур
void out_struct(studs *str0);

int main()
{
    studs **studarr=NULL;
    int slen,i,n,count;
    double ball;
    char **s2=NULL;
    char s1[MAXLEN];
    char sep;
    FILE *df;
```

```
// Открытие файла
   df=fopen("data01.txt","r");
    if(df!=NULL)
    {
        n=0;
// Подсчет строк в файле
        while((fgets(s1,MAXLEN,df))!=NULL) n++;
        rewind(df); // Возврат в начало файла
// Выделение памяти для массива структур
        studarr=(studs**)malloc(n*sizeof(studs*));
        if(studarr!=NULL)
        {
            sep=';';
            puts("Initial array:");
            print_header();
            for(i=0,count=0;i<n;i++,count++)
            {
                fgets(s1,MAXLEN,df);
                slen=strlen(s1);
                s1[slen-1]='\0';
                slen=strlen(s1);

                s2=simple_split(s1,slen,sep);
                if(s2!=NULL)
                {
                    studarr[i]=fill_struct(s2);
                    if(studarr[i]!=NULL) out_struct(studarr[i]);
                    else
                    {
                        puts("Structure not allocated!");
                        i=n;
                    }
                }
                else puts("Error at data reading!");
            }
        }
        else puts("Out of memory!");
        fclose(df);
    }
    else perror("Data error!");
    if(studarr&&(count<n))
    {
        for(i=0;i<count;i++) clear_struct(studarr[i]);
        free(studarr);
        studarr=NULL;
    }
    if(studarr&&n)
    {
        sort_structs(studarr,n);
        ball=avg_ball(studarr,n);
```

```
        printf("\nProcessed array:\n");
        print_header();
        for(i=0;i<n;i++)
        {
            if(studarr[i]->average>ball) out_struct(studarr[i]);
            clear_struct(studarr[i]);
        }
        free(studarr);
        studarr=NULL;
    }
    else puts("No data found!");
    return 0;
}
```

При успешном открытии исходного файла в первую очередь вычисляется количество строк в этом файле, затем выделяется память для хранения соответствующего количества указателей на структуры. Если память выделилась, строки из файла считываются по одной в промежуточную переменную `s1` и по заданному разделителю выделяются элементы служебного динамического массива строк `s2` с помощью функции `simple_split()`. Количество элементов массива в этой функции определяется по количеству символов-разделителей в каждой строке исходного файла.

Для каждого элемента массива структур память выделяется в «куче», после чего поля элемента массива структур заполняются данными из массива строк `s2`. Задача формирования элемента массива структур и получения значений полей выполняется функцией `fill_struct()`, код которой приведен далее.

```
studs *fill_struct(char **str)
{
    studs *str0=NULL;

    str0=(studs*)malloc(sizeof(studs));
    if(str0!=NULL)
    {
        str0->name=str[0];
        str0->gender=*str[1];
        str0->year_of_birth=atoi(str[2]);
        str0->spec=str[3];
        str0->year=atoi(str[4]);
        str0->group=str[5];
        str0->average=atof(str[6]);
    }
    return str0;
}
```

При удачном формировании элемента массива структур он выводится в форматированном виде с помощью функции `out_struct()`.

```
void out_struct(studs *str0)
```

```
{
    printf("|%20s |   %c  |%6d |%10s |%6d |%6s |%5.3f|\n",
        str0->name,str0->gender,str0->year_of_birth,str0->spec,
        str0->year,str0->group,str0->average);
}
```

Для вывода названий полей структур используется функция `print_header()`.

```
void print_header()
{
    printf("|%20s |%6s|%6s |%10s |%6s |%6s |%5s|\n",
       "fullname","gender","year","code","course","group","ball");
}
```

Если очередная попытка выделения памяти оказывается неудачной и элемент массива структур не создается, выводится сообщение и формирования следующего элемента массива структур уже не происходит, цикл чтения и обработки строк файла завершается. В переменной `count` сохраняется количество сформированных элементов массива структур. Если после завершения цикла обработки строк исходного файла количество элементов массива структур меньше, чем количество строк, память для всех ранее созданных элементов массива структур очищается с использованием функции `clear_struct()`, а затем очищается массив структур. Код функции `clear_struct()` приведен далее.

```
void clear_struct(studs *str0)
{
    free(str0->name);
    free(str0->group);
    free(str0->spec);
    str0->name=NULL;
    str0->group=NULL;
    str0->spec=NULL;
    free(str0);
    str0=NULL;
}
```

В функции `clear_struct()` сначала очищается динамическая память, выделенная для текстовых полей, а затем обнуляются их указатели, хранящиеся в структуре.

Если массив структур успешно сформирован, он сортируется по убыванию поля, содержащего год рождения, а затем выводятся только те элементы отсортированного массива, у которых поле со средним баллом имеет значение, превышающее значение общего среднего балла.

Сортировка выполняется в функции `sort_structs()` с использованием метода вставок. Сравните приведенный код с кодом функции сортировки из предыдущего раздела.

```
void sort_structs(studs **sarray, int n)
```

```
{
    studs *tmp;
    int i,j;

    for(i=0;i<n;i=i+1)
    {
        tmp=sarray[i];
for(j=i-1;(j>=0)&&((sarray[j]->year_of_birth)<
(tmp->year_of_birth));j--)
        sarray[j+1]=sarray[j];
        sarray[j+1]=tmp;
     }
}
```

Вычисление среднего балла выполняется в функции `avg_ball()`. Сравните приведенный далее код с кодом функции вычисления среднего балла из предыдущего раздела.

```
double avg_ball(studs **sarray, int n)
{
    double avgb;
    int i;

    avgb=0.0;
    for(i=0;i<n;i++) avgb=avgb+sarray[i]->average;
    return avgb/n;
}
```

После вывода каждого элемента массива структур, соответствующего условию задачи, следует освободить память, выделенную для этого элемента (функция `clear_struct()`), а после завершения работы – освободить память, выделенную для массива структур.

Задания по теме.

Для программы, созданной в предыдущем задании, добавить одну из следующих функций:

1. Функцию сортировки по значению текстового поля методом парных перестановок в лексикографическом порядке.

2. Функцию сортировки по значению числового поля методом парных перестановок по возрастанию.

3. Функцию сортировки по значению текстового поля методом вставок в лексикографическом порядке.

4. Функцию сортировки по значению текстового поля методом вставок в обратном лексикографическом порядке.

5. Функцию сортировки по значению текстового поля методом прямого выбора в лексикографическом порядке.

6. Функцию сортировки по значению текстового поля методом прямого выбора в обратном лексикографическом порядке.

7. Функцию сортировки по значению числового поля методом прямого выбора по возрастанию.

8. Функцию сортировки по значению числового поля методом прямого выбора по убыванию.

9. Функцию выборки из таблицы нечетных строк с упорядочиванием их по убыванию значения какого-либо поля.

10. Функцию выборки из таблицы четных строк с упорядочиванием их по возрастанию значения какого-либо поля.

ОДНОСВЯЗНЫЕ (ОДНОНАПРАВЛЕННЫЕ) СПИСКИ

Список – совокупность объектов (элементов), в которой каждый объект содержит информацию о размещении связанного с ним объекта (объектов). Список является динамической информационной структурой (динамической структурой данных), которая формируется в процессе выполнения программы. Для таких структур операции «удалить и/или добавить элемент» не связаны с глобальной передислокацией данных, которая характерная для массивов данных различных типов.

При хранении списка в памяти размещение объектов определяется их адресами.

Операции с элементами списка не зависят от внутренней структуры элементов, поэтому список можно рассматривать как абстрактный тип данных (АТД).

Элементы списка являются структурами. Обязательным полем в элементе списка является указатель на структуру такого же типа. Остальные поля являются информационными.

С элементами списка допустимы следующие операции:

- добавление элемента списка;
- определение длины списка;
- вывод значений информационный полей;
- поиск элемента по значению информационного поля;
- перестановка элементов списка;
- сортировка по значению информационного поля;
- удаление элемента/освобождение памяти.

Односвязный (однонаправленный) список получается, если в элементе списка содержится указатель на следующий элемент и перемещение по элемен-

там списка возможно только в одном направлении. Если существует последний элемент списка (содержащий NULL в поле указателя на следующий элемент), то такой список называется линейным.

Для любого списка должна быть определена точка начала (указатель на первый элемент списка – «голова»). Если этот указатель NULL, то считается что список пустой.

Далее показан простой пример определения структуры элемента списка и создания списка из одного элемента и «головы», являющейся указателем на первый элемент списка.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
// Определение структурного типа
struct LNode
{
    char data[32];
    struct LNode *next;
};
// Определение пользовательского типа
typedef struct LNode LN;

int main()
{
    LN *LHead=NULL; // Определение и инициализация головы
    LN *p=NULL;

    p=(LN*)malloc(sizeof(LN));
    if(p!=NULL)
    {
        strcpy(p->data,"dataword"); // инициализация элемента
        p->next=NULL;
        LHead=p; // Связывание головы с первым элементом

        printf("%p\n", LHead);
        free(p);
    }
    return 0;
}
```

«Голова» списка, сформированная в виде в виде указателя, имеет единственное назначение – хранение адреса первого элемента списка. Однако «голова» списка может быть и структурой, содержащей другие информационные поля помимо адреса первого элемента списка. В частности, можно создать «голову» в виде структуры, хранящей адреса первого и последнего элемента, а также некий уникальный индекс последнего элемента, добавленного в список, а также общее количество элементов списка.

Рассмотрим пример формирования линейного односвязного списка с операциями добавления элемента, поиска элемента по значению информационного поля, вывода адреса и всех полей найденного элемента и удаления найденного элемента. «Голова» списка определяется как структура, содержащая уникальный индекс последнего элемента, добавленного в список, общее количество элементов списка и указатели на первый и последний элементы списка. Для контроля сделан служебный вывод полей «головы» списка до и после удаления элемента, а также адресов и значений информационных полей элементов списка. Элементы списка содержат только два поля – поле `id` с автоинкрементом и текстовое информационное поле. Поскольку значения поля `id` автоматически увеличиваются на 1 для каждого следующего элемента списка, вводить требуется только значения текстового поля. Каждый следующий элемент списка добавляется в начало списка (перед первым). Ввод заканчивается, если очередная строка является сочетанием символов «\q».

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXLEN 64
// Структура для элемента списка
struct LNode
{
    int id; // уникальное значение
    char *word;  // поле данных
    struct LNode *next; // адрес следующего элемента
};
// Структура для головы списка
struct LHead
{
    int last_id; // сколько всего было элементов
    int cnt; // количество элементов
    struct LNode *first; // адрес первого элемента
    struct LNode *last; // адрес последнего элемента
};
// Определение пользовательских типов
typedef struct LHead head;
typedef struct LNode node;
// Функция создания головы
head *make_head();
// Функция создания нового элемента
node *make_node(char *new_word, int slen);
// Функция добавления элемента в начало списка
void add_node(head *my_head, node *new_node);
// Функция поиска элемента по значению id
node *select_id(head *my_head, int n);
// Функция удаления элемента с известным адресом
```

```
void delete_node(head *my_head, node *current_node);
// Функция очистки памяти
void clear_list(head *my_head);

int main()
{
    head *ph=NULL;
    node *p=NULL;
    char my_word[MAXLEN];
    int slen,n;

    ph=make_head();

    puts("--Press \"\\q\" for terminate input--\n");
    while(strncmp(my_word,"\\q",2)!=0)
    {
        printf("%s: ","Please enter your word");
        fgets(my_word,MAXLEN,stdin);
        slen=strlen(my_word);
        my_word[slen-1]='\0';

        if(strncmp(my_word,"\\q",2)!=0)
        {
            p=make_node(my_word,slen);
            add_node(ph,p);
            printf("%d %s\n",p->id, p->word);
        }
    }
    puts("\n--Input terminated. Your data are:--");
    p=ph->first;
    while(p!=NULL)
    {
        printf("Address: %p, data: %d %s\n",p, p->id, p->word);
        p=p->next;
    }
    printf("Head fields: %d %d %p %p\n",
          ph->last_id,ph->cnt,ph->first,ph->last);
    printf("%s: ","\nPlease enter your number");
    scanf("%d", &n);

    p=select_id(ph,n);
    if(p!=NULL)
    {
        printf("Address: %p, data: %d %s\n",p, p->id, p->word);
        delete_node(ph,p);
        puts("--Data after deleting selected node --");
        p=ph->first;
        while(p!=NULL)
        {
            printf("Address: %p, data: %d %s\n",p,
```

```
                p->id, p->word);
            p=p->next;
        }
        printf("Head fields: %d %d %p %p\n",
               ph->last_id,ph->cnt,ph->first,ph->last);
    }
    else printf("ERROR! Element not exists!\n");

    clear_list(ph);
    return 0;
}
```

В данном примере сначала определяются переменные структурных типов (для «головы» списка и элемента списка эти типы различаются), затем создается «голова» списка с помощью функции `make_head()`, имеющей следующую реализацию.

```
head *make_head()
{
    head *ph=NULL;

    ph=(head*)malloc(sizeof(head));
    if(ph!=NULL)
    {
        ph->last_id=0;
        ph->cnt=0;
        ph->first=NULL;
        ph->last=NULL;
    }
    return ph;
}
```

Функция `make_head()` возвращает адрес «головы» списка (указатель на структуру). В теле функции выделяется память для хранения структуры и при успешном выделении устанавливаются начальные значения полей. Поскольку на момент создания «головы» в списке нет элементов, в качестве адресов первого и последнего элементов списка указывается NULL.

Функция `make_node()` выделяет память и инициализирует значения полей для очередного элемента списка. Результатом является адрес очередного элемента. Код функции показан далее.

```
node *make_node(char *new_word, int slen)
{
   node *new_node=NULL; // адрес создаваемого элемента
   char *someword=NULL;

    new_node = (node*)malloc(sizeof(node));
    someword=(char*)malloc((slen+1)*sizeof(char));
    if(new_node&&someword)
    {
        strcpy(someword,new_word);
```

```
        new_node->id = 1; // установка начального ID
        new_node->word=someword;
        new_node->next = NULL; // следующий элемент отсутствует
    }
    return new_node; // возвращается адрес создаваемого элемента
}
```

В этой функции происходит выделение памяти для хранения структуры очередного элемента списка и при успешном выделении памяти – формирование начальных значений информационных полей. Поле `id` устанавливается в 1, а в текстовое поле заносится строка, прочитанная из потока ввода. Если память для структуры или для строки не выделяется, функция возвращает адрес NULL.

В зависимости от условий работы и архитектуры вычислительной системы два последовательно созданных элемента списка могут размещаться в различных участках «кучи». Связи между элементами, а также между «головой» и элементами списка обеспечивается занесением адресов элементов в соответствующие поля структур. Эту задачу выполняет функция `add_node()`, реализация которой показана далее.

```
void add_node(head *my_head, node *new_node)
{
    node *q=NULL;
    int k;
    if(my_head&&new_node)
    {
        q=my_head->first;
        if(q==NULL) // список пустой
        {
            my_head->last_id=1;
            my_head->cnt=1;
            my_head->first=new_node;
            my_head->last=new_node;
        }
        else // в списке есть элементы
        {
            k=my_head->last_id+1;
            my_head->cnt++;
            new_node->id=k;
            new_node->next=q;
            my_head->first=new_node;
            my_head->last_id=k;
        }
    }
}
```

Функция `add_node()` добавляет элементы всегда в начало списка (перед первым). Возможны две ситуации – когда список пустой (нет элементов) и когда в списке есть элементы. Если в списке нет элементов, то добавляемый элемент является первым (и последним), его `id` менять не надо. В полях «головы» списка количество элементов устанавливается в 1, `id` последнего элемента также устанавливается в 1, а адреса первого и последнего элемента совпадают с адресом нового элемента. В этом случае поля самого элемента списка не меняются и

остаются со значениями, установленными при инициализации элемента. Если в списке уже есть элементы, то при добавлении каждого следующего элемента нужно увеличить на 1 значения полей `last_id` и `cnt` в «голове» списка, затем сделать текущий первый элемент следующим для добавляемого, а добавляемый сделать первым. Эти действия требуют переназначения указателей. При переназначении указателей важен порядок действий. Ситуация, когда на один и тот же элемент существует несколько указателей, является вполне допустимой, а при отсутствии указателя на элемент он становится недоступным (т. к. нигде не хранится информация о его адресе). Также при добавлении элемента требуется увеличить значение поля `id` для добавляемого элемента на основе уже имеющегося значения `last_id` в «голове» списка. Такой подход приводит к тому, что значения полей `last_id` и `cnt` в «голове» списка могут существенно отличаться после нескольких удалений и добавлений элементов.

После завершения формирования списка и контрольного вывода адресов и полей элементов выполняется поиск элемента по значению поля `id`. Значения этого поля являются уникальными и не повторяются. Функция `select_id()` возвращает адрес найденного элемента.

```
node *select_id(head *my_head, int n)
{
    node *q;
    int k;

    q=my_head->first;
    k=my_head->last_id;
    if((n>0)&&(n<=k))
    {
        while(((q->id)!=n)&&(q!=NULL)) q=q->next;
    }
    else q=NULL;
    return q;
}
```

Значение поля `id` не должно быть отрицательным или равным 0, а также не должно превышать значение `last_id`, хранящегося в «голове». В функции производится перебор элементов, начиная с первого, до тех пор, пока значение поля `id` не совпадет с заданным или не будет достигнут последний элемент списка. При неверных данных или отсутствии совпадений (если какие-то элементы были удалены) функция вернет адрес NULL.

Если элемент не найден, то программа выдает сообщение об отсутствии элемента. Если элемент найден, то выводится его адрес и значения полей, а затем найденный элемент удаляется с помощью функции `delete_node()`, код которой приведен далее.

```
void delete_node(head *my_head, node *current_node)
{
    node *q, *q1;
```

```
    q=my_head->first;
    q1=my_head->last;
    if(current_node==q)
    {
        my_head->first=current_node->next;
        current_node->next=NULL;
        free(current_node);
    }
    else
    {
        while(q!=NULL)
        {
            if(q->next==current_node)
            {
                if(current_node==q1) my_head->last=q;
                q->next=current_node->next;
                current_node->next=NULL;
                free(current_node);
            }
            else q=q->next;
        }
    }
    my_head->cnt--;
}
```

При удалении элемента нужно обработать три ситуации:

- удаляемый элемент – первый;
- удаляемый элемент – последний;
- удаляемый элемент – в середине списка.

Если удаляемый элемент – первый, то нужно сделать первым элементом тот, который следует за удаляемым путем переназначения указателя в «голове» списка, после чего удалить (установить в NULL) указатель на следующий элемент в поле `next` удаляемого элемента и освободить память, выделенную для удаляемого элемента.

Если удаляемый элемент не является первым, то производится перебор элементов списка, пока не будет найден элемент с адресом `q`, содержащий в поле `next` адрес удаляемого элемента. Если адрес удаляемого элемента совпадает с адресом последнего элемента списка, хранящимся в «голове» списка, то элемент с адресом `q` делается последним элементом путем переназначения указателя в «голове» списка. Во всех случаях производится замена указателя на следующий элемент для элемента с адресом `q` на элемент, следующий за удаляемым. Для удаляемого элемента адрес следующего элемента устанавливается в NULL, после чего очищается память, выделенная для удаляемого элемента.

После удаления элемента снова выводятся адреса и значения информационных полей элементов списка, а также значения полей «головы» списка. При удалении элемента списка значение поля `last_id` в «голове» списка не изменяется, а значение поля `cnt` уменьшается на 1.

После завершения обработки списка производится очистка памяти, выделенной для элементов списка. Эту задачу решает функция clear_list(), код которой приведен далее.

```
void clear_list(head *my_head)
{
    node *q,*q1;
    q=my_head->first;
    while(q!=NULL)
    {
        q1=q->next;
        free(q);
        q=q1;
    }
    free(my_head);
}
```

В этой функции выполняется проход по элементам списка с первого и до последнего, на каждом шаге запоминается адрес следующего элемента, а память, выделенная для текущего элемента, очищается. Последним действием освобождается память, выделенная для «головы» списка.

В кольцевом (циклическом) односвязном (однонаправленном) списке в поле next последнего элемента содержится адрес первого элемента, так что проход по списку с проверкой значения поля next на NULL уже не работает.

Для приведенной выше задачи по формированию и обработке односвязного списка в случае кольцевого списка потребуется модификация некоторых функций и части кода функции main().

Директивы препроцессора, определения структур и типов, прототипы функций, функции инициализации «головы» make_head() и создания нового элемента make_node() не изменяются.

Поскольку в кольцевом списке отсутствует признак последнего элемента в качестве адреса NULL в поле next, воспользуемся тем обстоятельством, что в структуре «головы» списка имеется поле cnt, в котором хранится количество элементов списка. Поэтому часть циклов while() можно заменить на for() для известного количества элементов списка.

Далее приведен модифицированный код функции main().

```
int main()
{
    head *ph=NULL;
    node *p=NULL;
    char my_word[MAXLEN];
    int slen,i,n;

    ph=make_head();

    puts("--Press \"\\q\" for terminate input--\n");
    while(strncmp(my_word,"\\q",2)!=0)
    {
        printf("%s: ","Please enter your word");
```

```
        fgets(my_word,MAXLEN,stdin);
        slen=strlen(my_word);
        my_word[slen-1]='\0';

        if(strncmp(my_word,"\\q",2)!=0)
        {
            p=make_node(my_word,slen);
            add_node(ph,p);
            printf("%d %s\n",p->id, p->word);
        }
    }
    puts("\n--Input terminated. Your data are:--");
    p=ph->first;
    for(i=0;i<ph->cnt;i++)
    {
        printf("Address: %p, data: %d %s\n",p, p->id, p->word);
        p=p->next;
    }
    printf("Head fields: %d %d %p %p\n",
          ph->last_id,ph->cnt,ph->first,ph->last);
    printf("%s: ","\nPlease enter your number");
    scanf("%d", &n);

    p=select_id(ph,n);
    if(p!=NULL)
    {
        printf("Address: %p, data: %d %s\n",p, p->id, p->word);
        delete_node(ph,p);
        puts("--Data after deleting selected node --");
        p=ph->first;
        for(i=0;i<ph->cnt;i++)
        {
            printf("Address: %p, data: %d %s\n",
                  p, p->id, p->word);
            p=p->next;
        }
        printf("Head fields: %d %d %p %p\n",
              ph->last_id,ph->cnt,ph->first,ph->last);
    }
    else printf("ERROR! Element not exists!\n");

    clear_list(ph);
    return 0;
}
```

Здесь добавилась переменная для параметров циклов `for()`, которые используется для вывода исходного списка и списка после удаления найденного элемента.

При добавлении элемента в список при пустом списке нужно записать адрес первого элемента в поле `next` добавляемого элемента. Поскольку элементы добавляются перед первым, при наличии элементов в списке при добавлении нового элемента ничего не изменяется. Модифицированная функция `add_node()` показана далее.

```
void add_node(head *my_head, node *new_node)
{
    node *q=NULL;
```

```
    int k;
    if(my_head&&new_node)
    {
        q=my_head->first;
        if(q==NULL) // list is empty
        {
            my_head->last_id=1;
            my_head->cnt=1;
            my_head->first=new_node;
            my_head->last=new_node;
            new_node->next=my_head->first;
        }
        else // list has some elements
        {
            k=my_head->last_id+1;
            my_head->cnt++;
            new_node->id=k;
            new_node->next=q;
            my_head->first=new_node;
            my_head->last_id=k;
        }
    }
}
```

При поиске элемента с заданным `id` проход по списку может быть реализован с использованием значения поля `cnt` в «голове» списка. Для этого в функции `select_id()` ведена дополнительная переменная и изменены условия выполнения цикла `while()`, а также тело этого цикла. Код модифицированной функции `select_id()` приведен далее.

```
node *select_id(head *my_head, int n)
{
    node *q=NULL;
    int i,k;

    q=my_head->first;
    k=my_head->last_id;
    if((n>0)&&(n<=k))
    {
        while(((q->id)!=n)&&(i<=my_head->cnt))
        {
            q=q->next;
            i++;
        }
    }
    else q=NULL;
    return q;
}
```

При удалении элемента (функция `delete_node()`), если этот элемент является первым, необходимо изменить адрес, хранящийся в поле `next` последнего элемента кольцевого писка, на адрес элемента, следующего за удаляемым. При проходе по списку используется цикл `for()` с количеством повторений, равным значению поля `cnt` в «голове» списка, поэтому при удалении любого элемента, кроме первого, нужно уменьшить значение этого поля в теле цикла. Код модифицированной функции `delete_node()` приведен далее.

```
void delete_node(head *my_head, node *current_node)
{
    node *q, *q1;
    int i;

    q=my_head->first;
    q1=my_head->last;
    if(current_node==q)
    {
        my_head->first=current_node->next;
        q1->next=current_node->next;
        current_node->next=NULL;
        free(current_node);
        my_head->cnt--;
    }
    else
    {
        for(i=0;i<my_head->cnt;i++)
        {
            if(q->next==current_node)
            {
                if(current_node==q1) my_head->last=q;
                q->next=current_node->next;
                current_node->next=NULL;
                free(current_node);
                my_head->cnt--;
            }
            else q=q->next;
        }
    }
}
```

Наконец, при очистке памяти, выделенной для элементов списка (функция `clear_list()`), также используется цикл `for()`, для которого введена дополнительная переменная – параметр цикла.

```
void clear_list(head *my_head)
{
    node *q,*q1;
    int i;

    q=my_head->first;
    for(i=0;i<my_head->cnt;i++)
    {
        q1=q->next;
        free(q);
        q=q1;
    }
    free(my_head);
}
```

Задания по теме.

Написать программу, в которой создается односвязный список и реализуется две функции из указанного ниже перечня:

1. Функция `add_lastnode()`, добавляющая элементы односвязного линейного списка всегда после последнего элемента.

2. Функция поиска элементов односвязного линейного списка по подстроке текстового поля и вывода информационных полей найденных элементов списка.

3. Функция удаления элемента односвязного списка по заданному `id`.

4. Функция вывода односвязного линейного списка в обратном порядке.

5. Функция определения количества элементов односвязного линейного списка (длины списка), если «голова» является указателем на первый элемент списка.

6. Функция перестановки элементов односвязного списка (`swap()`) с использованием промежуточной структуры.

7. Функция для вывода значений полей элементов односвязного кольцевого списка, если «голова» списка является указателем на первый элемент, а не структурой.

8. Функция поиска элемента односвязного кольцевого списка по заданному `id`, если «голова» списка является указателем на первый элемент, а не структурой.

9. Функция удаления элемента односвязного кольцевого списка с известным адресом, если «голова» списка является указателем на первый элемент, а не структурой.

10. Функция очистки памяти для односвязного кольцевого списка, если «голова» списка является указателем на первый элемент, а не структурой.

ДВУСВЯЗНЫЕ (ДВУНАПРАВЛЕННЫЕ) СПИСКИ

В двусвязном (двунаправленном) списке каждый элемент списка содержит поля для хранения адресов предыдущего и следующего элементов списка (указатели на предыдущий и на следующий элемент). Таким образом, перемещение по элементам списка возможно в двух направлениях.

В двусвязном линейном списке адрес предыдущего элемента у первого элемента списка устанавливается в NULL, а у последнего элемента списка в NULL устанавливается адрес следующего элемента.

«Голова» двусвязного списка, как и в случае односвязного списка, может быть просто указателем на первый элемент, а может быть структурой, содержащей и другие поля кроме указателя на первый элемент списка.

Рассмотрим пример формирования линейного двусвязного списка с операциями добавления элемента, поиска элемента по значению информационного поля, вывода адреса и всех полей найденного элемента и удаления найденного элемента. «Голова» списка определяется как структура, содержащая уникальный индекс последнего элемента, добавленного в список, общее количество

элементов списка и указатели на первый и последний элементы списка. Для контроля сделан служебный вывод полей «головы» списка до и после удаления элемента, а также адресов и значений информационных полей элементов списка. Элементы списка содержат только два поля — поле `id` с автоинкрементом и текстовое информационное поле. Поскольку значения поля `id` автоматически увеличиваются на 1 для каждого следующего элемента списка, вводить требуется только значения текстового поля. Ввод заканчивается, если очередная строка является сочетанием символов «\q». Элементы списка добавляются всегда после последнего (в конец списка).

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXLEN 64
// Структура для элемента списка
struct LNode
{
    int id; // уникальное значение
    char *word;  // поле данных
    struct LNode *next; // адрес следующего элемента
    struct LNode *prev; // адрес предыдущего элемента
};
// Структура для головы списка
struct LHead
{
    int last_id; // сколько всего было элементов
    int cnt; // количество элементов
    struct LNode *first; // адрес первого элемента
    struct LNode *last; // адрес последнего элемента
};
// Определение пользовательских типов
typedef struct LHead head;
typedef struct LNode node;
// Функция создания головы
head *make_head2();
// Функция создания нового элемента
node *make_node2(char *new_word, int slen);
// Функция добавления элемента в конец списка
void add_node2(head *my_head, node *new_node);
// Функция поиска элемента по значению id
node *select_id2(head *my_head, int n);
// Функция удаления элемента с известным адресом
void delete_node2(head *my_head, node *current_node);
// Функция очистки памяти
void clear_list2(head *my_head);

int main()
{
    head *ph=NULL;
    node *p=NULL;
    char my_word[MAXLEN];
    int slen,n;
```

```
    ph=make_head2();

    puts("--Press \"\\q\" for terminate input--\n");
    while(strncmp(my_word,"\\q",2)!=0)
    {
        printf("%s: ","Please enter your word");
        fgets(my_word,MAXLEN,stdin);
        slen=strlen(my_word);
        my_word[slen-1]='\0';

        if(strncmp(my_word,"\\q",2)!=0)
        {
            p=make_node2(my_word,slen);
            add_node2(ph,p);
            printf("%d %s\n",p->id, p->word);
        }
    }
    puts("\n--Input terminated. Your data are:--");
    p=ph->first;
    while(p!=NULL)
    {
        printf("Address: %p, data: %d %s\n",p, p->id, p->word);
        p=p->next;
    }
    printf("Head fields: %d %d %p %p\n",
          ph->last_id,ph->cnt,ph->first,ph->last);
    printf("%s: ","\nPlease enter your number");
    scanf("%d", &n);

    p=select_id2(ph,n);
    if(p!=NULL)
    {
        printf("Address: %p, data: %d %s\n",p, p->id, p->word);
        delete_node2(ph,p);
        puts("--Data after deleting selected node --");
        p=ph->first;
        while(p!=NULL)
        {
            printf("Address: %p, data: %d %s\n",
                  p, p->id, p->word);
            p=p->next;
        }
        printf("Head fields: %d %d %p %p\n",
              ph->last_id,ph->cnt,ph->first,ph->last);
    }
    else printf("ERROR! Element not exists!\n");

    clear_list2(ph);
    return 0;
}
```

Директивы препроцессора в начале программы ничем не отличаются от примера, показанного в предыдущем разделе. Структура, определенная для «головы» списка, тоже не претерпела изменений.

Структура для элемента списка теперь содержит два поля указателей на элементы списка – на следующий и на предыдущий.

В названия функций при определении прототипов добавлен символ «2», показывающий, что они относятся к операциям с двусвязным списком. Соответственно изменились вызовы функций в функции main().

Функция создания «головы» двусвязного списка make_head2() полностью идентична функции создания «головы» для односвязного списка, поэтому нет необходимости приводить ее код.

Функция создания и инициализации полей элемента списка make_node2() отличается наличием дополнительного поля – указателя на предыдущий элемент, который для нового элемента списка получает значение NULL. Код функции make_node2() приведен далее.

```
node *make_node2(char *new_word, int slen)
{
   node *new_node=NULL; // адрес создаваемого элемента
   char *someword=NULL;

    new_node = (node*)malloc(sizeof(node));
    someword=(char*)malloc((slen+1)*sizeof(char));
    if(new_node&&someword)
    {
        strcpy(someword,new_word);
        new_node->id = 1; // установка начального ID
        new_node->word=someword;
        new_node->next = NULL; // следующий элемент отсутствует
        new_node->prev = NULL; // предыдущий элемент отсутствует
    }
    return new_node; // возвращается адрес создаваемого элемента
}
```

Функция добавления нового элемента в список add_node2() должна учитывать наличие указателя на предыдущий элемент. Поскольку элемент добавляется всегда после последнего, то добавляемый элемент становится следующим за ранее последним. Это существенно для списка, в котором уже есть элементы, а добавление элемента в пустой двусвязный линейный список можно сделать точно так же, как и в случае односвязного линейного списка. Однако, поскольку при работе функции нужен адрес последнего элемента списка, можно воспользоваться тем, что этот адрес также хранится в «голове» списка и сохранить его в служебную переменную. При добавлении элемента важен порядок переназначения указателей, что и реализовано в коде функции add_node2(), показанном далее.

```
void add_node2(head *my_head, node *new_node)
{
    node *q=NULL,*q1=NULL;
    int k;
    if(my_head&&new_node)
    {
        q=my_head->first;
        q1=my_head->last;
        if(q==NULL) // список пустой
```

```
        {
            my_head->last_id=1;
            my_head->cnt=1;
            my_head->first=new_node;
            my_head->last=new_node;
        }
        else // в списке есть элементы
        {
            k=my_head->last_id+1;
            my_head->cnt++;
            new_node->id=k;
            q1->next=new_node;
            new_node->prev=q1;
            my_head->last=new_node;
            my_head->last_id=k;
        }
    }
}
```

Функция поиска адреса элемента списка по заданному значению поля `id` (функция `select_id2()`) полностью идентична рассмотренной ранее функции `select_id()` для односвязного линейного списка, поэтому можно использовать ранее приведенный код.

При удалении найденного элемента списка (функция `delete_node2()`) реализуются различные сценарии в зависимости от того, является ли удаляемый элемент первым, последним или находится где-то в середине списка. Поскольку каждый элемент списка связан с двумя соседними элементами, для корректного переназначения указателей удобно ввести дополнительные промежуточные переменные.

Если удаляемый элемент является первым, то элемент `q2`, следующий за ним, должен быть объявлен первым путем переназначения указателя в «голове» списка и установки в NULL указателя на предыдущий элемент для элемента `q2`. Затем указатель на следующий элемент для удаляемого элемента устанавливается в NULL, и очищается память, выделенная для удаляемого элемента. При переназначении указателей важен порядок действий. Ситуация, когда на один и тот же элемент существует несколько указателей, является вполне допустимой, а при отсутствии указателя на элемент он становится недоступным (т. к. нигде не хранится информация о его адресе). Соответственно, становятся недоступными и связанные с ним элементы списка.

Если удаляемый элемент не является первым, то производится перебор элементов списка, пока не будет найден элемент с адресом `q`, содержащий в поле `next` адрес удаляемого элемента. Если такой элемент найден, то его адрес запоминается в переменную `q2` (см. код далее). Если адрес удаляемого элемента совпадает с адресом последнего элемента списка, хранящимся в «голове» списка, то элемент с адресом `q2` делается последним элементом путем переназначе-

ния указателя в «голове» списка и установки для элемента с адресом q2 указателя на следующий элемент в NULL. Если удаляемый элемент не является последним, то в переменную q3 запоминается адрес элемента, следующего за удаляемым и производится переназначение указателей. Далее для удаляемого элемента адреса следующего и предыдущего элементов устанавливается в NULL, после чего очищается память, выделенная для удаляемого элемента.

После удаления элемента снова выводятся адреса и значения информационных полей элементов списка, а также значения полей «головы» списка. При удалении элемента списка значение поля last_id в «голове» списка не изменяется, а значение поля cnt уменьшается на 1.

```
void delete_node2(head *my_head, node *current_node)
{
    node *q,*q1,*q2,*q3;

    q=my_head->first;
    q1=my_head->last;
    if(current_node==q)
    {
        q2=current_node->next;
        my_head->first=q2;
        q2->prev=NULL;
        current_node->next=NULL;
        free(current_node);
    }
    else
    {
        while(q!=NULL)
        {
            if(q->next==current_node)
            {
                q2=current_node->prev;
                if(current_node==q1)
                {
                    my_head->last=q2;
                    q2->next=NULL;
                }
                else
                {
                    q3=current_node->next;
                    q2->next=q3;
                    q3->prev=q2;
                }
                current_node->next=NULL;
                current_node->prev=NULL;
                free(current_node);
            }
            else q=q->next;
        }
    }
    my_head->cnt--;
}
```

Функция очистки памяти после завершения обработки двусвязного линейного списка (функция clear_list2()) полностью идентична соответствующей

функции для односвязного линейного списка, поэтому нет необходимости приводить ее код.

В двусвязном кольцевом списке в поле next последнего элемента списка хранится адрес первого элемента, а в поле prev первого элемента – адрес последнего элемента списка. Соответственно, требуется модификация функций добавления и удаления элементов списка, а в функции main() при выводе элементов списка можно использовать циклы for() на основе значения поля cnt в «голове» списка.

Модифицированная функция main() показана далее. Описания структур и прототипов функций, а также вызовы функций не изменяются по сравнению со случаем линейного двусвязного списка.

```
int main()
{
    head *ph=NULL;
    node *p=NULL;
    char my_word[MAXLEN];
    int slen,i,n;

    ph=make_head2();
    puts("--Press \"\\q\" for terminate input--\n");
    while(strncmp(my_word,"\\q",2)!=0)
    {
        printf("%s: ","Please enter your word");
        fgets(my_word,MAXLEN,stdin);
        slen=strlen(my_word);
        my_word[slen-1]='\0';
        if(strncmp(my_word,"\\q",2)!=0)
        {
            p=make_node2(my_word,slen);
            add_node2(ph,p);
            printf("%d %s\n",p->id, p->word);
        }
    }
    puts("\n--Input terminated. Your data are:--");
    p=ph->first;
    for(i=0;i<ph->cnt;i++)
    {
        printf("Address: %p, data: %d %s\n",p, p->id, p->word);
        p=p->next;
    }
    printf("Head fields: %d %d %p %p\n",
              ph→last_id,ph→cnt, ph->first,ph->last);
    printf("%s: ","\nPlease enter your number");
    scanf("%d", &n);
    p=select_id2(ph,n);
    if(p!=NULL)
    {
        printf("Address: %p, data: %d %s\n",p, p->id, p->word);
        delete_node2(ph,p);
        puts("--Data after deleting selected node --");
        p=ph->first;
        for(i=0;i<ph->cnt;i++)
        {
```

```
            printf("Address: %p, data: %d %s\n",
                  p, p->id, p->word);
            p=p->next;
        }
        printf("Head fields: %d %d %p %p\n",
                  ph->last_id,ph->cnt,ph->first,ph->last);
    }
    else printf("ERROR! Element not exists!\n");
    clear_list2(ph);
    return 0;
}
```

Для реализации циклов с параметром в функции `main()` введена дополнительная переменная.

При добавлении элемента списка следует обработать ситуацию, когда список пустой и добавляемый элемент одновременно является первым и последним. Если список не пустой, то новый элемент добавляется после последнего, что требует корректного переназначения указателей – новый элемент становится предыдущим для первого элемента списка, а первый элемент становится следующим для добавляемого элемента. Код функции `add_node2()` для двусвязного кольцевого списка показан далее.

```
void add_node2(head *my_head, node *new_node)
{
    node *q=NULL,*q1=NULL;
    int k;
    if(my_head&&new_node)
    {
        q=my_head->first;
        q1=my_head->last;
        if(q==NULL) // список пустой
        {
            my_head->last_id=1;
            my_head->cnt=1;
            my_head->first=new_node;
            my_head->last=new_node;
            new_node->next=my_head->first;
            new_node->prev=my_head->last;
        }
        else // в списке есть эелементы
        {
            k=my_head->last_id+1;
            my_head->cnt++;
            new_node->id=k;
            q1->next=new_node;
            new_node->prev=q1;
            new_node->next=q;
            my_head->last=new_node;
            my_head->last_id=k;
        }
    }
}
```

При удалении элемента списка следует обратить внимание на указатели на следующий и предыдущий элементы, хранящиеся в полях соответственно по-

следнего и первого элемента списка. Код функции delete_node2() для удаления элемента двусвязного кольцевого списка приведен далее.

```
void delete_node2(head *my_head, node *current_node)
{
    node *q,*q1,*q2,*q3;
    int i;

    q=my_head->first;
    q1=my_head->last;
    if(current_node==q)
    {
        q2=current_node->next;
        q1->next=q2;
        my_head->first=q2;
        q2->prev=my_head->last;
        current_node->next=NULL;
        current_node->prev=NULL;
        free(current_node);
        my_head->cnt--;
    }
    else
    {
        for(i=0;i<my_head->cnt;i++)
        {
            if(q->next==current_node)
            {
                q2=current_node->prev;
                if(current_node==q1)
                {
                    q3=my_head->first;
                    my_head->last=q2;
                    q2->next=q3;
                    q3->prev=q2;
                }
                else
                {
                    q3=current_node->next;
                    q2->next=q3;
                    q3->prev=q2;
                }
                current_node->next=NULL;
                current_node->prev=NULL;
                free(current_node);
                my_head->cnt--;
            }
            else q=q->next;
        }
    }
}
```

Если удаляется первый элемент списка, то адрес элемента, следующего за удаляемым, сохраняется в переменной q2, а затем производится переназначение указателей. Элемент с адресом q2 становится следующим за последним элементом списка, затем он объявляется первым элементом списка, а предыдущим элементом для элемента с адресом q2 становится последний элемент списка. Здесь же значение поля cnt в «голове» списка уменьшается на 1.

При удалении остальных элементов проход по списку выполняется с помощью цикла `for()` в соответствии со значением поля `cnt` в «голове» списка.

Если удаляется последний элемент списка, то адрес элемента, следующего за удаляемым сохраняется в переменной `q2`, адрес первого элемента списка сохраняется в переменную `q3`, а затем производится переназначение указателей. Элемент с адресом `q2` объявляется последним элементом, для которого следующим является первый элемент списка, а для первого элемента списка предыдущим делается новый последний элемент.

Если удаляется элемент из середины списка, то требуется запомнить адреса предыдущего и следующего элемента, после чего переназначить указатели. Затем следует уменьшить на 1 значение поля `cnt` в «голове» списка, чтобы не выполнять лишних итераций.

Функция `clear_list2()` для очистки памяти, выделенной для элементов двусвязного кольцевого списка полностью идентична соответствующей функции `clear_list()` для односвязного кольцевого списка.

Задания по теме.

Написать программу, в которой создается двусвязный список и реализуются две функции из указанного ниже перечня:

1. Функция `add_firstnode2()`, добавляющая элементы двусвязного линейного списка всегда перед первым элементом.

2. Функция поиска элементов двусвязного линейного списка по подстроке текстового поля и вывода информационных полей найденных элементов списка.

3. Функция удаления элемента двусвязного списка по заданному значению `id`.

4. Функция вывода двусвязного линейного списка в обратном порядке.

5. Функция определения количества элементов двусвязного линейного списка (длины списка), если «голова» является указателем на первый элемент списка, а не структурой.

6. Функция перестановки элементов двусвязного списка (`swap2()`) с использованием промежуточной структуры.

7. Функция вывода значений полей элементов двусвязного кольцевого списка, если «голова» списка является указателем на первый элемент, а не структурой.

8. Функция поиска элемента двусвязного кольцевого списка по заданному значению `id`, если «голова» списка является указателем на первый элемент, а не структурой.

9. Функция удаления элемента двусвязного кольцевого списка с известным адресом, если «голова» списка является указателем на первый элемент, а не структурой.

10. Функция очистки памяти для двусвязного кольцевого списка, если «голова» списка является указателем на первый элемент, а не структурой.

СТЕКИ И ОЧЕРЕДИ

Стек – динамическая структура данных (динамическая информационная структура) с методом доступа к элементам LIFO (от англ. Last In – First Out, «последним пришёл – первым вышел»). Стек представляет собой совокупность линейно-связанных однородных элементов (список), для которых разрешено добавлять или удалять элементы только с одного конца списка, который называется вершиной (головой) стека.

Вершина стека – единственный элемент, с которым можно работать.

При записи и выборке элементов изменяется только адрес вершины стека. Поэтому каждый стек имеет базовый адрес (основание), от которого производятся все операции со стеком. В случае, когда стек пуст, адреса вершины и основания стека совпадают.

Допустимыми операциями со (над) стеком являются:

- проверка стека на пустоту;
- добавление нового элемента;
- изъятие элемента с вершины стека;
- доступ к элементу на вершине, если стек не пуст;
- очитка стека.

Далее рассмотрим реализацию стека на односвязном линейном списке. Определение структуры и пользовательского типа для элемента стека приведено далее.

```
struct elem
{
    int id;
    char *data;
    struct elem *next;
};
typedef struct elem item;
```

Если для списка сформировать «голову» в виде структуры с указателями на первый и на последний элементы списка, то можно использовать уже рассмотренные ранее функции работы с односвязным списком с минимальными изменениями.

Более интересным является вариант, когда стек строится как односвязный список из элементов типа item, но у этого списка нет «головы» как отдельной структуры, имеется только текущий адрес вершины – top.

Примем следующие названия основных функций работы со стеком:

- функция push() добавляет новый элемент всегда перед первым;
- функция pop() выводит и удаляет всегда первый элемент;
- функция peek() выводит, но не удаляет первый элемент.

При добавлении или удалении элементов стека адрес вершины меняется внутри соответствующей функции, поэтому этот адрес передается как параметр по ссылке (через указатель) в функции push() и pop(), при этом любой элемент стека является указателем на структуру.

Соответственно, реализация функции push() может выглядеть как приведенный ниже код.

```
void push(item **top, char *word, int len, int n)
{
    item *tmp=NULL;
    char *someword=NULL;

    tmp=(item*)malloc(sizeof(item));
    someword=(char*)malloc((len+1)*sizeof(char));

    if(tmp&&someword)
    {
        tmp->next=*top;
        strcpy(someword,word);
        tmp->data=someword;
        tmp->id=n;
        *top=tmp;
    }
}
```

При добавлении элемента в стек в первую очередь выделяется память для элемента стека как структуры, а также для информационных полей структуры. Затем для вновь созданного элемента в качестве адреса следующего элемента устанавливается адрес текущей вершины. После этого заполняются информационные поля элемента стека и адрес текущего элемента объявляется новой вершиной стека.

При извлечении элемента из стека (функция pop()) адрес текущей вершины стека следует запомнить в служебную переменную. Затем вершиной назначается следующий элемент. Функция возвращает адрес элемента, который был вершиной до применения этой функции. Затем со структурой, размещенной в памяти по этому адресу, можно делать любые доступные операции. После работы с полученной переменной память, выделенную для нее, нужно очистить

обычным способом. В результате применения функции pop() количество элементов в стеке уменьшается на 1. Код функции pop() приведен далее.

```
item *pop(item **top)
{
    item *tmp=NULL;

    if(top)
    {
        tmp=*top;
        *top=(*top)->next;
    }
    return tmp;
}
```

Функция просмотра элемента на вершине стека (функция peek()) возвращает адрес вершины, и можно вывести значения информационных полей структуры, размещенной в памяти по этому адресу. Код функции peek() приведен далее. Применение этой функции не меняет состояние стека.

```
item *peek(item *top)
{
    return top;
}
```

Функция для определения количества элементов в стеке («глубины стека») может быть реализована следующим образом. В промежуточную переменную-указатель сохраняется адрес вершины стека, а затем выполняются переходы по адресам, хранящимся в поле next каждого элемента, пока в этом поле не окажется адрес NULL («основание» или «дно» стека). При каждом переходе счетчик элементов увеличивается на 1. Код функции (пусть она называется elem_count()) приведен далее.

```
int elem_count(item *top)
{
    item *tmp;
    int i;
    i=0;
    if(top)
    {
        tmp=top;
        while(tmp)
        {
            tmp=tmp->next;
            i++;
        }
    }
    return i;
}
```

Для очистки стека следует выполнять операцию pop() и освобождение памяти для каждого элемента, пока не будет достигнуто основание стека.

Очередь – динамическая структура данных (динамическая информационная структура) с методом доступа к элементам FIFO (от англ. First In – First Out, «первым пришёл – первым вышел»). Очередь представляет собой совокупность

линейно-связанных однородных элементов (список), для которых разрешено добавлять элементы в конец списка, а извлекать их из его начала.

С очередью возможны следующие типовые операции:

- добавление элемента;
- удаление элемента;
- чтение первого элемента;
- определение длины очереди;
- очистка очереди.

Рассмотрим реализацию очереди на двусвязном линейном списке. Если «голова» списка определена как структура, хранящая адреса первого и последнего элементов списка, то можно использовать уже рассмотренные ранее функции работы с двусвязным списком. Однако следует учесть, что элементы должны добавляться в начало списка (перед первым).

Пусть структура и пользовательский тип для элемента очереди определены, как показано далее.

```
struct elem
{
    int id;
    char *data;
    struct elem *prev;
    struct elem *next;
};

typedef struct elem item;
```

Пусть очередь строится из элементов типа `item` как двусвязный список, но у этого списка нет «головы» как специальной структуры, имеются только текущие адреса начала (`qbegin`) и конца (`qend`).

При добавлении или удалении элементов очереди эти адреса будут меняться внутри соответствующих функций, поэтому они передаются как параметры по ссылке (через указатели). При этом сами элементы очереди являются указателями на структуры.

При добавлении элемента в очередь (функция `put()`) следует выделить память для элемента очереди как структуры и для информационных полей элемента очереди. Затем следует заполнить информационные поля элемента очереди. Если очередь пустая (`qbegin` имеет значение NULL), то следует установить начала и конца очереди равными адресу созданного элемента, а значения полей next и prev для элементов, размещенных по адресам начала и конца очереди, следует установить в NULL.

Если очередь не пустая, то новый элемент объявляется концом очереди. При этом, если в очереди был один элемент, то новый элемент становится предыдущим для элемента в начале очереди.

Код функция put() приведен далее.

```
void put(item **qbegin, item **qend, char *word,int len, int n)
{
    item *tmp=NULL;
    char *someword=NULL;

    tmp=(item*)malloc(sizeof(item));
    someword=(char*)malloc((len+1)*sizeof(char));
    if(tmp&&someword)
    {
        strcpy(someword,word);
        tmp->data=someword;
        tmp->id=n;
        if(!(*qbegin)) // Пустая очередь
        {
            *qbegin=tmp;
            (*qbegin)->next=NULL;
            (*qbegin)->prev=NULL;
            *qend=tmp;
            (*qend)->next=NULL;
            (*qend)->prev=NULL;
        }
        Else // В очереди есть элементы
        {
            tmp->next=*qend;
            (*qend)->prev=tmp;
            *qend=tmp;
            if(!((*qbegin)->prev)) (*qbegin)->prev=tmp;
        }
    }
}
```

Функция получения элемента из начала очереди (функция get()) может быть реализована следующим образом. Адрес текущего начала очереди запоминается в промежуточную переменную tmp, затем началом очереди объявляется элемент, предыдущий для элемента с адресом tmp. Если адрес начала очереди не NULL (еще есть элементы в очереди), то указывается, что элемент в начале очереди – последний. Если элементов в очереди больше нет, то адрес конца очереди следует установить в NULL. Функция get() возвращает адрес элемента, который был началом очереди до применения этой функции. Затем со структурой, размещенной в памяти по полученному адресу, можно делать любые доступные операции. После работы с полученной переменной память, выделенную для нее, нужно очистить обычным способом. При выполнении функции get() количество элементов в очереди уменьшается на 1.

Код функции get() приведен далее.

```
item *get(item **qbegin, item **qend)
{
    item *tmp=NULL;

    if(*qbegin)
    {
        tmp=*qbegin;
        *qbegin=(*qbegin)->prev;
```

```
        if(*qbegin)
        {
            if((*qbegin)->next) (*qbegin)->next=NULL;
        }
        else *qend=NULL;
    }
    return tmp;
}
```

Функция просмотра элемента в начале очереди (функция `view()`) возвращает адрес начал очереди и полностью аналогична функции `peek()` для стека.

Функция для определения количества элементов в очереди («длины очереди») может быть реализована следующим образом. В промежуточную переменную-указатель сохраняется адрес конца очереди, а затем выполняются переходы по адресам, хранящимся в поле next каждого элемента, пока в этом поле не окажется адрес NULL (последний элемент списка – начало очереди). При каждом переходе счетчик элементов увеличивается на 1. Код функции (пусть она называется `elem_count()`) полностью аналогичен коду функции определения глубины стека.

Для очистки очереди последовательно выполняется извлечение элементов из начала очереди (функция `get()`) и освобождение памяти для каждого элемента, пока очередь не станет пустой.

Задания по теме.

Написать программу, в которой создается стек или очередь и реализуется для них две функции из указанного ниже перечня:

1. Функция `clear_stack()` для очистки стека, реализованного на односвязном списке.

2. Функция `clear_ queue()` для очистки очереди, реализованной на двусвязном списке.

3. Функция, выводящая в поток вывода только те элементы стека, в которых текстовое поле содержит хотя бы один заданный символ.

4. Функция, выводящая в поток вывода только те элементы очереди, в которых текстовое поле не содержит заданного символа.

5. Функция, записывающая в поток вывода элементы стека, имеющие четные позиции (вершина стека имеет позицию 0).

6. Функция, записывающая в поток вывода элементы очереди, имеющие нечетные позиции (начало очереди имеет позицию 0).

7. Функция, выводящая в поток вывода элементы стека, имеющие нечетные значения.

8. Функция, определяющая количество четных значений элементов очереди.

9. Функция, сортирующая стек по возрастанию значений чисел.

10. Функция, сортирующая очередь по убыванию значений чисел.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Подбельский В. В., Фомин С. С. Курс программирования на языке Си. 2-е изд. М.: ДМК-Пресс, 2018.

2. Столяров А. В. Программирование: Введение в профессию II. Низкоуровневое программирование. М.: МАКС Пресс, 2016.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ 3
СТРУКТУРЫ, МАССИВЫ СТРУКТУР 4
УКАЗАТЕЛИ НА СТРУКТУРЫ 13
ОДНОСВЯЗНЫЕ (ОДНОНАПРАВЛЕННЫЕ) СПИСКИ 19
ДВУСВЯЗНЫЕ (ДВУНАПРАВЛЕННЫЕ) СПИСКИ 31
СТЕКИ И ОЧЕРЕДИ 41
СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 47

Разумовский Геннадий Васильевич
Хахаев Иван Анатольевич

Обработка структур и списков на языке Си
Учебно-методическое пособие

Редактор

Подписано в печать Формат 60 × 84 1/16
Бумага офсетная Печать цифровая Печ. л. 3,0
Гарнитура «Times New Roman» Тираж 94 экз. Заказ

Издательство СПбГЭТУ ЛЭТИ
197376, С.-Петербург, ул. проф. Попова, 5