МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТРАНСПОРТА»

Кафедра «Информационные технологии»

Д. В. БАЛАЩЕНКО, Д. В. ЗАХАРОВ

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C/C++

Учебно-методическое пособие по дисциплине «Информатика и информационные технологии»

Часть 2

Гомель 2011

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТРАНСПОРТА»

Кафедра «Информационные технологии»

Д. В. БАЛАЩЕНКО, Д. В. ЗАХАРОВ

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ C/C++

Учебно-методическое пособие по дисциплине
«Информатика и информационные технологии»

Ч а с т ь 2

Одобрено методической комиссией
электротехнического факультета

Гомель 2011

УДК 004.43 (075.8)
ББК 32.973-01.8
Б20

Р е ц е н з е н т – зав. кафедрой “Физика” канд. физ.-мат. наук, доцент *В. А. Зыкунов* (УО «БелГУТ»)

Балащенко, Д. В.

Б20 Программирование на языке С/С++ : учеб.-метод. пособие по дисциплине «Информатика и информационные технологии». В 2 ч. Ч. 2 / Д. В. Балащенко, Д. В. Захаров ; М-во образования Респ. Беларусь, Белорус. гос. ун-т трансп. – Гомель : БелГУТ, 2011. – 56 с.
ISBN 978-985-468-779-5 (ч. 2)

Рассмотрены основы программирования на языке С/С++. Первая часть пособия была издана в 2006 году.
Предназначено для студентов электротехнических специальностей.

УДК 004.43 (075.8)
ББК 32.973-01.8

ISBN 978-985-468-779-5 (ч. 2)
ISBN 985-468-137-8

ОГЛАВЛЕНИЕ

1 СТРУКТУРЫ

Структура – это совокупность нескольких переменных, как правило, разных типов, сгруппированных под единым именем.

Допустим, нам необходимо описать точку на прямой. Каждая точка характеризуется именем, состоящим из одной латинской буквы, и координатой. Имя имеет тип char, а координата – тип double.

Данная структура объявляется следующим образом:

```
struct point {
  char name;
  double x;
};
```

Вышеописанная структура объединяет две части name и x. Эти части структуры называются полями.

В общем случае структура объявляется следующим образом:

```
struct тег_структуры {
  тип_поля_1 имя_поля_1;
  тип_поля_n имя_поля_n;
};
```

Необходимо отметить, что вышеприведенная конструкция просто объявляет структуру с заданным тегом (в нашем случае point) , но при этом переменные не создаются.

Для создания переменной pt1 с типом необходимо добавить ниже

```
struct point pt1;
```

Переменная pt1 называется „переменная типа структуры“, или „переменная-структура“.

В общем случае переменная-структура объявляется следующим образом:

```
struct тег_структуры имя_переменной-структуры
```

Доступ в программе к членам структуры осуществляется с помощью оператора точка(.).

Например, для обращения к полю name переменной-структуры pt1 необходимо записать

```
pt1.name
```

С полем структуры можно работать так же, как и с переменной, соответствующего типа.

Например, чтобы определить точку A с координатой, равной 2.5, необходимо полям переменной-структуры присвоить соответствующие значения:

```
pt1.name = 'A';
pt1.x = 2.5;
```

Следующая программа находит расстояние между точкой C, имеющей координаты 2.5, и точкой, параметры которой вводятся с клавиатуры.

Листинг 1.1

```
#include<stdio.h>
#include<math.h>
struct point {
  char name;
  double x;
};
int main (void){
  struct point pt1, pt2={'C',2.5};
  printf("Введите имя точки:");
  pt1.name=getchar();
  printf("Введите координаты точки:");
  scanf("%lf",&pt1.x);
  printf("Расстояние между точками %c и %c равно %f\n",
pt1.name, pt2.name, fabs(pt2.x - pt1.x) );
  return 0;
}
```

В данном случае переменная-структура pt2 инициализируется значениями 'C' (имя точки) и 2.5 (координата точки):

```
struct point pt2={'C',2.5};
```

При написании программы нахождения расстояния между двумя точками, когда параметры и одной и другой вводятся с клавиатуры, может возникнуть неприятность, если код ввода написать следующим образом:

```
printf("Введите имя точки:");
pt1.name=getchar();
printf("Введите координаты точки:");
scanf("%lf",&pt1.x);
printf("Введите имя точки:");
pt2.name=getchar();
printf("Введите координаты точки:");
scanf("%lf",&pt2.x);
```

При запуске программа пропустит ввод имени для второй точки.

Проблема заключается в чередовании ввода численных и символьных данных. При вводе численных данных с помощью функции scanf() в буфере ввода, как правило, остается символ конца строки '\n'. Если далее стоит следующая функция scanf() для ввода следующих численных данных, то ошибки не произойдет, т. к. в этом случае '\n' будет игнорироваться.

Функции ввода символов или строк воспринимают '\n' как введенный символ и записывают его во вводимую переменную.

В нашем случае в поле name переменной pt2 и запишется символ '\n'.

При выводе на экран '\n' проявит себя в виде перехода на следующую строку:

```
Расстояние между точками A и
 равно 2.500000
```

Чтобы не допускать этой ошибки, необходимо помнить, что всегда после ввода численных данных и перед вводом символов или строк необходимо очищать буфер ввода с помощью следующей конструкции:

```
while(getchar()!='\n')
  continue;
```

В этом случае код программы из листинга * примет следующий вид:

```
printf("Введите имя точки:");
pt1.name=getchar();
printf("Введите координаты точки:");
scanf("%lf",&pt1.x);
while(getchar()!='\n')
  continue;
printf("Введите имя точки:");
pt2.name=getchar();
printf("Введите координаты точки:");
scanf("%lf",&pt2.x);
```

Можно очищать буфер ввода вообще после каждой функции scanf() с численными спецификаторами независимо от следующего оператора ввода.

Изменим условие задачи.

Необходимо написать программу, которая запрашивает количество точек на прямой и их параметры (имя и координата). Далее программа запрашивает имена точек и выдает расстояние между ними на экран. После этого пользователю предлагается продолжить расчет расстояния между другими точками, либо закончить программу.

Точек может быть как угодно много. Поэтому объявление нескольких переменных-структур с разными названиями нецелесообразно.

Как при объединении переменных одного типа мы создаем массив данного типа, так и в нашем случае мы можем создать массив структур.

```
struct point pt[100];
```

В данном случае объявляется массив pt, состоящий из 100 элементов, каждый из которых является переменной-структурой со схемой point. Об-

ращение к полю name i-го элемента массива pt осуществляется следующим образом:

```
pt[i].name
```

Программа для данной задачи написана ниже в листинге 1.2:

Листинг 1.2

```
#include<stdio.h>
#include<math.h>
struct point {
  char name;
  double x;
};
int main (void){
  int i,k1,k2,n;
  char name_p;
  struct point pt[10];
  printf("Введите количество вводимых точек:");
  scanf("%d",&n);
  while(getchar()!='\n')
    continue;
  printf("Ввод параметров точек:\n");
  for (i=0; i<n; i++){
    printf("%d-я точка:\n",i);
    printf("\tВведите имя точки: ");
    pt[i].name=getchar();
    printf("\tВведите координаты точки: ");
    scanf("%lf",&pt[i].x);
    while(getchar()!='\n')
      continue;
  }
  printf("Ввод завершен\n\n");
  do {
    do {
      printf("Введите имя первой точки: ");
      name_p=getchar();
      while(getchar()!='\n')
        continue;
      i=0;
      for (i=0; (pt[i].name != name_p) && (i<n); i++)
        continue;
      if (i==n) {
        printf("Введенное имя первой точки"
               "не найдено\n");
        printf("Повторите ввод\n");
        name_p='\0';
      }
      else
```

```
            k1=i;

        }while( name_p == '\0');

        do {
          printf("Введите имя второй точки: ");
          name_p=getchar();
          while(getchar()!='\n')
            continue;
          i=0;
          for (i=0; (pt[i].name != name_p) && (i<n); i++)
            continue;
          if (i==n) {
            printf("Введенное имя первой точки"
                   "не найдено\n");
            printf("Повторите ввод\n");
            name_p='\0';
          }
          else
            k2=i;

        }while( name_p == '\0');

        printf("Расстояние между точками %c и %c равно %f\n",
pt[k1].name, pt[k2].name, fabs(pt[k2].x - pt[k1].x) );

        printf("Хотите выйти из программы? (нажмите y)");
        name_p=getchar();
        while(getchar()!='\n')
          continue;


      } while(name_p != 'y');

      printf("Конец программы\n");
      return 0;
    }
```

Ниже показана программа, позволяющая вводить атрибуты книги в виде ее названия и количества страниц, записывать данные параметры в массив структур, а также выводить все параметры книг на экран.

Листинг 1.3

```
#include <stdio.h>
#include <ctype.h>
#define SIZE 5
```

```
struct book {
    char nazvanie[20];
    int page;
};
int main(void){
    struct book a[SIZE];
    int punkt;
    int i, count=0;
    system("clear");
    while (1){
    printf("Выберите пункт меню: ");
    printf("1 - Ввод, 2 - Вывод, 3 - Выход из программы: ");
    scanf("%d", &punkt);
    while (getchar()!='\n')
      continue;
    switch (punkt) {
      case 1:
        for (count=0; count<SIZE; count++){
          printf("Введите название %d книги: ", count+1);
          gets(a[count].nazvanie);
          printf("Введите количество страниц в книге: ");
          scanf("%d", &a[count].page);
          while (getchar()!='\n')
            continue;
          printf("Закончить ввод?: y/n ");
          if (toupper(getchar())=='Y') {
            count++;
            break;
          }
          else {
            while (getchar()!='\n')
              continue;
            continue;
          }
        }
        break;
      case 2:
        for (i=0; i<count; i++)
          printf("%3d книга  -  %s %s %d\n", i+1,
                  a[i].nazvanie, a[i].page);
        break;
      case 3:
        printf("Программа завершила свою работу. "
                "Пока!!!\n");
        return 0;
        break;
```

```
        default:
          printf("Пункт меню выбран неправильно. Попробуйте"
                 "еще раз!!!\n");
          break;
      }
    }
    return 0;
}
```

Назовем эту программу “СПРАВОЧНИК”. Мы будем к ней обращаться во время изучения оставшихся глав пособия, постепенно добавляя новые функциональные возможности и расширяя ее.

Указатель на структуру, как и другие указатели, объявляются с помощью звездочки (*).

Общий вид объявления указателя на структуру:

```
struct тег_структуры *имя_указателя_на_переменную_структуру;
```

Например:

```
struct book *next;
```

Чтобы с помощью указателя на структуру получить доступ к ее полям, необходимо использовать оператор ->

```
указатель_на_структуру -> поле_структуры
```

Ниже приведен пример той же программы, но с использованием указателей на структуры и связанных списков.

Листинг 1.4

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
struct book {
  char nazvanie[30];
  int page;
  struct book *next;
};
int main(void){
  char choice;
  int punkt;
  int i, count=0;
  struct book *head=NULL;
  struct book *prev, *current;
  system("clear");
  while (1){
    printf("Выберите пункт меню: ");
    printf("1 - Ввод, 2 - Вывод, 3 - Выход из программы: ");
    scanf("%d", &punkt);
    while (getchar()!='\n')
      continue;
```

```
        switch (punkt) {
          case 1:
            if (count!=0){
              while (1) {
                printf("Создать новую таблицу или дописывать"
                       "в старую? (N- new, O - old): ");
                choice=toupper(getchar());
                if (choice=='N') {
                  count=0;
    //использовать free() для очистки памяти
                  prev=head;
                  while(prev != NULL){
                    current=prev->next;
                    free(prev);
                    prev=current;
                  }
    //-----------------------------------------------
                  head=NULL;
                }
                if (choice!='N' && choice != 'O'){
                  while (getchar()!='\n')
                    continue;
                  continue;
                }
                while (getchar()!='\n')
                  continue;
                break;
              }
            }
            for ( ; ; count++){
              current=(struct book *) malloc(sizeof(struct
book)); //выделяем место для новой записи
              if (head==NULL)
                head=current;
              else
                prev->next=current;
              current->next=NULL;
              printf("Введите название %d книги: ", count+1);
              gets(current->nazvanie);
              printf("Введите количество страниц в книге: ");
              scanf("%d", &current->page);
              while (getchar()!='\n')
                continue;
              prev=current;
              printf("Закончить ввод?: y/n ");
              if (toupper(getchar())=='Y') {
                count++;
```

```
              break;
            }
            else {
              while (getchar()!='\n')
                continue;
              continue;
            }
          }
          break;
        case 2:
          if (head==NULL)
            printf("Данные не введены!\n");
          else
            printf("Список книг:\n");
          current=head;
          i=0;
          while (current!=NULL){
            printf("%d книга  -  %s %d\n", ++i, current-
>nazvanie, current->page);
            current=current->next;
          }
          break;
        case 3:
       //использовать free() для очистки памяти
          prev=head;
          while(prev != NULL){
            current=prev->next;
            free(prev);
            prev=current;
          }
   //-----------------------------------------------
          printf("Программа завершила свою работу.
Пока!!!\n");
       //использовать free()
          return 0;
       default:
         printf("Пункт меню выбран неправильно. Попробуйте
еще раз!!!\n");
         break;
       }
     }
     return 0;
   }
```

В вышеприведенной программе данные хранятся с помощью связных списков.

В структуру добавляется поле next, которое является указателем на следующий элемент.

```
struct book {
  char nazvanie[30];
  int page;
  struct book *next;
};
```

Кроме этого вводятся указатель на начальный элемент head и указатели на текущий элемент current и prev.

Пример связных списков показан на рисунке 1.1.

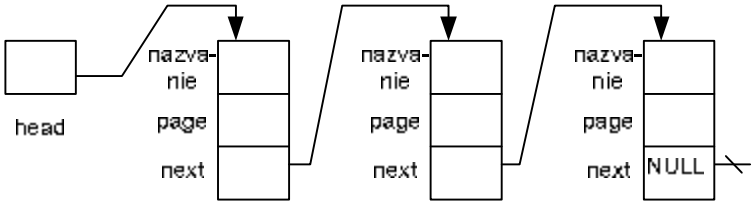

Рисунок 1.1 – Пример связного списка.

Указатель head инициализируется значением NULL, это означает, что список пока пустой.

При вводе данных под каждый элемент память выделяется с помощью функции malloc():

```
current=(struct book *) malloc(sizeof(struct book));
```

При этом данный адрес записывается в поле next предыдущего элемента-структуры или в переменную head, если вводится первый элемент и список был до этого пустой.

```
if (head==NULL)
  head=current;
else
  prev->next=current;
  current->next=NULL;
```

Если переменная-указатель head равна нулевому значению,то пользователь информируется, что данные не введены, в противном случае происходит последовательный вывод всех элементов:

```
if (head==NULL)
  printf("Данные не введены!\n");
else
  printf("Список книг:\n");
  current=head;
  i=0;
  while (current!=NULL){
  printf("%d книга  -  %s %d\n", ++i, current->nazvanie,
current->page);
  current=current->next;
```

```
}
break;
```

При выходе из программы с помощью функции free() последовательно освобождается память, отводимая под элементы. Цикл while выполняется до тех пор, пока указатель на следующий элемент не станет равным нулевому значению, что означает достижение конца списка.

```
prev=head;
while(prev != NULL){
  current=prev->next;
  free(prev);
  prev=current;
}
```

2 ФУНКЦИИ

2.1 Простые функции

Философия разработки программ на языке С заключается в использовании функций в качестве строительных блоков. В процессе разработки программ мы уже использовали различные функции, например такие как printf(), scanf(), pow() и др. Сейчас перед нами стоит более сложная задача – разработка собственных функций.

Функция – это самостоятельный фрагмент исходного текста программы, предназначенный для выполнения конкретной задачи. Функция может вызывать выполнение действий и возвращать значения. Функции избавляют пользователя от необходимости повторного программирования. Любая последовательность операторов, встречающаяся в программе более одного раза, будучи вынесенной в отдельную функцию, сокращает размер программы, т. к. ее код хранится только в одной области памяти.

Допустим, необходимо создать программу, которая должна выполнять следующие действия:

генерирует 10 чисел

сортирует их по возрастанию

находит их сумму

Для выполнения этой задачи можно было бы использовать следующую программу:

Листинг 2.1

```
#include <stdio.h>
#define SIZE 10
int main(void){
  float mass[SIZE];
  generator(mass, SIZE);
  sort(mass, SIZE);
```

```
  sum(mass, SIZE);
  return 0;
}
```

Конечно, в этом случае необходимо создать также функции generator(), sort()и sum(). Осмысленные имена функций позволяют достаточно легко понять, что делает программа и как она организована. Далее необходимо заняться каждой функцией, заставить ее правильно работать.

Напишем программу, в которой будет создана функция, выводящая 20 символов звездочки на экран.

Листинг 2.2

```
#include <stdio.h>
#define COUNT 20
void zvezdochka(void); //прототип функции
int main(void){
  zvezdochka();    //вызов функции
  printf("А сейчас еще раз вызовем функцию\n");
  zvezdochka();    //еще один вызов функции
  return 0;
}
void zvezdochka(){   //описание функции
  int i;
  for (i=0; i<COUNT; i++)
    putchar('*');
  putchar('\n');
}
```

Идентификатор zvezdochka используется в этой программе в трех различных контекстах: в *прототипе функции*, который сообщает компилятору, какого рода функцией является функция zvezdochka(); в *вызове функции*, приводящем к ее выполнению; и в определении функции, в котором указываются действия, выполняемые функцией.

По аналогии с переменными, функции имеют типы. Каждая программа, использующая функцию, должна объявлять тип этой функции до ее использования по аналогии с объявлением переменных. Поэтому прототип функции предшествует определению функции main():

```
void zvezdochka();
```

Круглые скобки говорят о том, что zvezdochka является именем функции. Ключевое слово void указывает тип функции; тип void показывает, что функция zvezdochka() не возвращает значение. Ключевое слово void, стоящее в круглых скобках справа от имени функции, показывает, что функция не принимает никаких аргументов. Точка с запятой говорит о том, что здесь выполняется объявление функции, а не ее определение.

Прототип функции говорит компилятору о том, что “функция, имеющая данные атрибуты, будет описана позже”.

Функция zvezdochka() дважды вызывается из функции main(). Эти два вызова выглядят одинаково:

```
zvezdochka();
```

Для вызова функции необходимо имя функции и круглые скобки, при этом не указывается тип возвращаемого значения. Выполнение оператора вызова функции инициирует выполнение самой функции. При этом управление передается операторам вызываемой функции, которые после своего выполнения передают управление оператору, следующему за вызовом функции.

Определение функции содержит код функции. Для функции zvezdochka() определение выглядит следующим образом:

```
void zvezdochka(){
  int i;
  for (i=0; i<COUNT; i++)
    putchar('*');
  putchar('\n');
  return ;
}
```

Определение состоит из *заголовка* и *тела* функции. Тело функции состоит из последовательности операторов, заключенной в фигурные скобки. Заголовок функции должен соответствовать своему прототипу. При этом имя функции и тип возвращаемого значения должны совпадать с указанными в прототипе. Аргументы функции, если они есть, должны иметь те же типы и быть записаны в том же порядке, в каком они указывались в прототипе функции.

В случае вызова функции, программа передает управление первому оператору тела функции, после чего выполняются операторы, находящиеся в теле функции, и когда достигается закрывающая фигурная скобка, управление передается обратно вызывающей функции.

В наших программах мы использовали различные библиотечные функции, такие как printf(), scanf() и др. Объявления библиотечных функций содержатся в заголовочных файлах, подключаемых к программе (в случае функций printf() и scanf() таким файлом является stdio.h). Определение библиотечных функций, уже скомпилированное в исполняемый код, находится в библиотечном файле, содержимое которого с помощью компоновщика автоматически включается в исполняемый код программы.

Если мы используем библиотечную функцию, нам не нужно самим создавать ее объявление и определение, но в случае разработки собственных функций объявление и определение должны присутствовать в исходном коде программы.

Переменная i в функции zvezdochka() является локальной переменной. Это означает, что она известна только в функции zvezdochka(). Имя i можно использовать в других функциях, в том числе и в функции main(), и это не

приведет к возникновению конфликта. Просто в этом случае в программе будут использоваться отдельные независимые переменные, имеющие одно и то же имя.

Если представить функцию zvezdochka() в виде черного ящика, то в его функции входит печать строки, состоящей из звездочек. Функция не имеет входного потока, т. к. не нуждается в использовании какой-либо информации от вызывающей функции. Она также не возвращает (не передает) никакую информацию функции main(), и поэтому функция zvezdochka() не имеет возвращаемого значения.

2.2 Передача аргументов функции

Аргументом называют единицу данных (например, переменную типа float), передаваемую программой в функцию. Аргументы позволяют функции оперировать различными значениями или выполнять различные действия в зависимости от переданных ей значений.

В качестве примера сделаем функцию zvezdochka() из предыдущей программы более гибкой. Для этого изменим ее таким образом, чтобы она выводила на экран не фиксированное количество звездочек, а любое количество любых символов. В качестве аргументов для функции zvezdochka() используется символ, предназначенный для вывода на экран, а также число раз, которое данный символ будет напечатан.

Листинг 2.3

```
#include <stdio.h>
void zvezdochka(char symbol, int count); //прототип функции
int main(void){
  zvezdochka('+', 40);   //вызов функции
  printf("А сейчас еще раз вызовем функцию\n");
  zvezdochka('*', 50);   //еще один вызов функции
  return 0;
}
void zvezdochka(char symbol, int count){ //описание функции
  int i;
  for (i=0; i<count; i++)
    putchar(symbol);
  putchar('\n');
}
```

Прототип новой функции zvezdochka() выглядит следующим образом:

```
void zvezdochka(char symbol, int count);
```

Когда функция принимает аргументы, в скобках указаны типы данных, которые будут иметь передаваемые в функцию аргументы: char и int. При желании имена переменных в прототипе могут быть опущены:

```
void zvezdochka(char, int);
```

При вызове функции вместо аргументов в скобках указываются их конкретные значения (в данном случае константы):

```
zvezdochka('+', 40); и
zvezdochka('*', 50);
```

Результатом первого вызова будет печать 40 символов +, а второго – печать 50 символов *. При указании значений аргументов важно соблюдать порядок их следования: сначала должно быть указано значение типа char, соответствующее символу, выводимому на экран, а затем значение типа int, которое определяет число раз, которое указанный символ будет напечатан. Кроме того, типы аргументов в объявлении и определении функции должны быть также согласованы.

Определение функции начинается со следующего функционального заголовка:

```
void zvezdochka(char symbol, int count)
```

В этой строке содержится информация о том, что функция zvezdochka() использует два аргумента, причем symbol имеет тип char, а count – тип int. Переменные symbol и count называются формальными переменными. Формальные переменные являются локальными переменными, частными для данной функции.

Стандарт языка С требует, чтобы каждой переменной предшествовал ее тип.

Хотя функция zvezdochka() принимает значения от функции main(), она не имеет возвращаемого значения.

Значения symbol и count передаются функции путем использования *фактических переменных*. Рассмотрим первый случай использования функции zvezdochka():

```
zvezdochka('+', 40);
```

Фактическими аргументами являются символ '+' и число 40. Эти значения присваиваются соответствующим формальным аргументам в функции zvezdochka() – переменным symbol и count. Фактическим аргументом может быть константа, переменная или более сложное выражение. Независимо от того, что является фактическим аргументом, он вычисляется и его значение копируется в соответствующий формальный аргумент функции. Таким образом, фактический аргумент – это конкретное значение, присвоенное переменной, называемой формальным аргументом.

В предыдущем примере было показано, как передавать информацию из вызывающей функции в вызываемую функцию. Для передачи информации в противоположном направлении используется возвращаемое значение функции. Как правило, возвращаемое функцией значение имеет отношение к решению задачи, возложенной на эту функцию.

Создадим собственную функцию, которая находит модуль большего из своих двух аргументов. Листинг программы, использующей такую функцию, приведен ниже:

Листинг 2.4

```
#include <stdio.h>
#include <math.h>
double modmax(float, float);
int main(){
  float a, b;
  printf("Введите первое и второе число: ");
  scanf("%f %f", &a, &b);
  printf("Модуль наибольшего числа из чисел %f и %f равен
%f", a, b, modmax(a,b));
  return 0;
}
double modmax(float a, float b){
  float max;
  if (a>b)
    max=a;
  else
    max=b;
return fabs(max);
}
```

Если функция возвращает значение, тип этого значения должен быть определен. Тип возвращаемого значения должен быть указан перед именем функции при объявлении и определении функции. В нашем примере функция modmax() возвращает значение типа double, что отражено в объявлении:

```
double modmax(float, float);
```

Слово double означает, что функция modmax() возвращает значение типа double, а слова float в круглых скобках – то, что у функции modmax() имеются два аргумента типа float. В предыдущем примере функция zvezdochka() не возвращала значения, поэтому перед именем функции стояло ключевое слово void.

Ключевое слово return в вызываемой функции означает, что значение следующего выражения становится возвращаемым значением функции. В данном случае функция возвращает значение fabs(max). Так как функция fabs() возвращает значение типа double, то и функция modmax() также имеет тип double.

Если функция возвращает значение, то вызов функции рассматривается как выражение, значение которого равно величине, возвращаемой функцией. Это выражение можно использовать как и любое другое выражение, например, присвоить его значение переменной:

```
result=modmax(a,b);
```

В этом примере переменной result присваивается значение, возвращенное функцией modmax().

Количество аргументов у функции может быть сколь угодно большим, а возвращаемое значение всегда только одно. Эта особенность функций является препятствием только тогда, когда необходимо вернуть несколько значений. Существуют способы, позволяющие возвращать и несколько значений при помощи функций. Один из таких способов – *передача аргументов по ссылке*. Для этого используются указатели.

Всегда следует указывать тип значения, возвращаемого функцией. Если функция не возвращает значения, то в качестве типа возвращаемого значения должно стоять ключевое слово *void*. Если не будет указан тип функции, то по умолчанию будет использоваться тип *int*.

2.3 Функции и массивы

Предположим, что необходимо написать функцию, вычисляющую сумму элементов массива.

Рассмотрим программу, представленную в листинге 2.5.

Листинг 2.5

```
#include <stdio.h>
#define SIZE 5
int sum(int * , int);
int main(void){
  int summa;
  int mass[SIZE]={6, 18, 9, 43, 56};
  summa=sum(mass, SIZE);
  printf("Сумма элементов массива равна %d\n", summa);
  return  0;
}
int sum(int * s, int count){
  int i;
  int total=0;
  for (i=0; i<count; i++)
    total+=s[i];
  return total;
}
```

Для передачи функции массива необходимо указать имя массива без всяких скобок:

```
sum(mass, SIZE);
```

Оператор в предыдущей строке передает массив в функцию по ссылке – вызываемая функция может менять значения элементов в исходном массиве вызывающей функции.

В языке С все вызовы передают аргументы по значению. При этом имеется возможность имитировать передачу аргумента по ссылке, используя операцию взятия адреса и косвенные операции. Массивы автоматически

передаются посредством имитации передачи аргумента по ссылке. Название массива является адрес первого элемента этого массива, поэтому действительный аргумент mass, который является адресом для int, необходимо присвоить формальному параметру, то есть указателю на int:

```
int sum(int *, int); //соответствующий прототип
```

При вызове функции она получает адрес первого элемента массива, где показано, что на этом месте находится значение int. На основе этой информации нельзя сделать вывод о количестве элементов в массиве. Поэтому с помощью второго аргумента функции sum() мы кодируем количество элементов в массиве.

В контексте прототипа функции или заголовка для определения функции, и только в этом контексте, можно заменить ***int ** *** на ***int []***.

В определении функции эквивалентны две формы:

```
int sum(int * s, int count){
  //здесь находится код функции
}
```

и

```
int sum(int s[], int count){
  //здесь находится код функции
}
```

2.4 Защита содержимого массива

Для сохранения целостности данных в С обычно передаются данные с помощью передачи по значению. При этом создается копия исходных данных, и функция, работающая с этой копией, не может случайно обновить исходные данные. Однако поскольку функции, обрабатывающие массивы, работают с исходными данными, они могут обновить массив. Иногда это нежелательно.

Для предотвращения возможного изменения данных в исходном массиве необходимо воспользоваться при объявлении формального параметра в прототипе и в определении функции ключевым словом **const**. Например, прототип и определение для функции sum() должны иметь следующий вид:

```
int sum(const int * , int);
int sum(const int * s, int count){
  int i;
  int total=0;
  for (i=0; i<count; i++)
    total+=s[i];
  return total;
}
```

В этом случае компилятор будет знать о том, что функция воспринимает массив, на который указывает **s**, как массив, содержащий постоянные данные. Если вдруг происходит изменение исходного массива, компилятор будет генерировать сообщение об ошибке.

Здесь важно понимать, что наличие ключевого слова const не требует, чтобы исходный массив являлся постоянным, только функция воспринимает массив как постоянный. Если необходимо использовать функцию, которая может обновлять массив, при объявлении параметра массива не используйте const. Если необходимо использовать функцию, не предназначенную для обновления массива, воспользуйтесь const при объявлении массива.

2.5 Многофайловая компиляция программ

Простейший подход к использованию нескольких функций – помещение их в один файл. После этого необходимо скомпилировать этот файл так, как если бы он содержал единственную функцию. При создании серьезных программ осуществляется разбиение программы на функции, при этом большие функции описываются в отдельных файлах. При этом определение именованных констант, прототипы функций, директивы препроцессора, описание типов данных удобнее выносить в отдельный заголовочный файл, который подключается посредством директивы #include. Для компиляции таких программ используется многофайловая компиляция.

В качестве примера рассмотрим создание многофайловой программы, пріведенной ранее в первой главе.

Данная программа будет состоять из четырех файлов, листинг которых приведен ниже.

Л и с т и н г 2.6

my.h

```
#include <stdio.h>
#include <ctype.h>
#define SIZE    5
void vvod(void);
void vyvod(void);
struct book {
  char nazvanie[20];
  int page;
};
```

main.c

```
#include "my.h"
struct book a[SIZE];
int count;
int main(void){
```

```
    int punkt;
    int i;
    count=0;
    system("clear");
    while (1){
      printf("Выберите пункт меню: ");
      printf("1 - Ввод, 2 - Вывод, ");
      printf"0 - Выход из программы: ");
      scanf("%d", &punkt);
      while (getchar()!='\n')
        continue;
      switch (punkt) {
        case 1:
          vvod();
          break;
        case 2:
          vyvod();
          break;
        case 0:
          printf("Программа завершила свою работу."
                 "Пока!!!\n");
          return 0;
        default:
          printf("Пункт меню выбран неправильно. "
                  "Попробуйте еще раз!!!\n");
          break;
      }
    }
    return 0;
  }
```

vvod.c

```
  #include “my.h”
  void vvod(void){
    extern count;
    extern struct book a[SIZE];
    char choice;
    //Выбор создания новой таблицы или дозаписи в старую
    if (count!=0){
      while (1) {
        printf("Создать новую таблицу или дописывать"
               "в старую? (N- new, O - old): ");
        choice=toupper(getchar());
        if (choice=='N') {
          count=0;
        }
        if (choice!='N' && choice != 'O'){
          while (getchar()!='\n')
```

```
            continue;
          continue;
        }
        while (getchar()!='\n')
          continue;
        break;
      }
    }
    //Ввод таблицы
    for (; count<SIZE; count++){
      printf("Введите название %d книги: ", count+1);
      gets(a[count].nazvanie);
      printf("Введите количество страниц в книге: ");
      scanf("%d", &a[count].page);
      while (getchar()!='\n')
        continue;
      printf("Закончить ввод?: y/n ");
      if (toupper(getchar())=='Y') {
        count++;
        break;
      }
      else {
        while (getchar()!='\n')
          continue;
        continue;
      }
    }
    return;
  }
```

vyvod.c

```
  #include "my.h"
  void vyvod(void){
    extern int count;
    extern struct book a[SIZE];
    int i;
    for (i=0; i<count; i++)
      printf("%3d книга  -  %s %d\n", i+1, a[i].nazvanie,
             a[i].page);
    return;
  }
```

В качестве отдельного заголовочного файла применяется файл my.h. В качестве основного файла, содержащего функцию main() используется файл main.c. Функции ввода и вывода данных помещены в файлы vvod.c и vyvod.c соответственно. В качестве глобальных переменных используется имя массива структур **a** и счетчик введенных записей **count**, объявленные в файле main.c до функции main(). Для обеспечения доступа к этим перемен-

ным, в других файлах осуществляется ссылочное объявление этих же переменных с ключевым словом **extern**, указывающим компилятору на необходимость поиска определяющих объявлений в другом файле.

Компиляция программы с помощью компилятора gcc может быть выполнена следующим образом:

```
gcc –o proga main.c vvod.c vyvod.c
```

что является не самым эффективным способом компиляции многофайловых программ.

Для более эффективной компиляции (с точки зрения времени и загрузки процессора) необходимо к этой программе добавить файл makefile, содержимое которого представлено в листинге 2.7.

Листинг 2.7

```
result: main.o vvod.o vyvod.o
        gcc -o result main.o vvod.o vyvod.o
main.o: main.c my.h
        gcc -c main.c
vvod.o: vvod.c my.h
        gcc -c vvod.c
vyvod.o: vyvod.c my.h
        gcc -c vyvod.c
clean:
        rm –fr *.o result
```

после чего для компиляции необходимо запускать программу make, которая будет компилировать лишь те файлы, в которые были внесены изменения с момента последней компиляции.

Более подробно о работе утилиты make и правилах составления makeфайла можно прочитать в приложении А.

В качестве примера рассмотрим создание многофайловой программы СПРАВОЧНИК ***с использованием указателей и связного списка.***

Данная программа будет состоять из четырех файлов, листинг которых приведен ниже.

Листинг 2.8

my.h

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
struct book {
  char nazvanie[30];
  int page;
  struct book *next;
};
struct book * vvod(struct book *);
```

main.c

```
   #include "my.h"
   int main(void){
     system("clear");
     struct book *head=NULL;
     int punkt;
     while (1){
       printf("Выберите пункт меню: ");
       printf("1 - Ввод, 2 - Вывод, 0 - Выход из программы:
");
       scanf("%d", &punkt);
       while (getchar()!='\n')
         continue;
       printf("%p\n", head);
       switch (punkt) {
         case 1:
           head=vvod(head);
           break;
         case 2:
           vyvod(head);
           break;
         case 0:
           printf("Программа завершила свою работу. По-
ка!!!\n");
           return 0;
         default:
           printf("Пункт меню выбран неправильно. Попробуйте
еще раз!!!\n");
           break;
       }
     }
     return 0;
   }
```

vvod.c

```
   #include "my.h"
   struct book * vvod(struct book *head){
     char choice;
     int i, count=0;
     struct book *prev, *current;
     if (head!=NULL){
       while (1) {
         printf("Создать новую таблицу или дописывать "
                "в старую? (N- new, O - old): ");
         choice=toupper(getchar());
         if (choice=='N') {
           count=0;
           //использовать free() для очистки памяти
```

```
            current=head;
            printf("head=%p\n",head);
            while(current != NULL){
              prev=current->next;
              free(current);
              current=prev;
            }
            //-------------------------------------------
            head=NULL;
          }
          if (choice=='O') {
            count=0;
            current=head;
            while(current != NULL){
              prev=current;
              current=current->next;
              count ++;
            }
          }
          if (choice!='N' && choice != 'O'){
            while (getchar()!='\n')
              continue;
            continue;
          }
          while (getchar()!='\n')
            continue;
          break;
        }
      }
      for ( ; ; count++){
        current=(struct book *) malloc(sizeof(struct book));
//выделяем место для новой записи
        if (head==NULL)
          head=current;
        else
          prev->next=current;
        current->next=NULL;
        printf("Введите название %d книги: ", count+1);
        gets(current->nazvanie);
        printf("Введите количество страниц в книге: ");
        scanf("%d", &current->page);
        while (getchar()!='\n')
          continue;
        prev=current;
        printf("Закончить ввод?: y/n ");
        if (toupper(getchar())=='Y') {
          count++;
```

```
        break;
      }
      else {
        while (getchar()!='\n')
          continue;
        continue;
      }
    }
    return head;

  }
```

vyvod.c

```
  #include "my.h"
  void vyvod(struct book *head){
    int i, count=0;
    struct book *current;
    while (1){
      if (head==NULL)
        printf("Данные не введены!\n");
      else
        printf("Список книг:\n");
      current=head;
      i=0;
      while (current!=NULL){
        printf("%d книга  -  %s %d\n", ++i,
                 current->nazvanie, current->page);
        current=current->next;
      }
      break;
    }
    return;
  }
```

3 СОРТИРОВКИ

3.1 Общие сведения

Сортировка – это упорядочивание набора однотипных данных по возрастанию или убыванию.

После сортировки по возрастанию в начале массива должен располагаться минимальный элемент, каждый последующий элемент должен быть не больше предыдущего, в конце массива должен располагаться максимальный элемент. Аналогично после сортировки по убыванию: в начале массива – максимальный элемент, каждый последующий элемент – не меньше предыдущего, в конце массива – минимальный элемент.

На рисунке 3.1 приведены примеры числовых массивов: неотсортированный, отсортированные по возрастанию и убыванию.

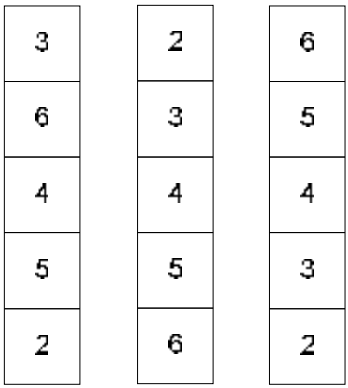

Рисунок 3.1 – Числовые массивы

Для массива с символами сортировка происходит по коду символа. Например, буква 'a' в кодировке ASCII будет иметь код 97, а буква 'b' – код 98, следовательно, значение символа 'a' будет меньше, чем значение символа 'b'.

В большинстве кодировок соблюдается следующий принцип: наименьший код имеет заглавная первая буква алфавита, далее идут заглавные буквы по алфавиту, с соответствующим увеличением кода, далее – заглавная последняя буква алфавита, после этого соответствующим образом идут строчные буквы, и, следовательно, самый большой код будет иметь строчная последняя буква алфавита.

Для того чтобы сортировка символьного массива происходила по алфавиту, необходимо отсортировать данный массив по возрастанию.

На рисунке 3.2 приведены примеры символьных массивов в кодировке ASCII: неотсортированный, отсортированные по возрастанию и по убыванию.

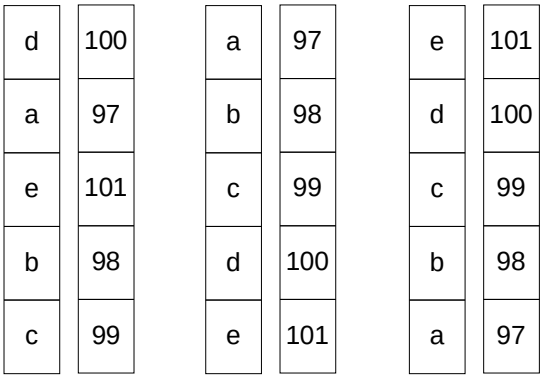

Рисунок 3.2 – Символьные массивы

Сортировка строк происходит по первой букве слова, если же первые буквы совпадают, то по второй, если и вторые буквы совпадают, то по третьей и т. д. На рисунке 3.3 показаны массивы строк: неотсортированный, отсортированный по возрастанию, отсортированный по убыванию. Для сортировки слов рекомендуется пользоваться функцией strcmp.

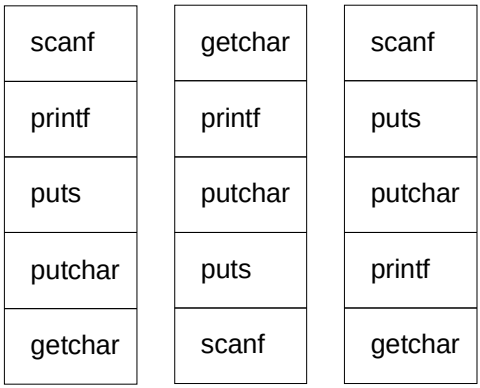

Рисунок 3.3 – Массивы строк

Существуют различные способы сортировок. К простым сортировкам причисляют следующие:

- пузырьковая,
- методом выбора,
- методом вставки.

3.2 Пузырьковая сортировка

Данный метод считается простейшим. Однако он эффективен лишь для небольших массивов. Смысл пузырьковой сортировки заключается в обмене двух соседних элементов.

При сортировке пузырьком выполняются несколько проходов по массиву. При этом на каждом проходе последовательно сравниваются каждые два соседних элемента. Если порядок неверный, то элементы меняются местами.

Листинг 3.1

```
void puzyrok(int *mass, int n)
{
  int i, j;
  int c;
  for (i = 1; i < n; i++) {
    for (j = 0; j < n-1; j++) {
      if (mass[j] > mass[j + 1]){
```

```
            c = mass[j];
            mass[j] = mass[j+1];
            mass[j+1] = c;
          }
        }
      }
    }
```

На рисунке 3.4 показан ход пузырьковой сортировки массива, состоящего из 5 элементов.

Сортировка методом пузырька неэффективна за счет многочисленных обменов между соседними элементами.

3.3 Сортировка методом выбора

Сортировка методом выбора позволяет обойтись без лишних обменов между соседними элементами. Принцип данной сортировки заключается в нахождении на каждом проходе максимального (минимального) элемента в неотсортированной части массива и установки его в нужную позицию. При этом размер неотсортированной части массива уменьшается на единицу. После n-1 проходов массив будет отсортирован.

Принцип сортировки показан на рисунке 3.5.

Первоначально весь массив неотсортирован. В массиве находится максимальный элемент (показан жирным шрифтом) который меняется с элементом, находящимся на последней позиции. В результатате максимальный элемент будет находиться на нужной позиции. На следующем шаге рассматриваем только неотсортированную часть массива (т. е. все элементы, кроме последнего). В этой части массива находим максимальное значение и ставим на последнее место в этой части.

Ниже показан листинг функции сортировки выбором.

Листинг 3.2

```
void vybor(int *mass, int n)
{
    int max, i, temp;

    for(;n > 1; n--) {
        max = 0;

        for(i = 1; i < n; i++)
            if (mass[i] > mass[max])
                max = i;

        temp = mass[n-1];
        mass[n-1] = mass[max];
        mass[max] = temp;
    }
 }
```

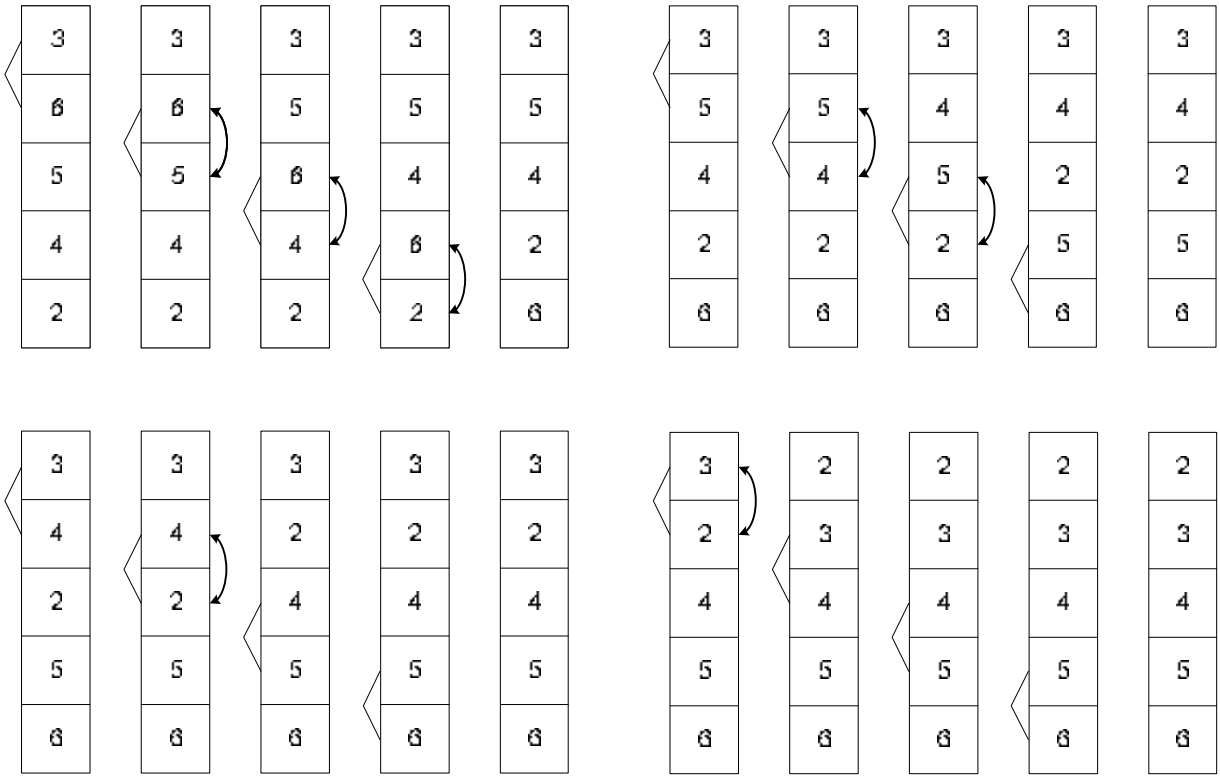

Рисунок 3.4 – Пузырьковая сортировка

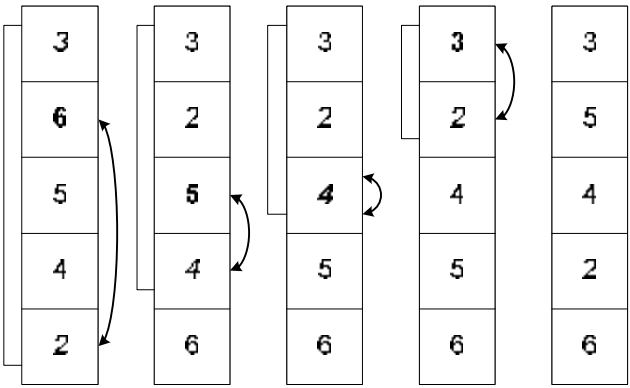

Рисунок 3.5 – Сортировка методом выбора

В функцию передаются значения указателя на начало массива mass и количества элементов массива n. Внешний цикл for определяет число проходов по массиву

3.4 Сортировка методом вставок

В данном методе сначала проверяется соответствие порядку первых двух элементов массива. Если порядок неправильный, то элементы переставляются. Таким образом, получаем отсортированную часть массива, состоящую из 2 элементов. Далее берем 3-й элемент и, последовательно сравнивая с предыдущими элементами, определяем в какую позицию необходимо его поставить, чтобы не нарушить порядок. Соответственно, элементы, имеющие индекс больше нового индекса данного элемента (т. е. являющиеся больше, чем данный элемент), сдвигаем на один элемент в сторону увеличения индекса. В итоге получаем отсортированную часть массива, состоящую из 3 элементов. Берем 4-й элемент и, аналогично, ищем место для вставки и т. д.

Функция с использованием данного метода показана в листинге 3.3.

Листинг 3.3

```
void vstavka(int *mass, int n) {
    int i, j, value;

    for(i = 1; i < n; i++) {
        value = mass[i];
        for (j = i - 1; j >= 0 && mass[j] > value; j--) {
            a[j + 1] = a[j];
        }
```

```
            a[j + 1] = value;
        }
    }
```

Принцип сортировки показан на рисунке 3.6.

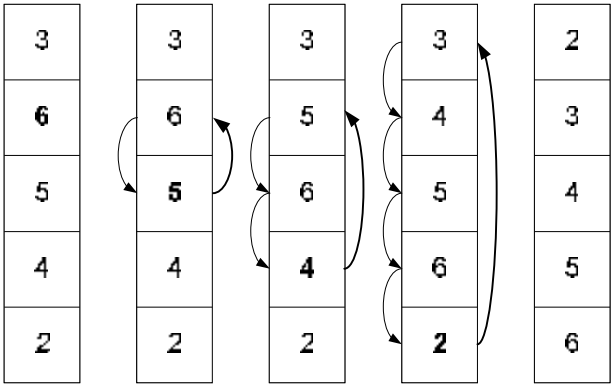

Рисунок 3.6 – Сортировка методом вставок

Вернемся к программе, написанной в предыдущей главе.

Рассмотрим функцию, которая сортирует введенные книги по алфавиту на примере метода пузырька.

Л и с т и н г 3.4

```
#include “my.h”
void sort_puz(void)
{
  extern count;
  extern struct book a[SIZE];
  struct book b;
  int i,j;
  for (i = 1; i <= count; i++) {
    for (j = 0; j < count-1; j++) {
      if (strcmp(a[j].nazvanie, a[j+1].nazvanie) > 0){
        b = a[j];
        a[j] = a[j+1];
        a[j+1] = b;
      }
    }
  }
}
```

В данной программе используется функция сравнения строк strcmp(), т. к. необходимо сравнить строковые значения. И хотя сравнение происхо-

дит по строковому полю nazvanie, в случае несоответствия порядка меняются между собой все поля сравниваемых элементов массива-структуры a посредством временной переменной-структуры b.

Рассмотрим на примере функции для многофайловой программы СПРАВОЧНИК второй способ сортировки пузырьком. Проведем сортировку по полю название.

Если в предыдущем варианте массив просматривался count раз, в данной функции массив будет просматриваться до тех пор, пока за проход будет происходить хотя бы одна перестановка (листинг 3.5).

Листинг 3.5

```
#include “my.h”
void sort_puz(void)
{
  extern count;
  extern struct book a[SIZE];
  struct book b;
  int i;
  char k;
  do {
    k=0;
    for (j = 0; j < count-1; j++) {
      if (strcmp(a[j].nazvanie, a[j+1].nazvanie) > 0){
        b = a[j];
        a[j] = a[j+1];
        a[j+1] = b;
        k=1;
      }
    }
  }while (k);
}
```

Программа сортировки пузырьком с использованием указателей показана в листинге 3.6.

Листинг 3.6

```
#include "my.h"
struct book* sort_puz(struct book *head)
{
  struct book *temp, *current, *prev;
  char k,d;
  if (head == NULL){
      printf("Данные не введены\n");
      return NULL;
  }
  do {
    k=0;
```

```
        d=1;
        current =head;
        while (current->next != NULL) {
            if (current->page > current->next->page){
          temp = current;
          current = current->next;
          temp->next = current->next;
          current->next = temp;
          if (d) {
              head = current;
          }
          else {
              prev->next = current;
          }
            k=1;
            }
            if (d) d=0;
            prev = current;
            current = current->next;
          }
      }while (k);
      printf("Сортировка завершена\n");
      return head;
    }
```

Расширим нашу многофайловую программу СПРАВОЧНИК, добавив функцию сортировки книг по возрастанию количества страниц методом пузырька. Не забудем так же внести изменения в файл main.c, my.h и Makefile. Измененные файлы программы представлена в листинге 3.7. Файлы vvod.c и vyvod.c оставлены без изменений и в листинге 3.7 не приводятся.

Листинг 3.7

my.h:

```
#include <stdio.h>
#include <ctype.h>
#define SIZE    5
void vvod(void);
void vyvod(void);
void sortpuz(void);
struct book {
    char nazvanie[20];
    int page;
};
```

main.c:

```
#include "my.h"
struct book a[SIZE];
int count;
```

```
int main(void){
    int punkt;
    int i;
    count=0;
    system("clear");
    while (1){
        printf("Выберите пункт меню: ");
        printf("1 - Ввод, 2 - Вывод, 3 - Сортировка по
возр. кол-ва страниц, 0 - Выход из программы: ");
        scanf("%d", &punkt);
        while (getchar()!='\n')
            continue;
        switch (punkt) {
            case 1:
                vvod();
                break;
            case 2:
                vyvod();
                break;
            case 3:
                sortpuz();
                break;
            case 0:
                printf("Программа завершила свою работу.
Пока!!!\n");
                return 0;
            default:
                printf("Пункт меню выбран неправильно.
Попробуйте еще раз!!!\n");
                break;
        }
    }
    return 0;
}
```

sort.c:

```
#include "my.h"
void sortpuz(void){
    extern int count;
    extern struct book a[SIZE];
    struct book t; //переменная, предназнач. для времен-
ного хранения
    //данных при обмене между ячейками
    int i,j;
    for (i=0; i<count; i++)
        for (j=0; j<count-1; j++)
            if (a[j].page>a[j+1].page){
                t=a[j];
```

```
                    a[j]=a[j+1];
                    a[j+1]=t;
                }
        printf("Сортировка завершена\n");
        return;
    }
```

Makefile:

```
result: main.o vvod.o vyvod.o sortpuz.o
        gcc -o result main.o vvod.o vyvod.o sortpuz.o
main.o: main.c my.h
        gcc -c main.c
vvod.o: vvod.c my.h
        gcc -c vvod.c
vyvod.o: vyvod.c my.h
        gcc -c vyvod.c
sortpuz.o: sortpuz.c my.h
        gcc -c sortpuz.c
clean:
        rm -rf *.o result
```

4 РАБОТА С ФАЙЛАМИ В ЯЗЫКЕ С

4.1 Аргументы командной строки

Командная строка – это строка, в которой осуществляется ввод для запуска программы. Аргументы командной строки – это дополнительные элементы в этой же строке. Например, при вводе команды

```
cp /etc/shells /home/user1/shells
```

cp является именем программы, а /etc/shells и /home/user1/shells – аргументами.

Программа на языке С может считывать эти аргументы для собственного использования.

При этом программа использует аргументы функции main().

Компиляторы языка С допускают, чтобы функция main() не имела ни одного аргумента либо имела два аргумента. Первый аргумент функции main() – количество аргументов командной строки при запуске программы (по традиции – argc), при этом система для разделения аргументов использует пробелы. Второй аргумент функции main() – массив указателей на строки. Каждый аргумент командной строки сохраняется в памяти и имеет присвоенный указатель, указывающий на него. По соглашению этот массив указателей называется argv. Когда это возможно, элементу argv[0] присваивается имя программы. Затем элементу argv[1] присваивается первый аргумент, элементу argv[2] – второй и т. д.

Во многих средах допускается использование кавычек для объединения нескольких слов в один аргумент. Например, команда

```
echo “Hello, world” yes
```

присвоила бы строку “Hello, world” элементу argv[1], а строку yes – элементу argv[2].

Рассмотрим программу, представленную в листинге 4.1. Она печатает на экране фразу “Hello, world!” столько раз, сколько указано в первом аргументе командной строки при вызове программы.

Л и с т и н г 4.1

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    int i;
    if (argc!=2){
        printf("Формат запуска программы:\n \"%s n\"\n
где n - целое число\n", argv[0]);
        exit (1);
    }
    for (i=0; i<atoi(argv[1]); i++)
        printf("Hello, world!\n");
    return 0;
}
```

При вызове программы без аргументов:

./programma

на экран будет выведено сообщение:

Формат запуска программы:

"./programma n"

где n - целое число

При вызове программы с аргументом – целым числом, например:

./programma 5

на экран будут выведены 5 строк фразы “Hello, world!”:

Hello, world!
Hello, world!
Hello, world!
Hello, world!
Hello, world!

Аргументы командной строки считываются в виде строк и для использования численного значения вначале строка должна быть преобразована в число. Если число является целым, то можно использовать функцию atoi(), которая принимает строку в качестве аргумента и возвращает соответствующее целочисленное значение. Функция atoi() прототипируется в файле stdlib.h. При вызове программы с аргументом, равным 5, аргумент будет сохранен в argv[1] в виде строки “5\0”. Вызов функции atoi(argv[1]) преобразует эту строку в число 5.

Если запустить эту программу без аргумента командной строки, проверка условия argc!=2 приведет к выводу сообщения о правильном исполь-

зовании программы и прерыванию программы с помощью функции exit(). Функция exit() возвращает в операционную систему целое число, завершая работу программы. Для ее использования необходимо подключать заголовочный файл stdlib.h.

4.2 Стандартный ввод/вывод в файлы

Часто необходимы программы, которые могут читать информацию из файлов или записывать результат в файл. Язык С предоставляет мощные методы для работы с файлами, с помощью которых можно открыть файл, а затем использовать специальные функции ввода/вывода для осуществления операций чтения/записи для данного файла.

Файл – это именованный раздел для сохранения информации, обычно раздел на диске. Язык С рассматривает файл как последовательность байтов, каждый из которых считывается в индивидуальном порядке. Это отвечает структуре файла в среде Unix, откуда происходит язык С. Так как другие среды могут не отвечать именно такой модели, стандарт языка С обеспечивает два способа представления файлов. Двумя способами представления файлов являются *бинарное(двоичное)* и *текстовое*. В бинарном представлении каждый байт файла доступен программе. В текстовом представлении то, что видит программа, может отличаться от того, что находится в файле. В системах Unix используется одна структура файла, поэтому оба представления при использовании в Unix будут идентичными.

Также можно выбрать один из двух уровней ввода/вывода, т. е. уровней доступа к файлу. При низкоуровневом вводе/выводе используются базовые функции ввода/вывода, которые поддерживаются операционной системой. При стандартном высокоуровневом вводе/выводе используется пакет библиотек функций на языке С и заголовочный файл `stdio.h`. Стандартом языка С гарантируется переносимость только высокоуровневой модели ввода/вывода, поэтому в дальнейшем будет рассматриваться только эта модель. Пакет стандартного(высокоуровневого) ввода/вывода обеспечивает два преимущества. Первое состоит в том, что он имеет много специализированных функций, упрощающих выполнение различных задач ввода/вывода. Второе – ввод и вывод буферизуется. Это значит, что информация передается большими блоками, что значительно повышает скорость обмена данными. Буферизация осуществляется в фоновом режиме, поэтому создается иллюзия посимвольного ввода/вывода.

При обсуждении стандартной библиотеки ввода/вывода мы будем отталкиваться от термина *поток*. Открыв или создав файл средствами стандартной библиотеки ввода/вывода, мы говорим, что связали поток с файлом.

Рассмотрим программу, выводящую на экран содержимое текстового файла и подсчитывающую количество символов в нем. Программа представлена в листинге 4.2.

Листинг 4.2

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]){
    int count;
    int ch;
    FILE *fp; //файловый указатель
    if (argc!=2){
        printf("Формат запуска программы:\n \"%s file\"\n
где file - имя выводимого на экран файла\n", argv[0]);
        exit (1);
    }
    if ((fp=fopen(argv[1], "r"))==NULL){
        printf("Невозможно открыть файл %s\n", argv[1]);
        exit (2);
    }
    count=0;
    while ((ch=getc(fp))!=EOF){
        putc(ch, stdout);
        count++;
    }
    fclose(fp);
    printf("Файл %s имеет %d символов\n", argv[1],
count);
    return 0;
}
```

При открытии потока стандартная функция fopen() возвращает указатель на объект FILE. Объект FILE является структурой, содержащей всю информацию, необходимую для управления потоком средствами стандартной библиотеки ввода/вывода: дескриптор файла, размер буфера, флаг ошибки и т. д. Прикладные программы никогда не работают с объектом FILE напрямую. Для ссылки на поток необходимо просто передать указатель на объект FILE в виде аргумента любой стандартной функции ввода/вывода. Мы будем называть указатель на объект FILE, тип FILE * *указателем на файл(файловый указатель).* В программе объявляется файловый указатель fd:

```
FILE *fp; //файловый указатель
```

Функция fopen(), объявленная в заголовочном файле stdio.h имеет два аргумента. Первый аргумент – адрес строки, содержащей имя открываемого файла. Второй аргумент – строка, определяющая режим открытия файла. В нашем случае файл открывается программой только для чтения. Строки, задающие режим открытия файла, представлены в таблице 4.1.

Для таких систем, как Unix (Linux, FreeBSD, Solaris, MacOS X), имеющих только один тип файла, режимы, в обозначении которых есть буква *'b'*, эквивалентны режимам, в обозначении которых эта буква не применяется.

Т а б л и ц а 4.1 – Режимы открытия файлов

Строка режима	Значение
“r”	Открыть текстовый файл для чтения
“w”	Открыть текстовый файл для записи, обнулив существующий файл или создав новый файл в случае его отсутствия
“a”	Открыть текстовый файл для записи, добавив запись к концу существующего файла или создав новый файл в случае его отсутствия
“r+”	Открыть текстовый файл для обновления, т. е. и для чтения, и для записи
“w+”	Открыть текстовый файл для обновления (чтения и записи), сначала обнулив существующий файл или создав новый файл в случае его отсутствия
“a+”	Открыть текстовый файл для обновления (чтения и записи), сначала обнулив существующий файл или создав новый файл в случае его отсутствия; чтение возможно для всего файла, а запись возможна только в конец файла
“rb”, “wb”, “ab”, “ab+”, “a+b”, “wb+”, w+b”, “a+b”, “ab+”	Эти режимы действуют так же, как и предыдущие, только вместо текстового режима доступа к файлу используется двоичный режим доступа

После успешного открытия файла функция fopen() возвратит файловый указатель, применяемый другими функциями ввода/вывода для осуществления ссылок. Функция fopen() возвращает нулевой указатель (определенный в файле stdio.h), если она не может открыть файл. Программа прекращает работу, если fp равен NULL. Такое может произойти, если переполнен диск, отсутствуют соответствующие права на открываемый файл, используется некорректное имя файла, превышены дисковые квоты и т. д. Поэтому необходимо производить обработку ошибок при осуществлении файлового ввода/вывода.

В программе встречаются функции getc() и putc(). Они работают почти так же, как getchar() и putchar(). Отличие состоит в том, что при использовании этих функций необходимо указать, какой файл необходимо использовать.

```
ch=getc(fp);
```

Данное выражение считывает символ из файла, на который указывает файловый указатель fp.

```
putc(ch, stdout);
```

А этот оператор отправляет символ ch в файл, идентифицируемый файловым указателем stdout, автоматически ассоциированным со стандартным потоком вывода, т. е. просто печатает символ на экране.

Напомним, что программы на языке С автоматически открывают 3 файла. Они называются стандартный поток ввода (*stdin*), стандартный поток вывода (*stdout*), стандартный поток вывода ошибок (*stderr*). Стандартный

поток ввода присоединяется, как правило, к клавиатуре. Стандартный поток вывода и стандартный поток вывода ошибок присоединяются по умолчанию к обычному устройству вывода, как правило, монитору. Тип перечисленных параметров (stdin, stdout, stderr) соответствует указателю на FILE, поэтому их можно использовать как аргументы стандартных функций ввода/вывода (так же, как и указатель fp в примере).

При считывании данных из файла программа должна останавливаться по достижении конца файла. Если при попытке считать символ функция getc() встречает символ конца файла, то возвращается специальное значение EOF. Программа на языке С обнаруживает конец файла после того, как пытается считать символ, находящийся за признаком конца файла.

Функция fclose() закрывает файл, идентифицируемый указателем fp. При этом очищаются буферы. Желательно проверять, был ли файл успешно закрыт. Функция fclose() возвращает число 0 при успешном закрытии файла или EOF – в остальных случаях. Программный код для проверки успешности закрытия файла может иметь следующий вид:

```
if (fclose(fp)!=0)
  printf("Ошибка при закрытии файла %s\n", argv[1]);
```

Функции ввода/вывода файлов fscanf() и fprintf() функционируют почти так же, как printf() и scanf(). Отличие заключается только в применении дополнительного первого аргумента, определяющего файл. Эти функции, а также функция rewind() применяются в программе, представленной в листинге 4.3

Листинг 4.3

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
    int i;
    float t;
    FILE *fp;
    if ((fp=fopen("datafile", "w+"))==NULL){
        fprintf(stdout, "Невозможно открыть файл
\"datafile\"\n");
        exit(1);
    }
    srand(time(NULL));
    for (i=0; i<20; i++){
        t= (float) rand()/RAND_MAX * 100;
        fprintf(fp, "%8.3f", t);
    }
    rewind(fp);
    puts("Содержимое файла \"datafile\"");
    for (;;){
        if (fscanf(fp, "%f", &t)!=EOF)
```

```
            fprintf(stdout, "%8.3f", t);
        else
            break;
    }
    putchar('\n');
    if (fclose(fp)!=0){
        fprintf(stderr, "Ошибка закрытия файла\n");
    }
    return 0;
}
```

Эта программа позволяет генерировать псевдослучайные вещественные числа, лежащие в диапазоне от 0 до 100 и записывать их в файл. Используя режим “w+”, программа может считывать информацию из файла, а также осуществлять запись в файл. При первом использовании программа создает файл с числами, а при дальнейшем использовании – сначала просто обнуляет файл, а затем снова записывает числа. Функция *rewind()* осуществляет возврат к началу файла так, чтобы заключительный цикл мог выводить содержимое файла на экран. Функция *rewind()* использует файловый указатель в качестве аргумента. При достижении конца файла функция fscanf() возвращает значение EOF, и программа выходит из бесконечного цикла с помощью оператора break. В конце программы осуществляется проверка закрытия файла.

Для осуществления файлового ввода/вывода можно также использовать функции fgets() и fputs(). Функция fgets() имеет три аргумента. Первый – адрес (тип char *), по которому необходимо сохранить результаты ввода. Второй аргумент – целое число, представляющее собой максимальный размер вводимой строки. Последний третий аргумент – файловый указатель, идентифицирующий считываемый файл. Вызов функции имеет следующий вид:

```
fgets(name, MAX, fp);
```

Здесь name – имя массива типа char, MAX – максимальный размер строки, а fp – указатель на FILE.

Функция fgets() считывает информацию из входного потока до тех пор, пока не встречает символ новой строки или пока считано менее, чем MAX-1 символов или же пока не достигнут конец файла. Затем функция fgets() добавляет конечный нулевой символ с целью формирования строки. Если fgets() считывает целую строку до того, как достигнут предел, в конце строки добавляется символ новой строки, как раз перед нулевым символом. В этом она отличается от функции gets(), которая считывает символ новой строки, но опускает его.

Как и функция gets(), fgets() возвращает значение NULL в случае, если встречается символ EOF. Это можно использовать для проверки того, достигнут ли конец файла.

Функция fputs() имеет два аргумента: адрес строки и файловый указатель. Она записывает строку, найденную по указателю в файл. В отличие от puts(), функция fputs() не добавляет символ новой строки при выводе. Вызов функции выглядит следующим образом:

```
fputs(name, fp);
```

Здесь name – адрес строки, а fp идентифицирует файл.

В связи с тем, что fgets() сохраняет символ новой строки, а fputs() не добавляет его, эти функции хорошо работают в тандеме.

В связи с тем, что функцию fgets() можно использовать для предотвращения переполнения массива, она является более удобной, чем gets() при выполнении серьезного программирования.

4.3 Двоичный ввод/вывод в языке С

Стандартные функции fprintf() и fscanf() ориентированы на работу с текстовыми файлами – они имеют дело с символами и строками. Функция fprintf() конвертирует числовые значения в строковые, изменяя их при этом для некоторых типов данных. В дальнейшем при считывании этого числа из файла будет невозможно вернуться к исходной точности. Поэтому наиболее точный и последовательный способ записи числа – сохранение его в таком же виде, как это делает программа. Например, значение типа double необходимо сохранять в контейнере, имеющем размер double. При этом преобразование числовой формы данных в строковую не происходит, и в таком случае говорят, что данные сохраняются в двоичной форме. В случае стандартного потока ввода/вывода эту работу выполняют функции fread() и fwrite().

Функция fwrite() записывает двоичные данные в файл и имеет следующий формат:

```
size_t fwrite(void *ptr, size_t size, size_t n, FILE *fp)
```

где size_t – тип, определенный с помощью стандартных типов языка С. Этот тип возвращается при выполнении операции sizeof (обычно это unsigned int), ptr – адрес блока данных, которые надо записать. Переменная size содержит размер блоков данных (в байтах), которые будут записаны, а n – количество таких блоков. Указатель fp идентифицирует файл.

При этом функция *fwrite()* возвращает количество успешно записанных блоков. Обычно это количество равно значению переменной n, но может быть и меньше, если при записи произошли какие-либо ошибки.

Функция *fread()* используется для чтения данных, записанных в файл с помощью функции *fwrite()* и имеет следующий формат:

```
size_t fread(void *ptr, size_t size, size_t n, FILE *fp)
```

где ptr – адрес в памяти, куда передаются считанные данные, а fp идентифицирует файл, из которого считываются данные.

При этом функция fread() возвращает количество успешно считанных блоков данных. Обычно это количество равно значению переменной n, но может быть и меньше, если при чтении произошли какие-либо ошибки или достигнут конец файла.

В листингах 4.4 и 4.5 представлены программы осуществляющие запись в файл и чтение из файла бинарных данных.

Листинг 4.4

```
#include <stdio.h>
#include <stdlib.h>
#define KOL 10
int main(){
    int i, res;
    FILE *out;
    float a[KOL];
    srand(time(NULL));
    for (i=0; i<KOL; i++)
        a[i]=rand()/2147483647.0;
    if((out=fopen("outdata", "wb"))==NULL){
        fprintf(stderr, "Невозможно открыть файл для за-
писи\n");
        exit(1);
    }
    res=fwrite(a, sizeof(float),KOL,out);
    if(res!=KOL){ //проверка правильности записи
        printf("Не все блоки записаны\n");
        exit(2);
    }
    fclose(out);
    return 0;
}
```

Листинг 4.5

```
#include <stdio.h>
#include <stdlib.h>
#define KOL 10
int main(){
    int i, res;
    FILE *in;
    char filename[20];
    float a[KOL];
    puts("Введите имя файла для открытия: ");
    gets(filename);
    if((in=fopen(filename, "rb"))==NULL){
        fprintf(stderr, "Невозможно открыть файл %s для
чтения\n", filename);
        exit(1);
```

```
        }
        res=fread(a, sizeof(float),KOL,in);
        printf("Было считано %d блоков данных\n", res);
        for (i=0; i<res; i++)
            printf("%f ", a[i]);
        putchar('\n');
        fclose(in);
        return 0;
    }
```

Расширим все ту же нашу многофайловую программу СПРАВОЧНИК, добавив функции сохранения справочника в файл и считывания из файла. Необходимо также внести изменения в файл main.c, my.h и Makefile. Измененные и новые файлы read.c и write.c представлены в листинге **, остальные файлы повторяются и здесь не приводятся.

Листинг 4.6

my.h:

```
    #include <stdio.h>
    #include <ctype.h>
    #include <stdlib.h>
    #define SIZE    5
    void vvod(void);
    void vyvod(void);
    void sortpuz(void);
    void write(void);
    void read(void);
    struct book {
        char nazvanie[20];
        int page;
    };
```

main.c:

```
    #include "my.h"
    struct book a[SIZE];
    int count;
    int main(void){
        int punkt;
        int i;
        count=0;
        system("clear");
        while (1){
            printf("Выберите пункт меню: ");
            printf("1 - Ввод, 2 - Вывод, 3 - Сортировка по
возр. кол-ва страниц,\n");
            printf("4 - Запись в файл, 5 -Чтение из файла, 0
- Выход из программы: ");
            scanf("%d", &punkt);
            while (getchar()!='\n')
```

```
                continue;
            switch (punkt) {
                case 1:
                    vvod();
                    break;
                case 2:
                    vyvod();
                    break;
                case 3:
                    sortpuz();
                    break;
                case 4:
                    write();
                    break;
                case 5:
                    read();
                    break;
                case 0:
                    printf("Программа завершила свою работу.
Пока!!!\n");
                    return 0;
                default:
                    printf("Пункт меню выбран неправильно.
Попробуйте еще раз!!!\n");
                    break;
            }
        }
        return 0;
    }
```

write.c:

```
    #include "my.h"
    void write(void){
        extern int count;
        extern struct book a[SIZE];
        int res; //количество записанных блоков данных
        FILE *out;
        if((out=fopen("database", "wb"))==NULL){
            printf("Невозможно открыть файл для записи\n");
            exit(1);
        }
        res=fwrite(a,sizeof(struct book), count, out);
        if (res!=count){
            fprintf(stderr, "Не все записи были записаны в
файл\n");
            exit(2);
        }
        fclose(out);
```

```
        printf("Запись в файл успешно завершена\n");
        return;
    }
```

read.c:

```
    #include "my.h"
    void read(void){
        extern int count;
        extern struct book a[SIZE];
        FILE *in;
        if((in=fopen("database", "rb"))==NULL){
            printf("Невозможно открыть файл для чтения\n");
            exit(1);
        }
        count=fread(a,sizeof(struct book), SIZE, in);
        fclose(in);
        printf("Было считано %d записей\n", count);
        return;
    }
```

Makefile:

```
    result: main.o vvod.o vyvod.o sortpuz.o write.o read.o
            gcc -o result main.o vvod.o vyvod.o sortpuz.o
write.o read.o
    main.o: main.c my.h
            gcc -c main.c
    vvod.o: vvod.c my.h
            gcc -c vvod.c
    vyvod.o: vyvod.c my.h
            gcc -c vyvod.c
    sortpuz.o: sortpuz.c my.h
            gcc -c sortpuz.c
    write.o: write.c my.h
            gcc -c write.c
    read.o: read.c my.h
            gcc -c read.c
    clean:
            rm -rf *.o result
```

5 ВВОД/ВЫВОД В ЯЗЫКЕ C++

5.1 Общие сведения

Язык C++ представляет собой объектно-ориентированный язык, который унаследовал возможности языка C. C++ является расширением языка C, при этом любая конструкция, написанная на языке C является корректной в языке C++. В то же время обратное является неверным. Поэтому все

ранее приобретенные знания о языке С будут так же актуальны и при изучении языка С. Наиболее важное отличие заключается в том, что в С++ используется объектно-ориентированная парадигма программирования. Также имеются усовершенствования при организации ввода/вывода.

Наша задача состоит в ознакомлении с возможностями языка С++ при организации ввода с клавиатуры и вывода на экран.

Наша первая программа на языке С++ представлена в листинге 5.1.

Листинг 5.1

```
#include <iostream>
using namespace std;
int main(void){
  cout<<”Hello, world!\n”;
  return 0;
}
```

Эта программа выводит на экран фразу “Hello, world”.

Каждая программа на языке С++ может быть разбита на несколько так называемых *пространств имен*. *Пространство имен* – это область программы, в которой распознается определенная совокупность имен. Директива using namespace std; означает, что все определенные ниже имена в программе будут относиться к пространству имен с именем std.

Идентификатор cout является объектом языка С++, предназначенным для работы со *стандартным потоком вывода*. Стандартный поток вывода обычно направлен на экран, хотя допускается его перенаправление на другие устройства вывода. Операция << называется *операцией вставки в поток*. Она копирует содержимое переменной, стоящей в правой части, в объект, стоящий в левой части. В нашей программе она копирует строку “Hello, world!\n” в объект cout, который и выводит эту строку на экран. В языке С операция << известна как операция *побитового сдвига влево*. В языке С++ операции могут быть перегружены, т. е. выполнять различные действия в зависимости от контекста, в котором они встречаются. Операция << и объект cout знают, каким образом отличать целые числа от строки и как при этом обрабатывать каждое из них. Символ '\n' в конце строки является примером управляющей последовательности.

Для компиляции программы с помощью компилятора gcc следует использовать следующую команду:

```
$ g++ -o 1 1.cpp
```

Расширение имени файла .cpp является традиционным для файлов с исходным кодом на языке С++.

Программа, использующая операторы ввода и вывода, представлена в листинге 5.2.

Листинг 5.2

```
#include <iostream>
using namespace std;
int main(){
  int a;
  cout<<"Введите a:"<<"\n";
  cin>>a;
  cout<<a<<a"*"<<a<<"="<<a*a<<endl;
  return 0;
}
```

Для вывода на экран используется “<<”, которая называется операцией вставки. Она вставляет данные в поток вывода. Оператор cin>>a; заставляет программу ждать ввода числа от пользователя. Ключевое слово cin является объектом, определенным в языке C++ для работы со стандартным потоком ввода. Этот поток содержит данные, вводимые с клавиатуры, так как стандартный поток ввода обычно присоединяется к клавиатуре. Для ввода используется “>>”, называемая операцией извлечения. Она извлекает данные из потокового объекта, стоящего в левой части, и присваивает эти данные переменной, стоящей справа. Ключевое слово endl означает вставку в поток символа окончания строки, хотя при этом так же осуществляется очистка выходного буфера. Обычно это не имеет большого значения, поэтому мы будем считать \n и endl равнозначными. Во втором операторе cout операция вставки каскадируется, то есть повторяется несколько раз.

5.2 Флаги форматирования

Флаги форматирования работают переключателями, определяющими различные форматы и способы ввода/вывода. Перечень флагов форматирования указан в таблице 5.1.

Есть несколько способов установки флагов форматирования в программе.

Так, например, все без исключения флаги могут быть установлены с помощью методов setf() и unsetf().

```
cout.setf(ios::left); //включить выравнивание текста по левому краю

cout<<"Этот текст выводится с выравниванием по левому краю";

cout.unsetf(ios::left); //вернуться к прежнему форматированию текста
```

Многие флаги могут быть установлены с помощью манипуляторов. Манипуляторы – это инструкции форматирования, которые вставляются прямо в поток ввода/вывода.

Таблица 5.1 – **Флаги форматирования класса ios (input-output stream)**

Флаг форматирования	Выполняемые действия
skipws	пропуск пробелов при вводе
left	выравнивание по левому краю
right	выравнивание по правому краю
dec	перевод в десятичную систему
oct	перевод в восьмеричную систему
hex	перевод в шестнадцатеричную систему
showpoint	показывать десятичную точку при выводе
showpos	показывать знак “+” перед положительными целыми числами
scientific	экспоненциальный вывод чисел с плавающей точкой
fixed	фиксированный вывод чисел с плавающей точкой

Наиболее распространенные флаги представлены ниже:

ws – включает пропуск пробелов при вводе;

dec – перевод в десятичную форму;

hex – перевод в шестнадцатеричную форму;

oct – перевод в восьмеричную форму;

endl – вставка разделителя строк и очистка выходного буфера;

setw(n) – установка ширины поля;

setfill(n) – устанавливается символ заполнения (по умолчанию используется пробел);

setprecision(n) – точность (устанавливается число выводимых знаков);

setiosflags(…) – устанавливает указанные флаги форматирования;

resetiosflags(…) – сбрасывает указанные флаги форматирования.

Для использования манипуляторов с аргументами необходимо использовать заголовочный файл <iomanip>.

В листинге 5.3 приведены примеры использования манипуляторов и флагов форматирования при организации вывода на экран.

Листинг 5.3

```
#include <iostream>
#include <iomanip>
#include <math.c>
using namespace std;
int main(){
    float logarifm;
```

```
        int i;
        cout.setf(ios::left);// устанавливаем выравнивание по
левому краю
        for (i=5; i<=20; i++){
      logarifm=log(i);
      cout<<setw(4)    //устанавливаем ширину поля, равную 4
          <<i
          <<setw(10)   //устанавливаем ширину поля, равную 10
          <<setprecision(3)  //устанавливаем точность 3 (кол-
во знаков после запятой)
          <<setiosflags(ios::fixed)//устанавливаем фиксиро-
ванный вывод
          <<logarifm
          <<endl;
        }
        cout.unsetf(ios::left);//отменяем выравнивание по ле-
вому краю
        return 0;
    }
```

В начале программы мы устанавливаем выравнивание выводимых данных по левому краю с использованием флага форматирования left. В результате использования этого метода все данные, выводимые до возврата к прежнему форматированию, будут выравниваться по левому краю. В нашем примере возврат к прежнему форматированию не является необходимым, т. к. программа сразу же прекращает свою работу после вывода таблицы логарифмов. В цикле for при выводе таблицы на экран используется несколько манипуляторов, применяющихся перед выводом числа на экран.

ПРИЛОЖЕНИЕ А
(обязательное)
Программа управления компиляцией make

Стремление к увеличению модульности при разработке программ может привести к тому, что проект будет состоять из множества файлов. Для формирования из них программы могут использоваться различные порождающие процедуры. Стандартом де-факто в данной области (особенно при использовании ОС Unix) является программа управления компиляцией make.

Утилита make автоматически определяет, какие части большой программы должны быть перекомпилированы и команды для их перекомпиляции.

При использовании make существенно повышается производительность труда программиста, так как он освобождается от ручной сборки программ, уменьшается загрузка компьютера, значительно уменьшается время компиляции модульных программ.

Перед использованием make необходимо создать make-файл, который описывает отношения между файлами вашей программы и содержит команды для обновления каждого файла. Файл описаний позволяет утилите make отслеживать зависимости между файлами, составляющими программную систему. Если изменен любой из них, утилита make порождает новую версию программы, перекомпилировав только те ее части, которые прямо или косвенно затронуты изменением.

Файлу описаний, задающему информацию о межфайловых зависимостях и последовательностях команд для порождения файлов, принято давать имя makefile или Makefile. В таком случае для сборки программы достаточно просто набрать *make* независимо от того, сколько файлов было изменено.

Для простых случаев файл описаний написать легко и изменяется он редко.

Основное действие утилиты make – обновление целевого файла при условии, что все файлы, от которых зависит целевой, существуют и не являются устаревшими. Целевой файл порождается заново, если исходные файлы модифицированы, а целевой – нет. Утилита make исследует граф зависимостей. Функционирование make основывается на анализе времени модификации файлов.

В качестве иллюстрации рассмотрим простой пример.

Программа proga получается из трех основных файлов x.c, y.c и z.c путем их компиляции. В соответствии с принятыми соглашениями результат работы С-компилятора будет помещен в файлы с именами x.o, y.o и z.o соответственно. Также предположим, что файлы x.c и y.c используют общие описания из включаемого файла zagolovok.h, а z.c – не использует. Пусть файлы x.c и y.c содержат строку

```
#include “zagolovok.h”
```

Тогда следующая спецификация файла описывает взаимосвязи и операции:

```
proga: x.o y.o z.o
  gcc x.o y.o z.o –o proga
x.o: x.c zagolovok.h
  gcc –c x.c
y.o: y.c zagolovoc.h
  gcc –c y.c
z.o: z.c
  gcc –c z.c
clean:
  rm –rf   *.o  proga
```

Если эту информацию поместить в файл makefile (Makefile), то команда ***make*** будет выполнять операции, необходимые для перегенерации файла proga после любых изменений, сделанных в каком-либо из четырех исходных файлов x.c, y.c, z.c или zagolovok.h. В приведенном выше примере в первой строке утверждается, что proga зависит от трех *.o*-файлов. Если эти файлы имеются в наличии, рассматривается вторая строка, которая описывает, как отредактировать связи между ними, чтобы создать proga. Третья строка говорит, что x.o зависит от x.c и от zagolovok.h. Четвертая строка говорит о том, что для создания файла x.o необходимо выполнить команду ***gcc –c x.c***. В этой строке имеется сдвиг команды ***gcc –c x.c*** вправо на две позиции “табуляция”. Этот сдвиг предусмотрен правилами оформления файла описаний (make-файла) и является обязательным.

Теперь перейдем к рассмотрению девятой и десятой строк. Часто в файл описаний включают правила с командами, которые в действительности не порождают файлов с соответствующими именами. Мнемонические имена играют роль точек входа, при обращении к которым выполняются определенные действия. Так, точка входа clean может служить для удаления ненужных файлов, что продемонстрировано в десятой строке: вызов команды ***make clean*** приведет к удалению всех объектных файлов в текущем каталоге и файла proga.

СПИСОК РЕКОМЕНДУЕМОЙ И ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

1 **Прата, С.** Программирование на языке С / С. Прата. − М. : Диасофт, 2004. − 896 с.

2 **Дейтел, Х.** Как программировать на С++ / Х. Дейтел. − М. : Бином, 2006. − 1152 с.

3 **Лафоре, Р.** Объектно-ориентированное программирование в С++/ Р. Лафоре. − СПб. : Питер, 2004. − 928 с.

4 **Седжвик, Р.** Фундаментальные алгоритмы на С / Р. Седжвик. − М. : Диасофт, 2003. − 672 с.

5 **Балащенко, Д. В.** Программирование на языке С/С++ : учеб.-метод. пособие по дисциплине «Информатика и информационные технологии». В 2 ч. Ч. 1 / Д. В. Балащенко, Д. В. Захаров. − Гомель : БелГУТ, 2006. − 84 с.

Учебное издание

Балащенко Дмитрий Валерьевич
Захаров Денис Владимирович

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С/С++

Учебно-методическое пособие по дисциплине
«Информатика и информационные технологии»
Ч а с т ь 2

Редактор *А.А. Павлюченкова*
Технический редактор *В.Н. Кучерова*

Подписано в печать 01.07.2011 г. Формат 60x84 $^{1}/_{16}$.
Бумага офсетная. Гарнитура Таймс. Печать на ризографе.
Усл. печ. л. 3,25. Уч.-изд. л. 3,84. Тираж 500 экз.
Зак № . Изд. № 101.

Издатель и полиграфическое исполнение
Белорусский государственный университет транспорта:
ЛИ № 02330/0552508 от 09.07.2009 г.
ЛП № 02330/0494150 от 03.04.2009 г.
246653, г. Гомель, ул. Кирова, 34