Федеральное агентство по образованию

Государственное образовательное учреждение высшего профессионального образования

Ульяновский государственный технический университет

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С

Методические указания к лабораторным работам
(второй семестр)

Составитель С.М. Наместников

Ульяновск
2016

УДК 621.394.343 (076)
ББК 32.88 я7
П33

Рецензент старший преподаватель кафедры «Радиотехника» Ульяновского государственного технического университета, канд. техн. наук, Смирнов П. В.

Одобрено секцией методических пособий научно-методического совета университета

Программирование на языке С: методические указания к лабораторным П33 работам /сост. С. М. Наместников. – Ульяновск : УлГТУ, 2016. – 44 с.

Указания по курсу «Информатика» для студентов направления 11.03.02 Инфокоммуникационные технологии и системы связи, профиль подготовки "Сети связи и системы коммутации" разработаны в соответствии с программой курса «Информатика» и предназначен для студентов специальности «Сети связи и системы коммутации», но может использоваться и студентами других специальностей. Лабораторные работы посвящены основам программирования на языке С.

Сборник подготовлен на кафедре «Телекоммуникации».

УДК 621.394.343 (076)
ББК 32.88 я7

СОДЕРЖАНИЕ

Лабораторная работа №1

УКАЗАТЕЛИ

Цель работы: изучить особенности и порядок работы с указателями.

Теоретические сведения

При объявлении переменных, структур, объединений и т.п. операционная система выделяет необходимый объем памяти для хранения данных программы. Например, задавая целочисленную переменную

```
int a = 10;
```

в памяти ЭВМ выделяется либо 2, либо 4 байта (в зависимости от стандарта языка С), которые расположены друг за другом, начиная с определенного адреса. Здесь под адресом следует понимать номер байта в памяти, который показывает, где начинается область хранения той или иной переменной или каких-либо произвольных данных. Условно память ЭВМ можно представить в виде последовательности байт (рис. 1).

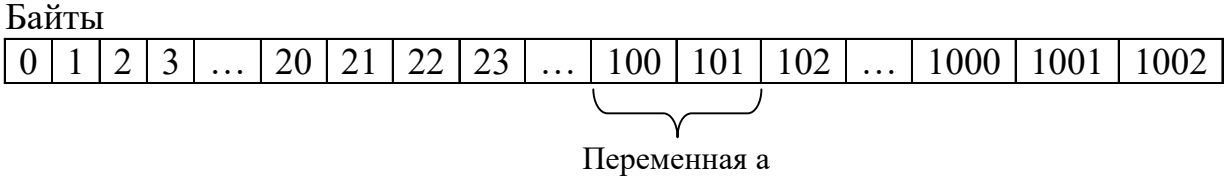

Рис. 1. Условное представление памяти ЭВМ с расположением переменной a

На рис. 1 переменная a расположена в 100 и 101 ячейках и занимает соответственно два байта. Адрес этой переменной равен 100. Учитывая, что значение переменной a равно 10, то в ячейке под номером 100 будет записано число 10, а в ячейке под номером 101 – ноль. Аналогичная картина остается справедливой и при объявлении произвольных переменных и структур, только в этом случае расходуется разный объем памяти в зависимости от типа переменной.

В языке С имеется механизм работы с переменными через их адрес. Для этого необходимо объявить указатель соответствующего типа. Указатель объявляется также как и переменная, но перед его именем ставится символ ‘*’:

```
int *ptr_a;
char *ptr_ch, *ptr_var;
```

Для того чтобы с помощью указателя ptr_a работать с переменной a он должен указывать на адрес этой переменной. Это значит, что значение указателя ptr_a должно быть равно адресу переменной a. Здесь возникает две задачи: во-первых, необходимо определить адрес переменной, и, во-вторых,

присвоить этот адрес указателю. Для определения адреса в языке C++ используется символ ‘&’ как показано ниже:

```
ptr_a = &a;      //инициализация указателя
```

По существу получается, что указатель это переменная, которая хранит адрес на заданную область памяти. Но в отличие от обычной переменной позволяет еще, и работать с данной областью, т.е. записывать в нее значения и считывать их. Допустим, что переменная a содержит число 10, а указатель ptr_a указывает на эту переменную. Тогда для того чтобы считывать и записывать значения переменной a с помощью указателя ptr_a используется следующая конструкция языка C:

```
int b = *ptr_a; //считывание значения переменной a
*ptr_a = 20;    //запись числа 20 в переменную a
```

Здесь переменной b присваивается значение переменной a через указатель ptr_a, a, затем, переменной a присваивается значение 20. Таким образом, для записи и считывания значений с помощью указателя необходимо перед его именем ставить символ ‘*’ и использовать оператор присваивания.

Каждый раз при работе с указателями необходимо выполнять их инициализацию, т.е. задавать адрес на выделенную область памяти. Сложность работы с указателями заключается в том, что при их объявлении они указывают на произвольную область памяти, с которой можно работать как с обычной переменной. Приведем такой пример

```
int* ptr;
*ptr = 10;
```

В результате в произвольную область памяти будет записано два байта со значениями 10 и 0. Это может привести к необратимым последствиям в работе программы и к ее ошибочному завершению. Поэтому перед использованием указателей всегда нужно быть уверенным, что они предварительно были инициализированы.

Рассмотрим возможность использования указателей при работе с массивами. Допустим, что объявлен массив целочисленного типа int размерностью в 20 элементов:

```
int array[20];
```

Элементы массивов всегда располагаются друг за другом в памяти ЭВМ, начиная с первого, индекс которого равен 0 (рис. 2).

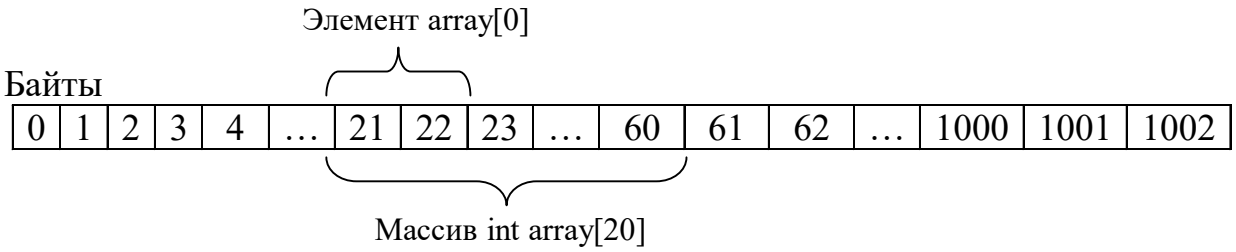

Рис. 2. Условное расположение массива int array[20] в памяти ЭВМ

Из рис. 2 видно, что для получения адреса массива array достаточно знать адрес его первого элемента array[0], который можно определить как адрес переменной и присвоить его указателю:

```
int* ptr_ar = &array[0];
```

Однако в языке С предусмотрена более простая конструкция для определения адреса массивов, которая записывается следующим образом:

```
int* ptr_ar = array;
```

т.е. имя массива задает его адрес. Следует отметить, что величины &array[0] и array являются константами, т.е. не могут менять своего значения. Это означает, что массив (как и переменная) не меняют своего адреса, пока существуют в зоне своей видимости.

Формально, имея указатель на массив, можно считывать и записывать в него значения элементов. Вначале, когда указатель указывает на первый элемент, его значение можно менять следующим образом:

```
int a = *ptr_ar;
*ptr_ar = 20;
```

Для того чтобы перейти к следующему элементу массива, достаточно выполнить операцию

```
ptr_ar += 1;
```

или

```
ptr_ar++;
```

Особенностью применения данной операции является то, что адрес, т.е. значение указателя ptr_ar изменится не на единицу, а на четыре, ровно на столько, сколько занимает один элемент в памяти ЭВМ, в данном случае четыре байта. В результате указатель ptr_ar будет указывать на следующий элемент массива, с которым можно работать также как и с предыдущим.

В языке С при работе с массивами через указатель допускается более простая форма чем рассмотренная ранее. Допустим, что ptr_ar указывает на

первый элемент массива array. Тогда для работы с элементами массива можно пользоваться следующей записью:

```
int *ptr_ar = array;
ptr_ar[0] = 10;
ptr_ar[1] = 20;
```

т.е. доступ к элементам массива осуществляется по его индексу.

Массивы удобно передавать функциям через указатели. Пусть имеется функция вычисления суммы элементов массива:

```
int sum(int* ar, int n);
```

и массив элементов

```
int array[5] = {1,2,3,4,5};
```

Тогда для передачи массива функции sum следует использовать такую запись:

```
int s = sum(array,5);
```

т.е. указатель ar инициализируется по имени массива array и будет указывать на его первый элемент.

Следует отметить, что все возможные изменения, выполненные с массивом внутри функции sum(), сохраняются в массиве array. Это свойство можно использовать для модификации элементов массива внутри функций. Например, рассмотренная ранее функция strcpy(char *dest, char* src), выполняет изменение массива, на который указывает указатель dest. Для того чтобы «защитить» массив от изменений следует использовать ключевое слово const либо при объявлении массива, либо в объявлении аргумента функции как показано ниже.

```
char* strcpy(char* dest, const char* src)
{
 while(*src != '\0') *dest++ = *src++;
 *dest = *src;
 return dest;
}
```

В этом объявлении указатель src не может вносить изменения в переданный массив при вызове данной функции, а может лишь передавать значения указателю dest. В приведенном примере следует обратить внимание на использование конструкции *dest++ и *src++. Дело в том, что приоритет операции ++ выше приоритета операции *, поэтому эти выражения аналогичны выражениям *(dest++) и *(src++). Таким образом, в строке

```
*dest++ = *src++;
```

сначала выполняется присваивание значений соответствующих элементов, на которые указывают dest и src, а затем происходит увеличение значений указателей для перехода к следующим элементам массивов. Благодаря такому подходу осуществляется копирование элементов одного массива в другой. Последняя строка примера *dest = *src, присваивает символ конца строки ‘\0’ массиву dest.

В общем случае можно выполнять следующие операции над указателями:

pt1 = pt2; //Присвоение значения одного указателя другому
pt1 += *pt2; //Увеличение значения первого указателя на величину *pt2
pt1 -= *pt2; //Уменьшение адреса указателя на величину *pt2
pt1-pt2; //Вычитание значений адресов первого и второго указателей
pt1++; и ++pt1; //Увеличение адреса на единицу информации
pt1--; и --pt1; //Уменьшение адреса на единицу информации

Если указатели pt1 и pt2 имеют разные типы, то операция присваивания должна осуществляться с приведением типов, например:

```
int* pt1;
double* pt2;
pt1 = (int *)pt2;
```

Язык С допускает инициализацию указателя строкой, т.е. будет верна следующая запись:

```
char* str = “Лекция”;
```

Здесь задается массив символов, содержащих строку «Лекция» и адрес этого массива передается указателю str. Таким образом, получается, что есть массив, но нет его имени. Есть только указатель на его адрес. Подобный подход является удобным, когда необходимо задать массив строк. В этом случае возможна такая запись:

```
char* text[] = {«Язык С++ имеет»,
                «удобный механизм»,
                «для работы с памятью.»};
```

При таком подходе задается массив указателей, каждый из которых указывает на начало соответствующей строки. Например, значение *text[0] будет равно символу ‘Я’, значение *text[1] – символу ‘у’ и значение *text[2] – символу ‘д’. Особенность такого подхода состоит в том, что здесь задаются три отдельных одномерных массива символов никак не связанных друг с другом. Каждый массив – это отдельная строка. В результате не расходуется лишний объем памяти характерный для двумерного массива символов

Язык С также позволяет инициализировать указатели на структуры. Допусти, что имеется структура

```
struct tag_person {
```

```
 char name[100];
 int old;
} person;
```

и инициализируется следующий указатель:

```
struct tag_person* pt_man = &person;
```

В этом случае, для доступа к элементам структуры можно использовать следующий синтаксис:

```
(*pt_man).name;
pt_man->name;
```

Последний вариант показывает особенность использования указателя на структуры, в котором для доступа к элементу используется операция ->. Данная операция наиболее распространена по сравнению с первым вариантом и является предпочтительной.

Каждый раз при инициализации указателя использовался адрес той или иной переменной. Это было связано с тем, что компилятор языка С автоматически выделяет память для хранения переменных и с помощью указателя можно без последствий работать с этой выделенной областью. Вместе с тем существуют функции malloc() и free(), позволяющие выделять и освобождать память по мере необходимости. Данные функции находятся в библиотеке <stdlib.h> и имеют следующий синтаксис:

```
void* malloc(size_t);          //функция выделения памяти
void free(void* memblock);     //функция освобождения памяти
```

Здесь size_t – размер выделяемой области памяти в байтах; void* - обобщенный тип указателя, т.е. не привязанный к какому-либо конкретному типу. Рассмотрим работу данных функций на примере выделения памяти для 10 элементов типа double.

Листинг 6. Программирование динамического массива.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
 double* ptd;
 ptd = (double *)malloc(10 * sizeof(double));
 if(ptd != NULL)
 {
      for(int i = 0;i < 10;i++)
           ptd[i] = i;
 } else printf("Не удалось выделить память.");
 free(ptd);
```

```
 return 0;
}
```

При вызове функции malloc() выполняется расчет необходимой области памяти для хранения 10 элементов типа double. Для этого используется функция sizeof(), которая возвращает число байт, необходимых для хранения одного элемента типа double. Затем ее значение умножается на 10 и в результате получается объем для 10 элементов типа double. В случаях, когда по каким-либо причинам не удается выделить указанный объем памяти, функция malloc() возвращает значение NULL. Данная константа определена в нескольких библиотеках, в том числе в <stdio.h> и <stdlib.h>. Если функция malloc() возвратила указатель на выделенную область памяти, т.е. не равный NULL, то выполняется цикл, где записываются значения для каждого элемента. При выходе из программы вызывается функция free(), которая освобождает ранее выделенную память. Формально, программа написанная на языке С при завершении сама автоматически освобождает всю ранее выделенную память и функция free(), в данном случае, может быть опущена. Однако при составлении более сложных программ часто приходится много раз выделять и освобождать память. В этом случае функция free() играет большую роль, т.к. не освобожденная память не может быть повторно использована, что в результате приведет к неоправданным затратам ресурсов ЭВМ.

Задание на лабораторную работу

1. Написать программу работы с указателями в соответствии с номером своего варианта.

Варианты заданий

Вариант	Задания программирования указателей
1	Написать функцию, принимающую указатель на строку и выполняющую удаление всех букв а из строки
2	Написать программу сортировки динамического целочисленного массива с помощью указателей
3	Написать функцию разбиения строки на слова (слова разделяются пробелом) и возвращающую их
4	Написать функцию, осуществляющую обмен двух строк между собой, которые передаются ей через указатели
5	Написать программу изменения порядка следования элементов динамического массива в обратном порядке

6	Написать функцию сравнения двух строк, используя указатели на них
7	Написать функцию выделения слов из переданной ей строки, содержащие заданный символ, и возвращающую их
8	Написать программу обмена данными между двумя динамическими массивами
9	Написать функцию сортировки массива структур с информацией по книгам по возрастанию года издания и возвращающую отсортированный массив
10	Написать функцию поиска книги с указанным автором в переданном ей массиве структур с информацией по книгам и возвращающую найденные книги

Содержание отчета

1. Титульный лист с названием лабораторной работы, номером варианта, фамилией студента и группы.
2. Текст программы.
3. Результаты действия программы.
4. Выводы о полученных результатах работы программы.

Контрольные вопросы

1. Для чего предназначены и как задаются указатели в языке С?
2. Что такое адрес переменной?
3. Объявите целочисленную переменную и проинициализируйте на нее указатель.
4. Чему будет равно значение указателя int* ptr = 0; после выполнения операции ptr++?
5. Каким образом можно задавать указатель на массив?
6. Для чего предназначена функция malloc()?
7. Запишите программу копирования одной строки в другую с помощью указателей на эти строки.
8. Что делает функция free() и в какой библиотеке она определена?
9. Какие операции с указателями допустимы?

Лабораторная работа №2

СТЕК

Цель работы: изучить теорию и научиться программировать стек.

Теоретические сведения

Применение указателей позволяет создавать различные динамические структуры для хранения данных. Наиболее простой из них является стек, представляющий собой последовательность объектов данных, связанных между собой с помощью указателей. Особенность организации структуры стека показана на рис. 3.

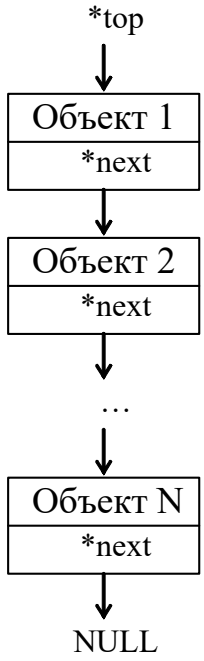

Рис. 3. Организация структуры стека

Из рис. 3 видно, что каждый объект стека связан с последующим с помощью указателя next. Если указатель next равен NULL, то достигнут конец стека. Особенность работы со стеком заключается в том, что новый объект всегда добавляется в начало списка. Удаление также возможно только для первого элемента. Таким образом, данная структура реализует очередь по принципу «первым вошел, последним вышел». Такой принцип характерен при вызове функций в рекурсии, когда адрес каждой последующей функции записывается в стеке для корректной передачи управления при ее завершении.

Для описания объектов составляющих стек можно воспользоваться структурами, например, следующее определение задает шаблон для описания объекта данных стека:

```
typedef struct tag_obj {
 char name[100];
  struct tag_obj* next;
} OBJ;
```

Здесь поле name – символическое имя объекта, а next – указатель на следующий объект. Зададим указатель top как глобальную переменную со значением равным NULL:

```
OBJ* top = NULL;
```

и введем функцию для добавления нового объекта в стек:

```
void push(char* name)
{
 OBJ* ptr = (OBJ *)malloc(sizeof(OBJ));
```

```
 strcpy(ptr->name,name);
 ptr->next = top;
 top = ptr;
}
```

Данная функция в качестве аргумента принимает указатель на строку символов, которые составляют имя добавляемого объекта. Внутри функции инициализируется указатель ptr на новый созданный объект. В поле name записывается переданная строка, а указатель next инициализируется на первый объект. Таким образом, добавленный объект ставится на вершину списка.

Для извлечения объекта из стека реализуется следующая функция:

```
void pop()
{
 if(top != NULL)
 {
      OBJ* ptr = top->next;
      printf("%s - deleted\n",top->name);
      free(top);
      top = ptr;
 }
}
```

В данной функции сначала выполняется проверка указателя top. Если он не равен значению NULL, то в стеке имеются объекты и самый верхний из них следует удалить. Перед удалением инициализируется указатель ptr на следующий объект для того, чтобы он был доступен после удаления верхнего. Затем вызывается функция printf(), которая выводит на экран сообщение об имени удаленного объекта. Наконец, вызывается функция free() для удаления самого верхнего объекта, а указатель top инициализируется на следующий объект.

Для анализа работы данных функций введем еще одну вспомогательную функцию, которая будет выводить имена объектов, находящихся в стеке.

```
void show_stack()
{
 OBJ* ptr = top;
 while(ptr != NULL)
 {
      printf("%s\n",ptr->name);
      ptr = ptr->next;
 }
}
```

Работа данной функции реализуется с помощью цикла while, который работает до тех пор, пока указатель ptr не достигнет конца стека, т.е. пока не будет равен значению NULL. Внутри цикла вызывается функция printf() для

вывода имени текущего объекта на экран и указатель ptr перемещается на следующий объект.

Рассмотрим применение данных функций в функции main():

```
int main()
{
 char str[100];
 for(int i = 0;i < 5;i++) {
      sprintf(str,"Object %d",i+1);
      push(str);
 }
 show_stack();
 while(top != NULL) pop();
 return 0;
}
```

Здесь создается стек, состоящий из 5 объектов, с помощью оператора цикла for. Внутри цикла инициализируется переменная str с именем объекта, которая, затем, передается в качестве аргумента функции push(). После вызова функции show_stack() на экране появляются следующие строки:

Object 5
Object 4
Object 3
Object 2
Object 1

Полученные результаты показывают, что последний 5-й добавленный объект находится на вершине стека, а первый – в конце. При вызове функции pop() в цикле while() осуществляется удаление элементов стека из памяти ЭВМ. В результате на экране появляются строки:

Object 5 - deleted
Object 4 - deleted
Object 3 - deleted
Object 2 - deleted
Object 1 – deleted

Таким образом, функция pop() удаляет верхние объекты стека с 5-го по 1-й.

Задание на лабораторную работу

1. Написать программу работы со стеком в соответствии с номером своего варианта.

Варианты заданий

Вариант	Задания на программирование стека
1	Написать программу, реализующую стек с информацией о студентах и отображающую стек в порядке убывания возраста студента
2	Запрограммировать стек-подобную структуру данных, в которой новый объект добавляется и удаляется с конца стека
3	Запрограммировать стек-подобную структуру данных, в которой объект добавляется в начало, а удаляется с конца
4	Запрограммировать стек-подобную структуру данных, в которой объект добавляется в конец, а удаляется с начала
5	Запрограммировать стек-подобную структуру данных, в которой информация о книгах сортируется по возрастанию года издания
6	Запрограммировать стек-подобную структуру данных, в которой информация о книгах сортируется по убыванию стоимости
7	Написать программу, реализующую стек с информацией о сотрудниках и отображающую стек в порядке возрастания возраста сотрудника
8	Написать программу копирования одного стека в другой
9	Написать программу замены одного стека на другой
10	Написать программу обмены первыми половинами двух стеков

Содержание отчета

1. Титульный лист с названием лабораторной работы, номером варианта, фамилией студента и группы.
2. Текст программы.
3. Результаты действия программы.
4. Выводы о полученных результатах работы программы.

Контрольные вопросы

1. Опишите структуру стека.
2. Объясните принцип работы функции вывода на экран элементов стека.
3. Куда добавляется новый объект стека?
4. Как в программе задается объект стека?
5. Каким образом осуществляется связь между объектами стека?
6. Откуда удаляется объект стека?

Лабораторная работа №3

СВЯЗНЫЕ СПИСКИ

Цель работы: изучить теорию и научиться программировать связные списки.

Теоретические сведения

Рассмотренные ранее типы данных и работа с ними позволяют писать программы разной степени сложности. Однако существуют задачи, в которых традиционное представление информации на основе переменных, структур, объединений и т.п. является не эффективным. Классическим примером такого рода может стать обработка табличных данных, у которых есть заданные поля, т.е. набор стандартных типов данных, и записи, представляющие собой конкретное наполнение таблицы. Формально для описания таблицы можно использовать простой или динамический массив структур. Но в этом случае возникает несколько проблем. Во-первых, наперед часто сложно указать приемлемое число записей (размер массива) для хранения информации. Во-вторых, при большом размере массива сложно выполнять добавление и удаление записей, находящихся между другими записями таблицы. И, наконец, любой используемый массив не будет эффективно использовать память ЭВМ, т.к. всегда будут зарезервированы не используемые записи на случай добавления новых. Эти основные проблемы обусловили необходимость создания нового механизма представления данных в памяти ЭВМ, который получил название связные списки.

Идея связных списков состоит в представлении данных в виде объектов, связанных друг с другом указателями (рис. 4).

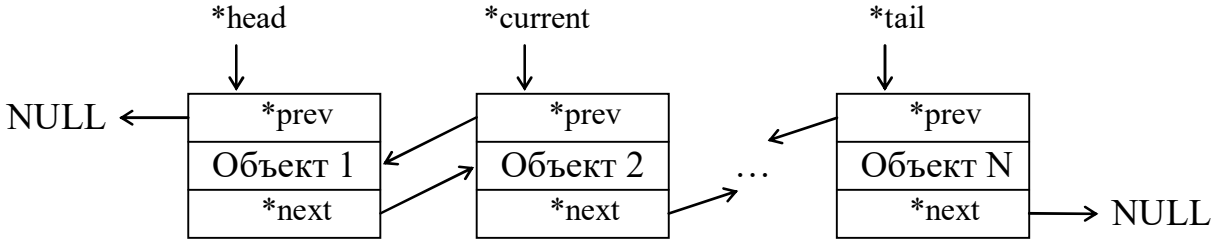

Рис. 4. Графическая интерпретация связных списков

Здесь *prev и *next – указатели на предыдущий и следующий объекты соответственно; *head и *tail – указатели на первый и последний объекты; *current – указатель на текущий объект, с которым идет работа. Если предыдущего или последующего объекта не существует, то указатели *prev и *next принимают значение NULL. Указатели *head и *tail являются вспомогательными и служат для быстрого перемещения к первому и последнему объекту соответственно. Рассмотрим работу связных списков на примере представления следующей информации.

	название	автор	год издания
lib	lib.title	lib.author	lib.year
lib	lib.title	lib.author	lib.year
lib	lib.title	lib.author	lib.year
⋮			
lib	lib.title	lib.author	lib.year

Строки данной таблицы можно описать с помощью структуры:

```
typedef struct tag_lib {
 char title[100];
 char author[100];
 int year;
} LIB;
```

Каждая строка хранится в памяти независимо от местоположения предыдущих и последующих строк, но имеет указатель на предыдущую и последующую строки таблицы. Благодаря этому обеспечивается единство таблицы. Таким образом, необходимо ввести еще одну структуру, которая будет описывать связи между строками таблицы, и представлять собой объект данных:

```
typedef struct tag_obj {
 LIB lib;
 LIB* prev, *next;
} OBJ;
```

Здесь *prev и *next – указатели на предыдущую и следующую строки соответственно.

По умолчанию указатели head и tail равны NULL:

```
OBJ* head = NULL, *tail = NULL;
```

При добавлении записей выполняется инициализация этих указателей, а также prev и next внутри объектов:

```
OBJ* add_obj(char* title, char* author, int year)
{
 OBJ* current = (OBJ *)malloc(sizeof(OBJ));
 strcpy(current->lib.title, title);
 strcpy(current->lib.author, author);
 current->lib.year = year;
 current->prev = tail;
 current->next = NULL;
 if(tail != NULL) tail->next = current;
 if(head == NULL) head = current;
```

```
 tail = current;
 return current;
}
```

Данная функция использует три параметра для ввода данных в структуру LIB. В первой строке функции создается новая структура типа OBJ. Во второй, третьей и четвертой строках осуществляется запись информации в структуру LIB. Затем, инициализируются указатели prev и next добавленного объекта. Учитывая, что добавление осуществляется в конец списка, то указатель next должен быть равен NULL, а указатель prev указывать на предыдущий объект, т.е. быть равен указателю tail. В свою очередь, объект, на который указывает указатель tail, становится предпоследним и его указатель next должен указывать на последний объект, т.е. быть равным указателю current. Затем проверяется, является ли добавляемый объект первым (head == NULL), и если это так, то указатель head приравнивается указателю current. Наконец, указатель tail инициализируется на последний объект. Последняя строка функции возвращает указатель на созданный объект.

Для полноценной работы связного списка необходимо ввести функцию удаления элемента, которая может быть записана следующим образом:

```
OBJ* del_obj(OBJ* current)
{
 if(current == head)
      if(current->prev != NULL) head = current->prev;
      else head = current->next;
 if(current == tail)
      if(current->next != NULL) tail = current->next;
      else tail = current->prev;
 if(current->prev != NULL)
      current->prev->next = current->next;
 if(current->next != NULL)
      current->next->prev = current->prev;
 free(current);
 return head;
}
```

Функция del_obj() в качестве аргумента использует указатель на объект, который следует удалить. Сначала выполняется проверка для инициализации указателя head, в том случае, если удаляется первый объект, на который он указывает. Аналогичная проверка осуществляется для tail. Затем осуществляется проверка: если предыдущий объект относительно текущего существует, то его указатель на следующий объект следует переместить. Аналогичная проверка выполняется и для следующего объекта относительно удаляемого. После настройки всех указателей вызывается функция free() для удаления объекта из памяти и возвращается указатель на первый объект.

Введенные функции в программе можно использовать следующим образом:

```
int main()
{
 OBJ *current = NULL;
 int year;
 char title[100], author[100];
 do
 {
      printf("Введите название книги: ");
      scanf("%s",title);
      printf("Введите автора: ");
      scanf("%s",author);
      printf("Введите год издания: ");
      scanf("%d",&year);
      current = add_obj(title,author,year);
      printf("Для выхода введите 'q'");
 } while(scanf("%d",&year) == 1);
 current = head;
 while(current != NULL)
 {
      printf("Title: %s, author %s, year = %d\n",
current->lib.title, current->author.old, current->lib.year);
      current = current->next;
 }
 while(head != NULL)
      del_obj(head);
 return 0;
}
```

Функция main() осуществляет ввод названия книги, автора и года издания в цикле do while(). Там же вызывается функция add_obj() с соответствующими параметрами, которая формирует связанный список на основе введенных данных. Пользователь выполняет ввод до тех пор, пока не введет какой либо символ на последний запрос. В результате цикл завершится, и указатель current передвигается на первый объект. Затем, в цикле while осуществляется вывод информации текущего объекта на экран, а указатель current передвигается на следующий объект. Данная процедура выполняется до тех пор, пока указатель не станет равным NULL, что означает достижение конца списка. Перед выходом из программы связный список удаляется с помощью функции del_obj(), у которой в качестве аргумента всегда используется указатель head. Если данный указатель принимает значение NULL, то это означает, что связный список пуст и не содержит ни одного объекта. После этого программа завершает свою работу.

Задание на лабораторную работу

1. Написать программу работы со связным списком в соответствии с номером своего варианта.

Варианты заданий

Вариант	Задание для программирования связного списка
1	Запрограммировать связный список, в котором информация о книгах сортируется по возрастанию года издания
2	Запрограммировать связный список, в котором информация о книгах сортируется по убыванию стоимости
3	Запрограммировать связный список, в котором новый объект добавляется и удаляется с конца списка
4	Запрограммировать связный список, в котором объект добавляется в начало, а удаляется с конца списка
5	Запрограммировать связный список, в котором объект добавляется в конец, а удаляется с начала списка
6	Написать программу замены одного связного списка на другой
7	Написать программу копирования одного связного списка в другой
8	Написать программу обмены первыми половинами двух связных списков
9	Написать программу, реализующую связный список с информацией о сотрудниках и отображающую список в порядке возрастания возраста сотрудника
10	Написать программу, реализующую связный список с информацией о студентах и отображающую список в порядке убывания возраста студента

Содержание отчета

1. Титульный лист с названием лабораторной работы, номером варианта, фамилией студента и группы.
2. Текст программы.
3. Результаты действия программы.
4. Выводы о полученных результатах работы программы.

Контрольные вопросы

1. Дайте понятие связного списка.
2. Какие удобства хранения информации представляет связный список по сравнению с массивом.
3. Объясните работу функции удаления элементов связного списка.
4. Как в программе описывается объект связного списка?
5. Объясните работу функции добавления элементов связного списка.
6. Объясните работу функции отображения элементов связного списка.

Лабораторная работа №4

БИНАРНЫЕ ДЕРЕВЬЯ

Цель работы: изучить теорию и научиться программировать бинарные деревья.

Теоретические сведения

Связные списки не охватывают весь спектр возможных представлений данных. Например, с их помощью сложно описать иерархические структуры подобные каталогам и файлам или хранения информации генеалогического древа. Для этого лучше подходит модель известная как бинарные деревья. Графически бинарные деревья можно изобразить как последовательность объектов, каждый из которых может быть связан с двумя последующими (рис. 5).

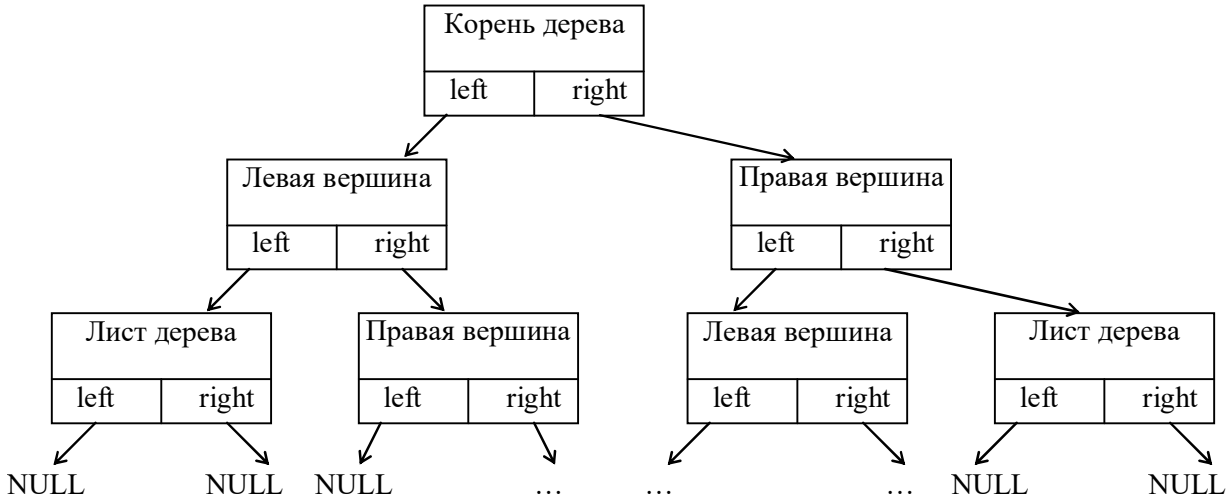

Рис. 5. Графическая интерпретация бинарного дерева

Каждый объект бинарного дерева имеет два указателя: на «левый» и «правый» вершины. Самый верхний уровень называется корнем дерева. Если указатели объекта left и right равны NULL, то он называет листом дерева.

При описании структуры каталогов с помощью бинарного дерева можно воспользоваться следующим правилом. Переход по «левым» указателям будет означать список файлов, а переход по «правым» – список каталогов. Например, для описания простой структуры (рис. 6), бинарное дерево будет иметь вид, представленный на рис. 7.

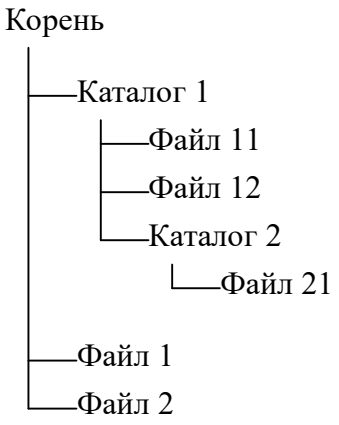

Рис. 6. Структура каталогов

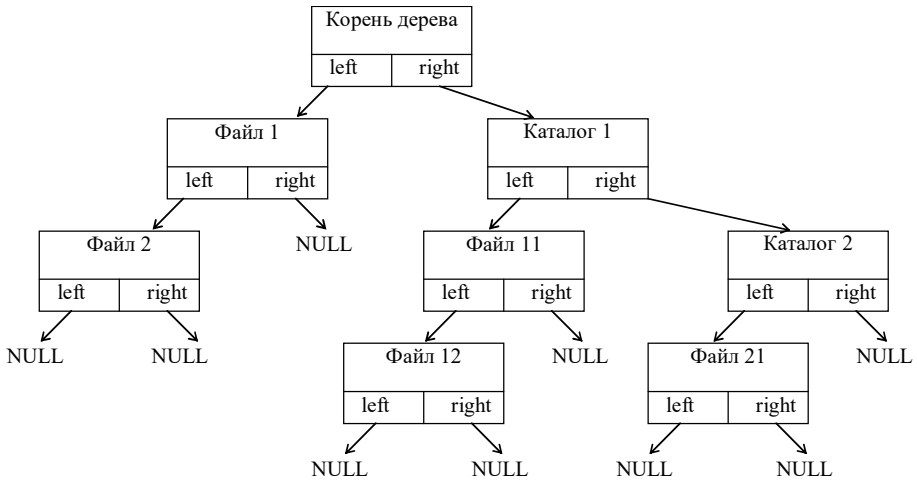

Рис. 7. Структура бинарного дерева

Объекты, из которых состоит бинарное дерево удобно представить в виде структур. Также как и в связных списках, первая структура будет описывать данные, хранящиеся в вершинах дерева, а вторая представлять связи между вершинами.

```
typedef struct tag_data {
 char name[100];
} DATA;

typedef struct tag_tree {
 DATA data;
 struct tag_tree* left, *right;
} TREE;
TREE* root = NULL;
```

Для формирования дерева введем функцию add_node(), которая в качестве аргументов будет принимать указатель на вершину дерева, к которому добавляются новые вершины, имя вершины и тип вершины: левая или правая. Кроме того, данная функция будет возвращать указатель на новую созданную вершину.

```
TREE* add_node(TREE* node,char* name, TYPE type = LEFT)
{
 TREE* new_node = (TREE *)malloc(sizeof(TREE));
 if(type == LEFT && node != NULL) node->left = new_node;
 else if(node != NULL) node->right = new_node;
 strcpy(new_node->data.name,name);
 new_node->left = NULL;
 new_node->right = NULL;
 return new_node;
}
```

Последний аргумент функции имеет тип TYPE, который удобно определить как перечисляемый тип:

```
typedef enum tag_type {RIGHT, LEFT} TYPE;
```

и определить параметр по умолчанию LEFT.

Для отображения дерева целесообразно воспользоваться рекурсивными функциями show_next(), которые вызываются из функции show_tree() следующим образом:

```
void show_next(TREE* node,int off)
{
 if(node != NULL)
 {
      for(int i=0;i < off;i++) putchar(' ');
      printf("%s\n",node->data.name);
      show_next(node->left,off);
      show_next(node->right,off+1);
 }
}
void show_tree()
{
 if(root != NULL)
 {
      printf("%s\n",root->data.name);
      show_next(root->left,0);
      show_next(root->right,1);
 }
}
```

Первый параметр функции show_next() служит для перемещения к следующим вершинам дерева, а второй – для смещения при отображении строк на экране монитора, принадлежащих разным вершинам. Рекурсия функций show_next() выполняется до тех пор, пока указатель на левую или правую вершину не станет равным NULL. В этом случае работа функции завершается, и управление передается предыдущей вызываемой функции. Таким образом, происходит отображение всего бинарного дерева.

При завершении программы необходимо удалить созданные объекты дерева. Для этого также удобно воспользоваться рекурсивными функциями. По аналогии введем рекурсивную функцию del_next(), и основную del_tree(), из которой вызывается функция del_next(). Реализация этих функций приведена ниже:

```
void del_next(TREE* node)
{
 if(node != NULL)
 {
      del_next(node->left);
      del_next(node->right);
      printf("node %s - deleted\n",node->data.name);
      free(node);
```

```
 }
}
void del_tree()
{
 if(root != NULL)
 {
      del_next(root->left);
      del_next(root->right);
      printf("node %s - deleted\n",root->data.name);
      free(root);
 }
}
```

Аргумент функции del_next() используется для перехода к следующим вершинам дерева. В самой функции выполняется проверка: если следующая вершина существует, т.е. указатель не равен NULL, то выполняется просмотр дерева сначала по левой вершине, а затем по правой. При достижении листьев дерева функции del_next() вызываются с аргументом NULL и не выполняют никаких действий, поэтому программа переходит к функции printf(), которая вывод на экран сообщение об имени удаляемой вершины, а затем вызывается функция free() для освобождения памяти, занятой данным объектом. После этого осуществляется переход к предыдущей функции del_next() и описанный процесс повторяется до тех пор, пока не будут удалены все объекты кроме корневого. Корень дерева удаляется непосредственно в функции del_tree(), после чего можно говорить об удалении всего дерева.

Использование описанных функций в функции main() реализуются следующим образом:

```
int main()
{
 root = add_node(NULL,"Root");
 TREE* current = add_node(root,"File 1",LEFT);
 current = add_node(current,"File 2",LEFT);
 current = add_node(root,"Folder 1",RIGHT);
 current = add_node(current,"File 11",LEFT);
 current = add_node(current,"File 12",LEFT);
  current = add_node(root->right,"Folder 2",RIGHT);
 current = add_node(current,"File 21",LEFT);
 show_tree();
 del_tree();
 root = NULL;
 return 0;
}
```

Данная функция сначала инициализирует глобальный указатель root как корень дерева и в качестве первого аргумента функции add_node() записывает NULL, что означает, что предыдущего объекта не существует. Затем вводится вспомогательный указатель current, с помощью которого выполняется создание двоичного дерева, описывающее файловую структуру (рис. 6). После создания

дерева вызывается функция show_tree() для его отображения на экран, затем оно удаляется и программа завершает свою работу.

Задание на лабораторную работу

1. Написать программу работы с бинарным деревом в соответствии с номером своего варианта.

Варианты заданий

Вариант	Задание на программирование бинарного дерева
1	Написать программу копирования одного бинарного дерева в другое
2	Написать программу копирования правых вершин одного бинарного дерева в другое
3	Написать программу копирования левых вершин одного бинарного дерева в другое
4	Написать программу обмена правых вершин одного бинарного дерева на соответствующие левые другого дерева
5	Написать программу замены одного бинарного дерева на другое
6	Написать программу копирования вершин, связанных с левым указателем корня дерева в соответствующие «правые» вершины
7	Написать программу обмена информацией между вершинами, связанными с правым и левым указателями корня дерева
8	Написать программу подсчета числа вершин в бинарном дереве
9	Написать программу подсчета левых вершин бинарного дерева
10	Написать программу подсчета правых вершин бинарного дерева

Содержание отчета

1. Титульный лист с названием лабораторной работы, номером варианта, фамилией студента и группы.
2. Текст программы.
3. Результаты действия программы.
4. Выводы о полученных результатах работы программы.

Контрольные вопросы

1. Опишите структуру бинарного дерева.
2. Какой тип информации удобно представлять с помощью бинарных деревьев?
3. Объясните принцип работы рекуррентных функций для отображения и удаления элементов бинарного дерева.
4. Что является объектом бинарного дерева?
5. Что такое вершина бинарного дерева?
6. Чему равны указатели left и right вершины бинарного дерева?

Лабораторная работа №5

ПОРАЗРЯДНЫЕ ОПЕРАЦИИ И БИТОВЫЕ ПОЛЯ

Цель работы: изучить теорию и научиться программировать поразрядные операции и битовые поля.

Поразрядные операции языка С

Поразрядные операции состоят из четырех основных операций: отрицание, логическое И, логическое ИЛИ и исключающее ИЛИ. Рассмотрим данные операции по порядку.

При выполнении операции поразрядного отрицания все биты, равные 1, устанавливаются равными 0, а все биты равные нулю, устанавливаются равными 1. Для выполнения данной операции в языке С++ используется символ ‘~’ как показано в следующем примере:

```
unsigned char var = 153;  //двоичная запись 10011001
unsigned char not = ~var; //результат 01100110 (число 102)
```

В результате переменная not будет содержать число 102. В ходе выполнения операции поразрядного И результирующий бит будет равен 1, если оба бита в соответствующих операндах равны 1, т.е.

10010011 & 00111101 даст результат

00010001.

Для выполнения операции логического И используется символ & следующим образом:

```
unsigned char var = 153; //двоичная запись 10011001
unsigned char mask = 0x11;         // число 00010001 (число 17)
unsigned char res = var & mask;    // результат 00010001
```

или

```
var &= mask;     // то же самое, что и  var = var & mask;
```

В ходе выполнения двоичной операции ИЛИ результирующий бит устанавливается равным 1, если хотя бы один бит соответствующих операндов равен 1. В противном случае, результирующее значение равно 0. Для выполнения данной логической операции используется символ ‘|’ как показано ниже:

```
unsigned char var = 153;   //двоичная запись 10011001
unsigned char mask = 0x11; // число          00010001
unsigned char res = var | mask; // результат 10011001
```

Также допускается применение такой записи

```
var |= mask; // то же самое, что и var = var | mask;
```

Наконец, при операции исключающее ИЛИ результирующий бит устанавливается равным 0, если оба бита соответствующих операндов равны 1, и 1 в противном случае. Для выполнения данной операции в языке C++ используется символ ‘^’:

```
unsigned char var = 153; //двоичная запись   10011001
unsigned char mask = 0x11; // число          00010001
unsigned char res = var ^ mask; // результат 10001000
```

или

```
var ^= mask; // то же самое, что и var = var ^ mask;
```

Рассмотрим примеры использования логических операций, которые часто применяются на практике. Самой распространенной по использованию является операция логического И. Данная операция обычно используется совместно с так называемыми масками. Под маской понимают битовый шаблон, который служит для выделения тех или иных битов числа, к которому она применяется. Например, если требуется определить, является ли нулевой бит числа установленным в 1 или нет, то для этого задается маска 00000001, которая соответствует числу 1 и выполняется операция поразрядного И:

```
unsigned char flags = 3; // 00000011
unsigned char mask = 1;  // 00000001
if((flag & mask) == 1) printf(“Нулевой бит включен”);
else printf(“Нулевой бит выключен”);
```

Здесь переменная flags, представленная одним байтом, содержит восемь флаговых битов. Для того чтобы узнать установлен или нет нулевой флаговый бит задается маска со значением 1 и выполняется операция логического И. В результате все биты переменной flags будут равны нулю за исключением нулевого, если он изначально имел значение 1. Таким образом, маска является шаблоном, который как бы накладывается на битовое представление числа, из которого выделяются биты, соответствующие единичным значениям маски. Рассмотренный пример показывает, как одна байтовая переменная flags может содержать восемь флаговых значений и тем самым экономить память ЭВМ.

Следующим примером использования логических операция является возможность включать нужные биты в переменной, оставляя другие без изменений. Для этого используется логическая операция ИЛИ. Допустим, в переменной flags необходимо установить второй бит равным 1. Для этого задается маска в виде переменной mask = 2 (00000010) и реализуется операция логического ИЛИ:

```
unsigned char flags = 0; // 00000000
unsigned char mask = 2;  // 00000010
flags |= mask;
```

Этот код гарантирует, что второй бит переменной flags будет равен 1 без изменений значений других битов.

Для отключения определенных битов целесообразно использовать две логические операции: логическое И и логическое НЕ. Допустим, требуется отключить второй бит переменной flags. Тогда предыдущий пример запишется следующим образом:

```
unsigned char flags = 0; // 00000000
unsigned char mask = 2;  // 00000010
flags = flags & ~mask;
```

или

```
flags &= ~mask;
```

Работа этих двух операций заключается в следующем. Приоритет операции НЕ выше приоритета операции И, поэтому переменная mask в двоичной записи будет иметь вид 11111101. Применяя операцию логического умножения переменной flags с полученным числом ~mask все биты останутся неизменными, кроме второго, значение которого будет равно нулю.

Наконец, операция исключающее ИЛИ позволяет переключать заданные биты переменных. Идея переключения битов основывается на свойствах операции исключающего ИЛИ: 1^1 = 0, 1^0 = 1, 0^0 = 0 и 0^1 = 1. Анализ данных свойств показывает, что если значение бита маски будет равно 1, то соответствующий бит переменной, к которой она применяется, будет переключен, а если значение бита маски равно 0, то значение бита переменной останется неизменным. Следующий пример демонстрирует работу переключения битов переменной flags.

```
unsigned char flags = 0; //00000000
unsigned char mask = 2;  //00000010
flags ^= mask;           //00000010
flags ^= mask;           //00000000
```

Кроме логических операций в языке C++ существуют операции поразрядного смещения битов переменной. Операция смещения битов влево определяется знаком << и смещает биты значения левого операнда на шаг, определенный правым операндом, например, в результате выполнения команды

```
10001010 << 2;
```

получится результат 00101000. Здесь каждый бит перемещается влево на две позиции, а появляющиеся новые биты устанавливаются нулевыми. Рассмотрим особенности действия данной операции на следующем примере:

```
int var = 1;
var = var <<1; //00000010 – значение 2
var <<= 1;     //00000100 – значение 4
```

Можно заметить, что смещение битов переменной на одну позицию влево приводит к операции умножения числа на 2. В общем случае, если выполнить сдвиг битов на n шагов, то получим результат равный умножению переменной на 2^n. Данная операция умножения на число 2^n является более быстрой, чем обычное умножения, рассматриваемое ранее.

Аналогично, при операции смещения вправо >> происходит сдвиг битов переменной на шаг, указанный в правом операнде. Например, сдвиг

```
00101011 >> 2;
```

приведет к результату 00001010. Здесь, также как и при сдвиге влево, новые появляющиеся биты устанавливаются равными нулю. В результате выполнения последовательностей операций

```
int var = 128;  //1000000
var = var >> 1; //0100000 – значение 64
var >>= 1;      //0010000 – значение 32
```

значение переменной var каждый раз делится на 2. Поэтому сдвиг var >>= n можно использовать для выполнения операции деления значения переменной на величину 2^n.

Битовые поля

Стандарт C99, который часто является основой языка C++, позволяет описывать данные на уровне битов. Это достигается путем использования битовых полей, представляющие собой переменные типов signed или unsigned int, у которых используются лишь несколько бит для хранения данных. Такие переменные обычно записываются в структуру и единую последовательность бит. Рассмотрим пример, в котором задается структура flags, внутри которой задано 8 битовых полей:

```
struct {
 unsigned int first : 1;
 unsigned int second : 1;
 unsigned int third : 1;
 unsigned int forth : 1;
 unsigned int fifth : 1;
 unsigned int sixth : 1;
 unsigned int sevnth : 1;
```

```
 unsigned int eighth : 1;
} flags;
```

Теперь, для определения того или иного бита переменной flags достаточно воспользоваться операцией

```
flags.first = 1;
flags.third = 1;
```

В этом случае будут установлены первый и третий биты, а остальные равны нулю, что соответствует числу 5. Данное значение можно отобразить, воспользовавшись функцией printf():

```
printf("flags = %d.\n",flags);
```

но переменной flags нельзя присваивать значения как обычной переменной, т.е. следующий программный код будет неверным:

```
flags = 5; //неверно, так нельзя
```

Также нельзя присваивать значение flags переменным, например, следующая запись приведет к сообщению об ошибке:

```
int var = flags; //ошибка, структуру нельзя присваивать переменной
```

Так как поля first,…, eighth могут содержать только один бит информации, то они принимают значения 0 или 1 для типа unsigned int и 0 и -1 - для типа signed int. Если полю присваивается значение за пределами этого диапазона, то оно выбирает первый бит из присваиваемого числа.

В общем случае можно задавать любое число бит для описания полей, например

```
struct {
 unsigned int code1 : 2;
 unsigned int code2 : 2;
 unsigned int code3 : 8;
} prcode;
```

Здесь создается два двух битовых поля и одно восьмибитовое. В результате возможны следующие операции присваивания:

```
prcode.code1 = 0;
prcode.code2 = 3;
prcode.code3 = 128;
```

Структуры flags и prcode удобно использовать в условных операторах if и switch. Рассмотрим пример использования структуры битовых полей для описания свойств окна пользовательского интерфейса.

```
struct tag_window {
 unsigned int show : 1    //показать или скрыть
 unsigned int style : 3   //WS_BORDER, WS_CAPTION, WS_DLGFRAME
 unsigned int color : 3   //RED, GREEN, BLUE
} window;
Определим следующие константы:
#define WS_BORDER  1      //001
#define WS_CAPTION 2      //010
#define WS_DLGFRAME 4     //100
#define RED     1
#define GREEN 2
#define BLUE 4
#define SW_SHOW 0
#define SW_HIDE 1
```

Пример инициализации структуры битовых полей

```
window.show = SW_SHOW;
window.style = WS_BORDER | WS_CAPTION;
window.color = RED | GREEN | BLUE;  //белый цвет
```

Задание на лабораторную работу

1. Написать программу по работе с поразрядными операциями в соответствии с номером своего варианта.
2. Написать программу по работе с битовыми полями в соответствии с номером своего варианта.

Варианты заданий

Вариант	Поразрядные операции	Битовые поля
1	Используя операцию «логическое И», выделите первые три бита числа	Написать программу компактного представления календарной даты с соответствующим набором констант
2	Используя операцию «логическое ИЛИ», включите первый и третий биты числа	Написать программу компактного представления свойств окна (цвет, размер, наименование, меню, прозрачность в %) с соответствующим набором констант
3	Используя операцию «логическое НЕ» и «логическое И», выключите пятый и шестой биты числа	Написать программу компактного представления времени (мин:сек:час флаг – am/pm) с соответствующим набором констант
4	Используя операцию «исключающее ИЛИ», переключите 7-й бит числа	Напишите программу работы с 4-битовой переменной и задайте

		соответствующий набор констант для работы с ней
5	С помощью поразрядных операций выполните умножение числа 4 на 8	Напишите программу копирования одной 6-ти битовой переменной в другую 6-ти битовую переменную
6	С помощью поразрядных операций выполните деление числа 128 на 4	Напишите программу сортировки массива по возрастанию, состоящего из 4-х битовых переменных
7	Используя операцию «логическое НЕ» и «логическое И», выключите третий и пятый биты числа	Напишите программу сортировки массива по убыванию, состоящего из 6-ти битовых переменных
8	Используя операцию «исключающее ИЛИ», переключите 5-й бит числа	Напишите программу определения максимального значения элемента массива, состоящего из 4-х битовых переменных
9	С помощью поразрядных операций выполните умножение числа 8 на 4	Напишите программу определения максимального значения элемента массива, состоящего из 7-ми битовых переменных
10	С помощью поразрядных операций выполните деление числа 32 на 4	Напишите программу вычисления суммы элементов массива, состоящего из 4-х битовых переменных

Содержание отчета

1. Титульный лист с названием лабораторной работы, номером варианта, фамилией студента и группы.
2. Текст программ.
3. Результаты действия программ.
4. Выводы по полученным результатам работы программ.

Контрольные вопросы

1. Дайте понятие битового поля.
2. С помощью какой поразрядной операции осуществляется выделение битов переменных?
3. Какая поразрядная операция позволяет переключать биты переменной?
4. Для каких целей следует использовать битовые поля?
5. Объясните смысл поразрядной операции «логическое ИЛИ».
6. С помощью какой поразрядной операции осуществляется деление переменной на 2?

Лабораторная работа №6

РАБОТА С ФАЙЛАМИ

Цель работы: изучить теорию и научиться программировать операции файлового ввода/вывода.

Работа с текстовыми файлами

Для работы с файлами в языке С имеется набор функций, определенных в библиотеке stdio.h. Перед началом работы с файлом его следует открыть, что достигается с помощью функции fopen(), имеющей следующий синтаксис:

```
FILE *fopen( const char *filename, const char *mode );
```

Здесь filename – строка, содержащая путь и имя файла; mode – строка, определяющая режим открытия файла: на чтение или на запись; FILE – специальный тип данных для работы с файлами. Данная функция возвращает значение NULL, если файл не был успешно открыт, иначе – другое значение. Рассмотрим последовательность действий по созданию простого текстового файла на языке С и записи в него текстовой информации.

Листинг 7. Запись текстовой информации в файл.

```
#include <stdio.h>
int main()
{
 char str_file[]="Строка для файла";
 FILE* fp = fopen("my_file.txt","w");
 if(fp != NULL)
 {
      printf("Идет запись информации в файл…\n");
      for(int i=0;i < strlen(str_file);i++)
           putc(str_file[i],fp);
 }
 else printf("Невозможно открыть файл на запись.\n");
 fclose(fp);
 return 0;
}
```

В данном примере задается специализированный указатель fp типа FILE, который инициализируется функцией fopen(). Функция fopen() в качестве первого аргумента принимает строку, в которой задан путь и имя файла. Вторым параметром определяется способ обработки файла, в данном случае, значение “w”, которое означает открытие файла на запись с удалением всей прежней информации из него. Если файл открыт успешно, то указатель fp не будет равен NULL и с ним возможна работа. В этом случае с помощью

функции putc() выполняется запись символов в файл, на который указывает указатель fp. Перед завершением программы открытый файл следует закрыть во избежание в нем потери данных. Это достигается функцией fclose(), которая принимает указатель на файл и возвращает значение 0 при успешном закрытии файла, иначе значение EOF.

Рассмотрим теперь пример программы считывания информации из файла.

Листинг 8. Считывание текстовой информации из файла.

```
#include <stdio.h>
int main()
{
 char str_file[100];
 FILE* fp = fopen("my_file.txt","r");
 if(fp != NULL)
 {
      int i=0;
      char ch;
      while((ch = getc(fp)) != EOF)
           str_file[i++]=ch;
      str_file[i] = '\0';
      printf(str_file);
 }
 else printf("Невозможно открыть файл на чтение.\n");
 fclose(fp);
 return 0;
}
```

В приведенном листинге функция fopen() открывает файл на чтение, что определяется значением второго аргумента равного «r». Это значит, что в него невозможно произвести запись данных, а только считывание. Сначала выполняется цикл while, в котором из файла считывается символ с помощью функции getc() и выполняется проверка: если считанное значение не равно символу конца файла EOF, то значение переменной ch записывается в массив str_file. Данный цикл будет выполняться до тех пор, пока не будут считаны все символы из файла, т.е. пока не будет достигнут символ EOF. После завершения цикла формируется строка str_file, которая выводится на экран с помощью функции printf(). Перед завершением программы также выполняется функция закрытия файла fclose().

Работа с текстовыми файлами через функции putc и getc не всегда удобна. Например, если необходимо записать или считать строку целиком, то желательно иметь функции, выполняющие эту работу. В качестве таковых можно воспользоваться функциями fputs() и fgets() для работы со строками. Перепишем предыдущие примеры с использованием данных функций.

Листинг 9. Использование функций fpust() и fgets().

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

```
int main()
{
 char str_file[]=”Строка для файла”;
 FILE* fp = fopen(“my_file.txt”,”w”);
 if(fp != NULL) fputs(str_file,fp);
 else printf(“Невозможно открыть файл на запись.\n”);
 fclose(fp);

 fp =      fopen(“my_file.txt”,”r”);
 if(fp != NULL)
 {
      fgets(str_file,sizeof(str_file),fp);
      printf(str_file);
 }
 fclose(fp);

 return 0;
}
```

Аналогичные действия по записи данных в файл и считывания информации из него можно выполнить и с помощью функций fprintf() и fscanf(). Однако эти функции предоставляют большую гибкость в обработке данных файла. Продемонстрируем это на следующем примере. Допустим, имеется структура, хранящая информацию о книге: название, автор, год издания. Необходимо написать программу сохранения этой информации в текстовый файл и их считывания. Пример использования данных функций представлен в листинге 10.

Листинг 10. Использование функций fprintf() и fscanf().

```
#include <stdio.h>
#define N 2
struct tag_book
{
 char name[100];
 char author[100];
 int year;
} books[N];

int main(void)
{
 for(int i=0;i < N;i++)
 {
      scanf("%s",books[i].name);
      scanf("%s",books[i].author);
      scanf("%d",&books[i].year);
 }

 for(i=0;i < N;i++)
 {
      puts(books[i].name);
```

```
        puts(books[i].author);
        printf("%d\n",books[i].year);
    }

    FILE* fp = fopen("my_file.txt","w");
    for(i=0;i < N;i++)
        fprintf(fp,"%s %s %d\n",books[i].name,books[i].author,
books[i].year);
    fclose(fp);
    fp = fopen("my_file.txt","r");
    for(i=0;i < N;i++)
        fscanf(fp,"%s %s %d\n",books[i].name,books[i].author,
&books[i].year);
    fclose(fp);
    printf("-----------------------------------------------\n");
    for(i=0;i < N;i++)
    {
        puts(books[i].name);
        puts(books[i].author);
        printf("%d\n",books[i].year);
    }
      return 0;
}
```

При выполнении данной программы вводится информация по книгам в массив структур books и выводится введенная информация на экран. Затем открывается файл my_file.txt на запись, в который заносится информация по книгам в порядке: наименование, автор, год издания. Так как число книг в данном случае равно двум, то выходной файл будет выглядеть следующим образом:

Onegin Pushkin 1983
Oblomov Griboedov 1985

Затем, файл my_file.txt открывается на чтение и с помощью функции scanf() осуществляется считывание информации в элементы структуры. В заключении считанная информация выводится на экран монитора.

Представленный пример показывает возможность структурированной записи информации в файл и ее считывания. Это позволяет относительно просто сохранять разнородные данные в файле для их дальнейшего использования в программах.

При внимательном рассмотрении предыдущих примеров можно заметить, что функции считывания информации из файла «знают» с какой позиции следует считывать очередную порцию данных. Действительно, в последнем примере функция fscanf(), вызываемая в цикле, «знает» что нужно считать сначала первую строку из файла, затем вторую и т.д. И программисту нет необходимости задавать позицию для считывания данных. Все происходит автоматически. Вследствие чего появляется такая особенность работы? Дело в

том, что у любого открытого файла в программе написанной на С имеется указатель позиции (номера), с которой осуществляется считывание данных из файла. При открывании файла на чтение номер этой позиции указывает на начало файла. Поэтому функция fscanf(), вызванная первый раз, считывает данные первой строки. По мере считывания информации из файла, позиция сдвигается на число считанных символов. И функция fscanf() вызванная второй раз будет работать уже со второй строкой в файле. Несмотря на то, что указатель позиции в файле перемещается автоматически, в языке С имеются функции fseek() и ftell(), позволяющие программно управлять положением позиции в файле. Синтаксис данных функций следующий:

```
int fseek( FILE *stream, long offset, int origin );
long ftell( FILE *stream );
```

где *stream – указатель на файл; offset – смещение позиции в файле (в байтах); origin – флаг начального отсчета, который может принимать значения: SEEK_END – конец файла, SEEK_SET – начало файла; SEEK_CUR – текущая позиция. Последняя функция возвращает номер текущей позиции в файле.

Рассмотрим действие данных функций на примере считывания символов из файла в обратном порядке.

Листинг 11. Использование функций fseek() и ftell().

```
#include <stdio.h>
int main(void)
{
 FILE* fp = fopen("my_file.txt","w");
 if(fp != NULL)
 {
      fprintf(fp,"It is an example using fseek and ftell
functions.");
 }
 fclose(fp);
 fp = fopen("my_file.txt","r");
 if(fp != NULL)
 {
      char ch;
      fseek(fp,0L,SEEK_END);
      long length = ftell(fp);
      printf("length = %ld\n",length);
      for(int i = 1;i <= length;i++)
      {
           fseek(fp,-i,SEEK_END);
           ch = getc(fp);
           putchar(ch);
      }
 }
 fclose(fp);
 return 0;
```

```
}
```

В данном примере сначала создается файл, в который записывается строка “It is an example using fseek and ftell functions.”. Затем этот файл открывается на чтение и с помощью функции fseek(fp,0L,SEEK_END) указатель позиции помещается в конец файла. Это достигается за счет установки флага SEEK_END, который перемещает позицию в конец файла при нулевом смещении. В результате функция ftell(fp) возвратит число символов в открытом файле. В цикле функция fseek(fp,-i,SEEK_END) смещает указатель позиции на –i символов относительно конца файла, после чего считывается символ функцией getc(), стоящий на i-й позиции с конца. Так как переменная i пробегает значения от 1 до length, то на экран будут выведены символы из файла в обратном порядке.

Работа с бинарными файлами

Следует отметить, что во всех рассмотренных выше примерах функция fopen() в режимах “r” и “w” открывает текстовый файл на чтение и запись соответственно. Это означает, что некоторые символы форматирования текста, например возврат каретки ‘\r’ не могут быть считаны как отдельные символы, их как бы не существует в файле, но при этом они там есть. Это особенность текстового режима файла. Для более «тонкой» работы с содержимом файлов существует бинарный режим, который представляет содержимое файла как последовательность байтов где все возможные управляющие коды являются просто числами. Именно в этом режиме возможно удаление или добавление управляющих символов недоступных в текстовом режиме. Для того чтобы открыть файл в бинарном режиме используется также функция fopen() с последним параметром равным “rb” и “wb” соответственно для чтения и записи. Продемонстрируем особенности обработки бинарного файла на примере подсчета числа управляющих символов возврата каретки ‘\r’ в файле, открытый в текстовом режиме и бинарном.

Листинг 12. Программа подсчета числа символов ‘\r’ в файле.

```
#include <stdio.h>
int main(void)
{
 FILE* fp = fopen("my_file.txt","w");
 if(fp != NULL)
 {
      fprintf(fp,"It is\nan example using\nan binary file.");
 }
 fclose(fp);
 char ch;
 int cnt = 0;
 fp = fopen("my_file.txt","r");
 if(fp != NULL)
```

```
    {
        while((ch = getc(fp)) != EOF)
            if(ch == '\r') cnt++;
    }
    fclose(fp);
    printf("Text file: cnt = %d\n",cnt);
    cnt=0;
    fp = fopen("my_file.txt","rb");
    if(fp != NULL)
    {
        while((ch = getc(fp)) != EOF)
            if(ch == '\r') cnt++;
    }
    fclose(fp);
    printf("Binary file: cnt = %d\n",cnt);
    return 0;
}
```

Результат работы будет следующий:

Text file: cnt = 0
Binary file: cnt = 2

Анализ полученных данных показывает, что при открытии файла в текстовом режиме, символы возврата каретки ‘\r’ не считываются функцией getc(), а в бинарном режиме доступны все символы.

Еще одной особенностью текстового формата файла является запись чисел в виде текста. Действительно, когда в предыдущих примерах выполнялась запись числа в файл с помощью функции fprintf(), например, года издательства книги, то число заменялось строкой. А когда она считывалась функцией fscanf(), то преобразовывалась обратно в число. Если мы хотим компактно представлять данные в файле, то числа следует хранить как числа, а не как строки. При этом целесообразно использовать бинарный режим доступа к файлу, т.к. будет гарантия, что любое записанное число не будет восприниматься как управляющий символ и будет корректно считан из файла.

Для работы с бинарными файлами предусмотрены функции fread() и fwrite() со следующим синтаксисом:

```
    size_t fread( void *buffer, size_t size, size_t count, FILE
*stream );
```

где *buffer – указатель на буфер памяти, в который будут считываться данные из файла; size – размер элемента в байтах; count - число считываний элементов; *stream – указатель на файл.

```
    size_t fwrite( void *buffer, size_t size, size_t count, FILE
*stream );
```

где *buffer – указатель на буфер памяти, из которого будут считываться данные в файл; size – размер элемента в байтах; count - число записей; *stream – указатель на файл.

Приведем пример использования функций fwrite() и fread().

Листинг 13. Использование функций fwrite() и fread().

```
#include <stdio.h>
void main( void )
{
   FILE *stream;
   char list[30];
   int  i, numread, numwritten;
   if( (stream = fopen( "fread.out", "wb" )) != NULL )
   {
      for ( i = 0; i < 25; i++ )
         list[i] = (char)('z' - i);
      numwritten = fwrite( list, sizeof( char ), 25, stream );
      printf( "Wrote %d items\n", numwritten );
      fclose( stream );
   }
   else  printf( "Problem opening the file\n" );
   if( (stream = fopen( "fread.out", "rb" )) != NULL )
   {
      numread = fread( list, sizeof( char ), 25, stream );
      printf( "Number of items read = %d\n", numread );
      printf( "Contents of buffer = %.25s\n", list );
      fclose( stream );
   }
   else printf( "File could not be opened\n" );
}
```

В данном примере массив list выступает в качестве буфера для вывода и ввода информации из бинарного файла. Сначала элементы буфера инициализируются буквами латинского алфавита от z до b, а затем записываются в файл с помощью функции fwrite(list, sizeof(char), 25, stream). Здесь оператор sizeof(char) указывает размер элемента (буквы), а число 25 соответствует числу записываемых букв. Аналогичным образом осуществляется считывание информации из файла fread(list, sizeof(char), 25, stream), где в массив list помещаются 25 символов, хранящихся в файле.

Функции fwrite() и fread() удобно использовать при сохранении данных структуры в файл. Запишем пример хранения информации по двум книгам в бинарном файле.

Листинг 14. Пример сохранения структур в бинарном файле.

```
#include <stdio.h>
#define N 2
struct tag_book
{
```

```
 char name[100];
 char author[100];
 int year;
} books[N];

int main(void)
{
 for(int i=0;i < N;i++)
 {
      scanf("%s",books[i].name);
      scanf("%s",books[i].author);
      scanf("%d",&books[i].year);
 }
 FILE* fp = fopen("my_file.txt","wb");
 fwrite(books, sizeof(books),1,fp);
 fclose(fp);

 fp = fopen("my_file.txt","rb");
 fread(books,sizeof(books),1,fp);
 fclose(fp);
 printf("-----------------------------------------------\n");
 for(i=0;i < N;i++)
 {
      puts(books[i].name);
      puts(books[i].author);
      printf("%d\n",books[i].year);
 }
    return 0;
}
```

В данном примере с помощью функции fwrite() целиком сохраняется массив books, состоящий из двух элементов, а оператор sizeof(books) определяет размер массива books. Аналогичным образом реализуется и функция fread(), которая считывает из файла сразу весь массив. По существу функции fwrite() и fread(), в данном примере, осуществляют копирование заданной области памяти в файл, а затем обратно. Это их свойство удобно использовать при хранении «сложных» форм данных, когда простая поэлементная запись данных в файл становится трудоемкой или невозможной.

Следует отметить, что функция fopen() при открытии файла на запись уничтожает все данные из этого файла, если они были. Вместе с тем существует необходимость добавлять данные в файл, не уничтожая ранее записанную информацию. Это достигается путем открытия файла на добавление информации. В этом случае функции fopen() третьим аргументом передается строка “a” или “ab”, что означает открыть файл на добавление информации в его конец. Продемонстрируем работу данного режима на следующем примере.

Листинг 15. Добавление информации в файл.

```
#include <stdio.h>
#define N 2
struct tag_book
{
 char name[100];
 char author[100];
 int year;
} books[N];
int main(void)
{
 for(int i=0;i < N;i++)
 {
      scanf("%s",books[i].name);
      scanf("%s",books[i].author);
      scanf("%d",&books[i].year);
 }
 FILE* fp = fopen("my_file.txt","wb");
 fwrite(&books[0], sizeof(tag_book),1,fp);
 fclose(fp);
 fp = fopen("my_file.txt","ab");
 fwrite(&books[1], sizeof(tag_book),1,fp);
 fclose(fp);
 fp = fopen("my_file.txt","rb");
 fread(books,sizeof(books),1,fp);
 fclose(fp);
 printf("-------------------------------------------------\n");
 for(i=0;i < N;i++)
 {
      puts(books[i].name);
      puts(books[i].author);
      printf("%d\n",books[i].year);
 }
    return 0;
}
```

В данном примере сначала создается файл my_file.txt, в который записывается информация по первой книге. Затем открывается этот же файл в режиме добавления и записывается информация по второй книге. В результате файл my_file.txt содержит информацию по обеим книгам, что подтверждается считыванием данных из этого файла и выводом информации на экран.

Когда стандартные функции возвращают EOF, это обычно означает, что они достигли конца файла. Однако это также может означать ошибку ввода информации из файла. Для того чтобы различить эти две ситуации в языке C++ существую функции feof() и ferror(). Функция feof() возвращает значение отличное от нуля, если достигнут конец файла и нуль в противном случае. Функция ferror() возвращает ненулевое значение, если произошла ошибка чтения или записи, и нуль – в противном случае. Пример использования данных функций представлен в листинге 16.

Листинг 16. Использование функции ferror().

```
#include <stdio.h>
void main( void )
{
   int  count, total = 0;
   char buffer[100];
   FILE *fp;

   if( (fp = fopen( "my_file.txt", "r" )) == NULL )
      return;
   while( !feof( fp ) )
   {
      count = fread( buffer, sizeof( char ), 100, fp );
      if( ferror( fp ) )      {
         perror( "Read error" );
         break;
      }
      total += count;
   }
   printf( "Number of bytes read = %d\n", total );
   fclose( fp );
}
```

В языке С имеются также функции remove() и rename() для удаления и переименования файлов. Их синтаксис следующий:

```
int remove( const char *path );
```

где *path – путь с именем удаляемого файла. Данная функция определена в библиотеках stdio.h и io.h, возвращает нуль при успешном удалении и -1 в противном случае.

```
int rename( const char *oldname, const char *newname );
```

где *oldname – имя файла для переименования; *newname – новое имя файла. Данная функция определена в библиотеках stdio.h и io.h, возвращает нуль при успешном удалении и не нуль в противном случае.

Задание на лабораторную работу

1. Написать программу по работе с файлами в соответствии с номером своего варианта.

Варианты заданий

Вариант	Задание на программирование файлов
1	Написать программу сохранения в файл информации по 10-ти книгам, которые находятся в массиве структур
2	Написать программу загрузки из файла в массив структур

	информации по 10-ти книгам
3	Написать программу копирования содержимого одного файла в другой
4	Написать программу обмена данными между двумя файлами
5	Написать программу сохранения в файл и считывания из файла стека
6	Написать программу сохранения в файл и считывания из файла связного списка
7	Написать программу подсчета слов в текстовом файле (слова разделяются пробелом)
8	Написать программу удаления из текстового файла символов перевода строки ‘\n’ и перевода каретки ‘\r’
9	Написать программу удаления из текстового файла букв ‘а’
10	Написать программу добавления в текстовый файл после каждой буквы ‘о’ одного пробела

Содержание отчета

1. Титульный лист с названием лабораторной работы, номером варианта, фамилией студента и группы.
2. Текст программы.
3. Результаты действия программы.
4. Выводы о полученных результатах работы программы.

Контрольные вопросы

1. Дайте понятие файла.
2. Для чего предназначена функция fopen() и в какой библиотеке она определена?
3. Чему должен быть равен второй аргумент функции fopen() для открытия файла на чтение?
4. Какое значение возвращает функция fopen() при неудачном открытии файла?
5. Дайте понятие текстового режима доступа к файлу.
6. Для чего предназначены функции getc(), fgets() и fscanf()?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Дейтел, Харвин М. Как программировать на C++.: Пер. с англ. – 3-е изд. – М.: Бином, 2003.

2. Дэвис, Стефан Р. C++ «для чайников».: Пер. с англ. – 4-е изд.- М. [и др.]: Диалектика, 2001.

3. Культин, Никита. C/C++ в задачах и примерах.: учеб. пособие для вузов. – СПб.: BHV-Санкт-Петербург, 2001.

4. Литвиненко, Николай Аркадьевич. Технология программирования на C++. Начальный курс.: учеб. для вузов. – СПб.: БХВ-Петербург, 2005.

5. Мейн, Майкл. Структура данных и другие объекты в C++.: Пер с англ. – 2-е изд. – М.: Изд. дом «Вильямс», 2002.

Учебное издание

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С

Методические указания к лабораторным работам

Составитель НАМЕСТНИКОВ Сергей Михайлович

Редактор О. А. Семенова

Подписано в печать 09. 09. 2008. Формат 60×84/16.
Усл. печ. л. 1,65.
Тираж 60 экз. Заказ
Ульяновский государственный технический университет,
432027, Ульяновск, Сев. Венец, 32.
Типография УлГТУ, 432027, Ульяновск, Сев. Венец, 32.