Федеральное агентство по образованию

Государственное образовательное учреждение
высшего профессионального образования

Московский государственный институт электроники и математики
(технический университет)

Кафедра «Управление и информатика
в технических системах»

Программирование на языке Си

Учебное пособие по дисциплине «Информатика»

Москва 2010 г.

Составитель ст. преп. В.Г. Кулаков

Учебное пособие по дисциплине «Информатика» предназначено для студентов первого курса дневного отделения. Основной целью пособия является освоение студентами правил элементарного программирования на языке Си.

Учебное пособие составлено в соответствии с программой и планом для специальности «Управление и информатика в технических системах» - 220100.

УДК 004.451.9

Программирование на языке Си. Учебное пособие по дисциплине «Информатика» / Моск. гос. ин-т электроники и математики; Сост. В.Г. Кулаков. М., 2010, 44 с.

Рис. 23. Библиогр.: 4 назв.

Содержание Стр.

1. Язык программирования Си 4
2. Конструкции языка Си 4
3. Условный оператор и операторы циклов 7
4. Примеры простых программ на Си 8
5. Стандартные функции ввода-вывода 10
6. Математические функции 14
7. Операции с текстовыми строками 14
8. Стандартные функции для обработки строк 16
9. Перестановки, размещения, сочетания, факториал 17
10. Вычисление элементарных функций 18
11. Матрицы и операции над ними 20
12. Перестановка элементов массива 23
13. Основные операции обработки изображений 24
14. Функций для генерации случайных чисел 28
15. Пузырьковая сортировка 29
16. Быстрая сортировка 30
17. Функция time 31
18. Измерение производительности 31
19. Структуры в языке Си 32
20. Обработка списков 33
21. Очередь 37
22. Стек 39
23. Двоичные деревья 40
24. Распределение свободной памяти 42
Список использованной литературы 43

1. Язык программирования Си

Язык Си был разработан Деннисом Ритчи в 1972 году. От других языков программирования он отличается тем, что изначально был ориентирован на работу с аппаратурой и, вследствие этого, является удобным средством для освоения аппаратного обеспечения компьютера.

Язык Си послужил прототипом для создания многих современных языков программирования, например C++ и C#. В настоящее время он считается устаревшим и используется только в учебном процессе (для освоения основ программирования), а также при программировании микроконтроллеров и самых примитивных микропроцессоров с небольшим объемом адресного пространства памяти. Дело в том, что Си имеет более простую структуру, чем объектно-ориентированные языки, а составленные на нем программы занимают намного меньше оперативной памяти.

Так как язык Си устарел, компиляторы с этого языка уже не разрабатывают. На персональных компьютерах для трансляции Си-программ используются компиляторы с языка C++ – усовершенствованной версии Си. Для того, чтобы компилятор использовал при трансляции правила Си, а не C++, достаточно изменить расширение имени файла программы с «.cpp» на «.c».

2. Конструкции языка Си

Исходная программа – это совокупность директив, указаний компилятору, объявлений и определений. Определение переменной задает имя, тип и начальное значение переменной. Определение функции задает имя функции, ее структуру, параметры и тип возвращаемой величины.

Программа может быть разделена на несколько файлов. Исходный файл – это текстовый файл, который содержит часть программы или всю программу. Отдельные файлы перед компиляцией можно соединять в один большой файл посредством директивы #include. Если имя файла в директиве заключено в угловые скобки, поиск файла осуществляется в стандартном каталоге интегрированной среды Си, а если имя заключено в двойные кавычки – в текущем каталоге. Если имя указано полностью (включая путь доступа к файлу), то файл берется из указанного места и поиск не производится.

В тексте программы для пояснения выполняемых действий можно использовать комментарии. **Комментарий** – это последовательность символов, которая начинается с комбинации символов «/*» и заканчивается комбинацией «*/». Такой комментарий может занимать несколько строк. В языке C++ введен однострочный комментарий, который начинается с комбинации «//». Компилятором комментарии игнорируются.

Используемые в программе неизменные объекты называют **константами,** изменяющиеся объекты – **переменными.**

Идентификаторы – это имена переменных, функций и меток. Допускается любое число символов в идентификаторе. Компилятор Си рассматривает буквы верхнего и нижнего регистров как **различные** символы.

Указатель – это переменная, которая указывает на другую переменную, то есть хранит адрес (местоположение) этой переменной в оперативной памяти.

Константа – это число, символ или строка символов. Константы используются в программе как неизменяемые величины. В языке Си различают четыре типа констант: целые, с плавающей точкой, символы и строки.

Целая константа – это десятичное, восьмеричное или шестнадцатеричное число, которое представляет целую величину. Программист может задать для константы тип long, приписав букву l или L в конец константы.

Константа с плавающей точкой – это действительное десятичное число, которое может включать целую и дробную части, а также экспоненту. Целая часть отделяется от дробной десятичной точкой, экспонента отделяется символом E или e.

Константа-символ – это буква, цифра или знак пунктуации, заключенный в одиночные кавычки.

Строка – это последовательность букв, цифр и символов, заключенная в двойные кавычки. В памяти строка хранится в виде массива символов. Строка завершается значением 0 (признаком конца строки). При подсчете длины строки завершающий ноль не учитывается.

В символьных и строковых константах можно использовать специальные символьные комбинации – ESC-последовательности. Часто используются последовательность «\n» – перевод строки и последовательность «\t» – табуляция (переход к следующей колонке шириной 8 позиций).

Операции – это специальные комбинации символов, специфицирующие действия по преобразованию различных величин.

Мультипликативные операции: «*» – умножение, «/» – деление, «%» – получение остатка от деления.

Аддитивные операции: «+» – сложение, «–» – вычитание или арифметическое отрицание.

Операции отношений: «<» – меньше, «<=» – меньше или равно, «>» – больше, «>=» – больше или равно, «==» – равно, «!=» – не равно.

Логические операции: «!» – НЕ, «&&» – И, «||» – ИЛИ.

Операции присваивания: «++» – инкремент (увеличение на 1), «––» – декремент (уменьшение на 1), «=» – простое присваивание, «+=» – сложение с присваиванием, «–=» – вычитание с присваиванием, «*=» – умножение с присваиванием.

Операции с указателями: & – получение ссылки (указателя) на переменную; * – извлечение значения переменной, на которую ссылается указатель.

Операнд – это константа, идентификатор, вызов функции или выражение. Каждый операнд имеет тип. Операнд может быть преобразован из оригинального типа к другому типу посредством операции преобразования типов.

Оператор – это языковая конструкция, выполняющая определенные действия. Операторы управляют процессом выполнения программы.

В языке Си используются следующие операторы:
break – завершение цикла;
continue – переход к следующей итерации цикла;
do – цикл с постусловием;
for – цикл со счетчиком;
goto – безусловный переход по метке;
if – условие;
return – завершение функции;
switch – переключатель;
while – цикл с предусловием.

Имеются также оператор-выражение и составной оператор (блок). Составной оператор ограничивается фигурными скобками. Все другие операторы Си заканчиваются точкой с запятой.

Выражение – это комбинация операндов и операций, значением которой является отдельная величина.

Блок – это составной оператор (группа операторов, ограниченная фигурными скобками). Составные операторы состоят из объявлений и операторов. Блоки могут быть вложенными.

Функция – это независимая совокупность объявлений и операторов, обычно предназначенная для выполнения определенной задачи. Исходная программа включает по крайней мере одну функцию – главную функцию main, которая может вызывать другие функции..

Все переменные Си должны быть явно объявлены перед их использованием. Функции могут быть объявлены явно или неявно. Объявление может включать один или несколько деклараторов.

Декларатор – это идентификатор, который может быть определен с квадратными скобками, звездочкой или круглыми скобками для объявления массива, указателя или функции соответственно. Когда объявляется простая переменная, структура или объединение, то декларатор – это идентификатор.

Язык Си поддерживает следующие **базовые типы данных:**
char – символ (принимает значения в диапазоне –128…127),
int – целое (диапазон значений зависит от реализации компилятора),
short – короткое целое (–32768…32767),
long – длинное целое (–2147483648…2147483647),
unsigned char – символ без знака (0…255),
unsigned – целое без знака (диапазон зависит от реализации компилятора),
unsigned short – короткое целое без знака (0…65535),
unsigned long – длинное целое без знака (0…4294967295),
float – число с плавающей запятой,
double – число с плавающей запятой удвоенной точности.

Имеется также тип void, предназначенный для объявления функций, которые не возвращают значения.

Объявление переменной определяет ее имя и тип.

Примеры объявлений переменных:

```
int x;          //Переменная целого типа
float f;        //Переменная вещественного типа
int list[20]; //Массив из 20 целых чисел
char *cp;       //Указатель на символ
```

При объявлении массива количество элементов, содержащихся в массиве, указывается после имени массива в квадратных скобках.

В объявлении переменной может быть присвоено начальное значение посредством **инициализатора.**

Инициализатор состоит из знака равенства и присваиваемого значения. Если требуется инициализировать массив чисел, элементы массива перечисляются в фигурных скобках через запятую. Если в массив записывается текстовая константа, то можно не указывать количество элементов массива – компилятор определит его автоматически.

Примеры использования инициализаторов:

```
int i=0;
float x=5.0, y=10.0;
int A[5]={1,2,3,4,5};
char text[]="Текст";
```

В первой строке примера объявлена целая переменная i, которой присваивается значение 0. Во второй строке вещественной переменной x присваивается значение 5, а переменной y – значение 10. В третьей строке задан массив из 5 элементов, в который записываются значения от 1 до 5. В четвертой строке задан символьный массив text, в который заносится текстовая строка.

3. Условный оператор и операторы циклов

Условный оператор **if** имеет следующий синтаксис:

```
if (условие) оператор1 else оператор2
```

Тело оператора if выполняется селективно, в зависимости от условия: если оно истинно, то выполняется оператор1, иначе – оператор2. Предложение else может быть опущено.

Оператор **for** обычно задает цикл со счетчиком и имеет следующий синтаксис:

```
for (инициализация; условие; приращение) оператор
```

Цикл for выполняется, пока условие истинно. Первым шагом является вычисление выражения инициализации. Далее проверяется условие и, если оно истинно, выполняется оператор тела цикла, а затем вычисляется приращение и цикл повторяется.

Оператор **while** задает цикл с предусловием и имеет следующий вид:

```
while (условие) оператор
```

Цикл while выполняется, пока условие истинно. Вначале проверяется условие, затем выполняется оператор тела цикла.

Оператор **do** задает цикл с постусловием и имеет вид:

```
do оператор while (условие);
```

Цикл do выполняется до тех пор, пока условие истинно. Вначале выполняется оператор тела цикла, затем проверяется условие.

Оператор **break** прерывает выполнение цикла.

Оператор **continue** передает управление на следующую итерацию цикла.

4. Примеры простых программ на Си

Освоение любого языка программирования следует начинать с простых примеров. Рассмотрим пример простой программы на языке Си:

```
#include <stdio.h>
int main()
{
  printf("Привет!\n");
}
```

Первая строка программы сообщает компилятору, что он должен включить информацию о стандартной библиотеке ввода-вывода. В файле stdio.h содержится описаний функций ввода-вывода, которые необходимы практически в любой программе.

Следующая строка содержит заголовок главной функции программы – функции main. Функция main присутствует в любой программе на языке Си. С этой функции начинается выполнение программы, и в дальнейшем она управляет вычислительным процессом, вызывая при необходимости другие функции.

Третья строка содержит открывающую фигурную скобку, с которой начинается блок операторов. Блок – это группа операторов, которые должны выполняться совместно. Тело функции всегда является блоком – даже в том случае, если содержит только один оператор.

Четвертая строка содержит обращением к функции с именем printf и аргументом "Привет!\n". Функция printf – универсальная функция форматных преобразований, первым аргументом которой является строка символов (форматная строка). Если в форматной строке нет знаков %, как данном примере, она печатается без изменений. Последовательность «\n» обозначает символ новой строки, который служит указанием для перехода к левому краю следующей строки на терминале (printf не обеспечивает автоматического перехода на новую строку).

В последней строке программы находится закрывающая фигурная скобка, которая завершает блок операторов и функцию main.

С знака % в форматной строке функции printf начинаются спецификации преобразования. Каждая спецификация указывает, куда должен подставляться очередной из следующих за форматной строкой аргументов функции и в какой форме он должен печататься. Форма представления значения аргумента задается буквой, следующей за знаком %. Например, спецификация «%d» означает, что значение должно быть напечатано как целое число, а спецификация «%f» – как число с плавающей точкой.

В следующем примере мы рассмотрим программу, которая использует цикл со счетчиком, чтобы вывести на экран столбец цифр от 0 до 9:

```
#include <stdio.h>
void main(void)
{
```

```
  int i;
  for(i=0; i<=9; i++) printf("%d\n", i);
}
```

По умолчанию главная функция main имеет тип int, так как при завершении своей работы может возвращать операционной системе некое целочисленное значение – так называемый «код завершения». Однако обычно никакой код возвращать не требуется и для функции main задают тип void.

Если функция не имеет аргументов, то в круглых скобках после имени также пишут void, чтобы дать четкое указание компилятору об отсутствии аргументов (тогда компилятор не будет выдавать лишних предупреждений в процессе работы).

Если в цикле производится вычисление суммы, то перед началом цикла значение суммы следует **обнулить**. Если в цикле вычисляется произведение, перед началом цикла произведению нужно присвоить значение **1.**

Для того, чтобы прибавить к сумме некоторое значение, обычно используют оператор сложения с присваиванием «+=»; для того, чтобы умножить произведение не некоторое значение, используют оператор умножения с присваиванием «*=».

Пример программы, которая вычисляет сумму чисел от 1 до 99 и произведение чисел от 1 до 7, а затем выводит на экран полученные значения:

```
#include <stdio.h>
void main(void)
{
  int i, s, p;
  // Вычисление суммы чисел от 1 до 99
  s = 0;
  for(i=1; i<100; i++) s += i;
  printf("Сумма чисел от 1 до 99 равна %d.\n", s);
  // Вычисление произведения чисел от 1 до 7
  p = 1;
  for(i=1; i<=7; i++) p *= i;
  printf("Произведение чисел от 1 до 7 равно %d.\n", p);
}
```

Пример программы, использующей цикл while для вычисления длины строки:

```
#include <stdio.h>
void main(void)
{
  int i;
  char s[]="Строка";
  i=0;
  while(s[i] != 0) i++;
  printf("Длина строки %d символов.\n", i);
}
```

Цикл while в данном примере продолжается до тех пор, пока не найдено значение 0, являющееся признаком конца строки (как уже было указано выше, завершающий символ при определении длины строки не учитывается).

Индекс i в цикле пробегает по всем элементам массива. Счет элементов массива в языке Си ведется не с единицы, а с **нуля.**

В следующем примере требуется задать массиве чисел с плавающей запятой F из 5 элементов, найти в массиве максимальное значение, а затем напечатать это значение и номер содержащего его элемента:

```
#include <stdio.h>
void main(void)
{
  int i, k;
  float F[5] = {1, 12.6, -7.3E3, 124.5, -15};
  float b;
  k = 0;    //Запомнить номер первого элемента массива
  b = F[0]; //Запомнить значение первого элемента
  // Цикл по всем элементам, начиная с второго
  for(i=1; i<5; i++)
  {
    if(F[i]>b) //Сравнить очередное значение с ранее записанным
    {
      //Если новое значение больше старого:
      k = i;    //Запомнить номер элемента
      b = F[i]; //Запомнить новое значение
    }
  }
  printf("Номер максимального элемента: %d, значение: %f\n",k,b);
}
```

5. Стандартные функции ввода-вывода

Функции, которые часто используются в программах на Си, принято собирать в так называемые библиотеки стандартных функций. Описания функций не являются составной частью описания языка Си – функции стандартизируются отдельно. Функции обычно группируют по назначению и каждую группу описывают в отдельном заголовочном файле (если в программе используются функции из какой-либо группы, соответствующий заголовочный файл должен быть подключен в начале программы директивой #include).

Функции, обеспечивающие ввод и вывод данных, собраны в стандартную библиотеку ввода-вывода, которая описана в файле stdio.h. Исходный файл, который обращается к функциям из этой библиотеки, должен содержать следующую строку:

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

Библиотека ввода-вывода содержит десятки функций, предназначенных для выполнения различных операций ввода-вывода. Мы рассмотрим только те функции, которые будут использоваться в последующих упражнениях.

Функция **gets** читает строку из потока стандартного ввода stdin и запоминает ее в указанном буфере buffer:

```
char *gets(char *buffer);
```

Вводимая строка состоит из набора символов и заканчивается символом новой строки «\n». Функция gets записывает строку в массив, который был задан в качестве аргумента. После завершения работы функция возвращает свой аргумент –

указатель на введенную строку (значение указателя NULL свидетельствует об ошибке или достижении конца файла). Перед возвратом функция замещает символ новой строки признаком завершения строки – кодом с значением 0.

Функция **puts** записывает выбранную строку string в стандартный поток вывода stdout, заменяя символ окончания строки символом новой строки «\n». Функция объявлена следующим образом:

```
int puts (char *string);
```

Аргументом функции puts является указатель на строку. Функция возвращает последний записанный символ, которым обычно является символ новой строки (значение EOF свидетельствует об ошибке).

Обычно потоком стандартного ввода является клавиатура, потоком вывода – экран терминала, однако при необходимости эти потоки можно перенаправить в файлы. В последующих упражнениях перенаправление ввода-вывода не требуется, поэтому для краткости в дальнейшем поток ввода будем называть клавиатурой, поток вывода – экраном.

В качестве примера рассмотрим программу, которая вводит строку с клавиатуры и выводит ее на экран:

```
#include <stdio.h>
void main(void)
{
  char line[80];//Массив, предназначенный для запоминания строки
  gets(line);   //Ввести строку с клавиатуры
  puts(line);   //Вывести строку на экран
}
```

Так как при вводе строки она также отображается на экране, после выполнения программы на экране будет две одинаковых строки.

Функция **getchar** вводит с клавиатуры один символ и возвращает код символа, который имеет целочисленный тип int. Функция объявлена следующим образом:

```
int getchar(void);
```

Как видно из описания, входных аргументов функция не имеет.

В процессе отладки функция getchar часто используется, чтобы приостановить закрытие окна пользователя до тех пор, пока он не нажмет какую-либо клавишу. Функция также позволяет приостановить вывод длинного текста или большой таблицы: после вывода на экран очередной «страницы» текста программа будет ждать нажатия любой клавиши.

В качестве примера рассмотрим программу, которая выводит на экран натуральные числа от 1 до 100 порциями по 20 чисел:

```
#include <stdio.h>
void main(void)
{
  int i;
  for(i=1; i<=100; i++)
  {
    printf("%d\n",i);
    if(i%20==0)
    {
```

```
      printf("Для продолжения вывода нажмите любую клавишу.\n");
      getchar(); //Ожидать нажатия любой клавиши
    }
  }
}
```

В этом примеру внутрь цикла вывода чисел добавлено условие, в котором проверяется делимость очередного числа на 20. Если число делится без остатка, выдается текстовое сообщение и вывод данных приостанавливается до тех пор, пока не будет нажата клавиша.

Функции **printf** (для вывода) и **scanf** (для ввода) позволяют преобразовывать численные величины в символьное представление и обратно. Функции описаны следующим образом:

```
int printf(const char *format, ...);
int scanf(const char *format, ...);
```

Функция printf возвращает количество выведенных символов (значение EOF свидетельствует об ошибке). Функция scanf возвращает количество значений, успешно присвоенных указанным переменным.

Первый аргумент функций printf и scanf представляет собой **управляющую (форматную) строку**, которая указывает, в каком виде следует выводить или вводить последующие аргументы. Управляющая строка содержит два типа объектов: обычные символы и **спецификации преобразований**. Каждая спецификация преобразования начинается с символа % и заканчивается символом преобразования. Символы преобразования: **d** – десятичная форма, **o** – восьмеричная форма, **x** – шестнадцатеричная форма, **u** – беззнаковая десятичная форма, **c** – отдельный символ, **s** – строка, **e** – число в научном формате (с экспонентой), **f** – число с плавающей запятой. В спецификациях преобразований функции printf между знаком % и символом преобразования могут находиться:

- число, задающее ширину поля,
- точка-разделитель,
- число, задающее точность представления,
- модификатор длины l для данных типа long.

Функции printf и scanf могут иметь произвольное количество аргументов. Аргументы, следующие за управляющей строкой, для функции printf представляют собой подлежащие выводу значения переменных и выражений, а для функции scanf – **указатели** на переменные, в которых должны быть сохранены введенные значения.

Таким образом, перед каждым аргументами функции scanf, который является именем переменной, должен стоять амперсанд – символ «&». Если вводится строка, то перед именем массива амперсанд не нужен, так как имя массива само по себе является указателем.

Приведем в качестве примера программу, которая вводит значение N и печатает таблицу квадратов чисел от 1 до N:

```
#include <stdio.h>
void main(void)
{
```

```
  int i, N;
  printf(”Введите N: ”);
  scanf(”%d”, &N);
  printf(”Таблица квадратов:\n”);
  for(i=1; i<=N; i++) printf(”%2d %4d\n”, i, i*i);
}
```

Для того, чтобы колонки таблицы выводились ровно, требуется задать для каждого выводимого значения ширину поля. В данном случае ширина первой колонки – два знакоместа, ширина второй – четыре.

В качестве аргументов функции printf можно использовать выражения. В приведенном примере с помощью выражения вычисляется квадрат числа i.

Рассмотрим пример другой программы, которая использует цикл while для вычисления длины строки:

```
#include <stdio.h>
void main(void)
{
  int i;
  char s[100];
  printf(”Введите строку:\n”);
  gets(s);
  i=0;
  while(s[i] != 0) i++;
  printf(”Длина строки %d символов.\n”);
}
```

Для ввода строки в этом примере использована функция gets, так как scanf обеспечивает только ввод отдельных слов.

В следующем примере по радиусу круга вычисляется его площадь:

```
#include <stdio.h>
main()
{
  double r, pi=3.1415926535;
  printf("Введите радиус круга: ");
  scanf("%lf", &r);
  printf("Площадь круга: %f\n", pi*r*r);
}
```

В стандарте языка Си отсутствует специальное обозначение для числа π и его приходится задавать в виде вещественной константы.

При выводе чисел типа double использовать в спецификации преобразования признак удвоенной точности (букву l) не обязательно. Однако при вводе значений типов long и double такой признак в спецификация необходим, иначе значение будет введено неверно.

6. Математические функции

Математические функции Си описаны в заголовочном файле math.h. Программа, которая использует математические функции, должна содержать следующую строку:

```
#include <math.h>
```

Наиболее часто применяются следующие функции:

double cos(double x) – вычисляет косинус аргумента;
double fabs(double x) – вычисляет абсолютное значение аргумента;
double exp(double x) – вычисляет экспоненциальную функцию аргумента;
double log(double x) – вычисляет натуральный логарифм аргумента;
double log10(double x) – вычисляет десятичный логарифм аргумента;
double sin(double x) – вычисляет синус аргумента;
double sqrt(double x) – вычисляет квадратный корень аргумента;
double tan(double x) – вычисляет тангенс аргумента.

Из приведенных описаний видно, что математические функции работают с вещественными числами удвоенной точности: аргументы имеют типа double и возвращаемое значение также имеет тип double.

В качестве примера рассмотрим программу, которая вводит значение *x*, вычисляет значение e^x и выводит полученное значение на экран:

```
#include <stdio.h>
#include <math.h>
main()
{
  double x, y;
  printf("Введите значение x: ");
  scanf("%lf", &x);
  y = exp(x);
  printf("exp(x)=%f\n", y);
}
```

7. Операции с текстовыми строками

Строка представляет собой массив символов типа char, ограниченный нулем.

Операции обработки строк по своей сути очень просты и, поэтому, удобны для изучения основ алгоритмизации и программирования. Рассмотрим в качестве примера функцию, которая определяет длину строки (завершающий ноль не учитывается):

```
int Len(char *string)
{
  int j=0;
  while(string[j] != 0) j++;
  return j;
}
```

В данном примере индекс массива инкрементируется в цикле до тех пор, пока не найден символ завершения строки. При выходе функция возвращает индекс последнего найденного ненулевого элемента массива. Так же, как стандартная функция strlen, функция Len при определении длины строки не учитывает завершающий символ.

Если для выполнения некоторой операции над массивом используется подпрограмма, то в качестве аргумента в нее передается не сам массив, а только указатель на его местоположение в памяти, В приведенном примере передается указатель на текстовую строку string.

При выполнении операции вставки символа в строку необходимо освободить место для символа, подвинув на одну позицию вправо все символы строки, расположенные за позицией вставки (включая завершающий 0). Последовательность действий показана на рисунке 1. Перестановка символов начинается с конца строки, поэтому должен использоваться цикл с декрементом счетчика.

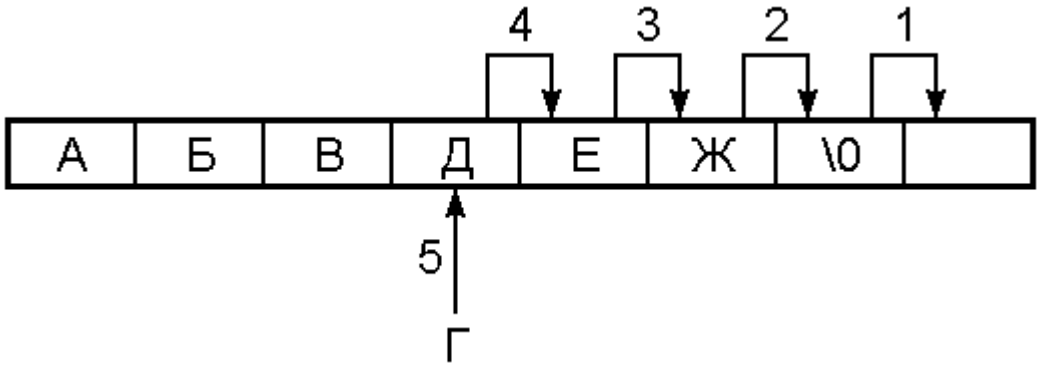

Рис. 1. Последовательность действий при выполнении операции вставки символа

Если требуется вставить в строку слово длинной k символов, то перед началом копирования слова необходимо освободить для него место, подвинув на k позиций вправо все символы строки, расположенные за позицией вставки.

Если требуется добавить символ в конец строки, то проводить перестановку элементов строки не нужно: новый символ записывается за последним символом строки, а в следующую позицию заносится нулевой символ окончания строки ('\0').

При выполнении операции удаления символа из строки все элементы (включая символ окончания строки), расположенные после удаляемого элемента, сдвигаются на одну позицию влево. Последовательность действий при удалении символа показана на рисунке 2.

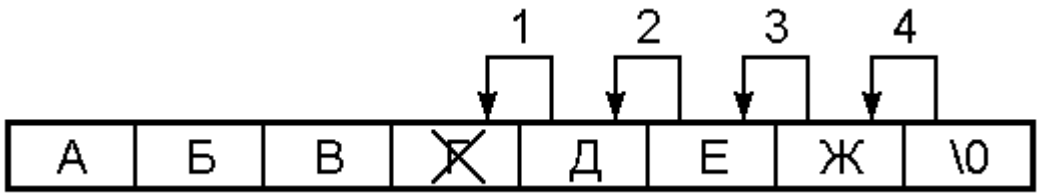

Рис. 2. Последовательность действий при выполнении операции удаления символа

Текст в простейшем случае представляет собой двумерный массив символов, содержащий строки. Недостатки такого примитивного способа хранения данных: неэффективное использование памяти и необходимость перемещения большого объема информации при добавлении и удалении строк.

8. Стандартные функции для обработки строк

Язык Си долгое время использовался для создания текстовых редакторов, поэтому он располагает широким набором средств для обработки текстовой информации.

В языке Си имеется набор стандартных функций для работы с текстовыми строками. Описания строковых функций находятся в заголовочном файле **string.h**, который необходимо подключить директивой #include в начале программы..

Рассмотрим функции, которые используются наиболее часто.

Функция **strcat** добавляет одну строку к другой. Функция объявлена следующим образом:

```
char *strcat(char *dest, const char *src);
```

Функция добавляет копию строки src в конец dest и возвращает указатель на результирующую строку.

Функция **strchr** ищет в строке первое вхождение заданного символа. Функция объявлена следующим образом:

```
char *strchr(const char *s, int c);
```

Функция ищет первое вхождение символа c в строку s. Она возвращает указатель на первое вхождение символа c в s; если c не обнаружен в s, то strchr возвращает NULL.

Функция **strcmp** сравнивает одну строку с другой. Функция объявлена следующим образом:

```
int strcmp(const char *s1, const char *s2);
```

Функция strcmp осуществляет сравнение строк s1 и s2, начиная с первого символа каждой строки, до тех пор, пока очередные соответствующие символы в строках не будут различны или пока не будут достигнуты концы строк. Функция возвращает отрицательное значение, если s1 меньше чем s2, ноль, если s1 равна s2, и положительное значение, если s1 больше чем s2

Функция **strcpy** копирует одну строку в другую. Функция объявлена следующим образом:

```
char *strcpy(char *dest, const char *src);
```

Функция копирует строку src в dest и возвращает dest.

Функция **strlen** вычисляет длину строки. Функция объявлена следующим образом:

```
size_t strlen(const char *s);
```

Функция вычисляет длину строки s и возвращает полученное значение (символ конца строки не учитывается).

Функция **strstr** ищет в строке вхождение заданной подстроки. Функция объявлена следующим образом:

```
char *strstr(const char *s1, const char *s2);
```

Функция осуществляет поиск в s2 первого вхождения подстроки s1. Функция возвращает указатель на элемент в строке s1, с которого начинается s2. Если s2 не обнаружена в s1, функция возвращает NULL.

9. Перестановки, размещения, сочетания, факториал

Пусть Q – некоторое конечное множество, состоящее из n элементов:

$$Q = \{q_1, q_2, ..., q_n\}.$$

Будем образовывать из элементов множества Q упорядоченные множества. В качестве первого возьмем множество, в котором элементы расположены в порядке возрастания их номеров, второе образуем, поменяв местами первый и второй элементы и т.д.

Всевозможные конечные упорядоченные множества, которые можно получить путем перестановки элементов множества Q, называются **перестановками** из n элементов. Таким образом, перестановка есть не что иное, как способ упорядочивания элементов множества.

Число перестановок из n элементов (**n факториал**) равно произведению из n последовательных натуральных чисел, начиная с единицы:

$$n! = 1 \cdot 2 \cdot ... \cdot (n-1) \cdot n = \prod_{i=1}^{n} i.$$

Примечание: принято считать, что 0! = 1.

Рассмотрим пример программы, которая запрашивает ввод значения n, вычисляет $n!$ и выводит результат на экран:

```
#include <stdio.h>
void main()
{
  int i, n, nf;
  printf("Введите n: ");
  scanf("%d",&n);
  // Цикл вычисления факториала
  nf=1;
  for(i=1;i<=n;i++) nf*=i;
  printf("n! = %d\n",nf);
}
```

Если в цикле производится вычисление произведения, то перед началом цикла значение следует присвоить значение 1 соответствующей переменной. В данном примере значение произведения накапливается в переменной nf.

Конечные упорядоченные подмножества, содержащие m различных элементов, выбранных из n элементов заданного множества Q, называются **размещениями** из n

элементов по *m*. Общее число возможных различных размещений из *n* элементов по *m* обозначается как A_n^m и вычисляется по следующей формуле:

$$A_n^m = \frac{n!}{(n-m)!}.$$

Конечные неупорядоченные подмножества, содержащие *m* различных элементов, выбранных из *n* элементов заданного множества *Q*, называются **сочетаниями** из *n* элементов по *m*. Общее число возможных различных сочетаний из *n* элементов по *m* обозначается как C_n^m и вычисляется по следующей формуле:

$$C_n^m = \frac{n!}{m!(n-m)!}.$$

10. Вычисление элементарных функций

Вычисление факториала используется в некоторых алгоритмах вычисления элементарных функций. Например, при вычислении тригонометрических функций и экспоненты применяется разложение в ряд Тейлора:

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots;$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots;$$

$$e^{-x} = 1 - x + \frac{x^2}{2!} - \frac{x^3}{3!} + \frac{x^4}{4!} - \ldots.$$

Многоточие означает, что ряд имеет бесконечное количество членов. Однако вычислительный цикл не должен быть бесконечным, поэтому количество членов ряда ограничивают в соответствии с заданной погрешностью вычислений (если пренебречь ошибками округления, ошибка вычислений не превосходит первого отброшенного члена ряда).

На языке Си программа вычислении синуса выглядит следующим образом:

```
#include <stdio.h>
void main()
{
  double x, sinx, xx, px, dx, sgn, n, nf;
  // Ввести значение x
  printf("Введите x: ");
  scanf("%lf",&x);
```

```
  // Вычислить квадрат x
  xx=x*x;
  // Присвоить переменным начальные значения,
  // соответствующие первому члену ряда
  px=x;
  sgn=1;
  n=1;
  nf=1;
  dx=x;
  // Сумме присвоить значение первого члена ряда
  sinx=x;
  // Цикл для вычисления очередного члена ряда
  while(dx>0.0001 || dx<-0.0001)
  {
    // Сменить знак
    sgn=-sgn;
    // Вычислить значение степени x
    px*=xx;
    // Вычислить факториал
    n++;
    nf*=n;
    n++;
    nf*=n;
    // Вычислить значение очередного члена
    dx=sgn*px/nf;
    // Прибавить вычисленное значение к сумме
    sinx+=dx;
  }
  // Вывод результата на экран
  printf("sin(x) = %f\n", sinx);
}
```

Программа начинает вычисление с первого члена ряда и продолжает работу до тех пор, пока не будет найден член ряда, не превосходящий по абсолютной величине 0,0001, то есть в данном примеры вычисление выполняется с точностью до четырех знаков после запятой.

В программе используются следующие переменные:

- переменная x содержит значение x;
- переменная xx является вспомогательной и содержит значение x^2;
- переменная n содержит значение n для очередного члена ряда;
- переменная px содержит значение x^n;
- переменная nf содержит значение $n!$;
- переменная sgn отвечает за смену знака перед очередным членом ряда (поочередно принимает значения 1 и –1).
- переменная dx содержит значение очередного члена ряда;
- переменная sinx содержит сумму членов ряда.

С целью ускорения вычислений при расчете значений $n!$ и x^n для очередного члена ряда используются значения, полученные для предыдущего члена.

11. Матрицы и операции над ними

Прямоугольная таблица, состоящая из $m \times n$ чисел (рис. 3), называется **матрицей** из m строк и n столбцов (матрицей размера $m \times n$).

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{21} \\ \dots & \dots & \dots & \dots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix}$$

Рис. 3. Матрица чисел размера $m \times n$

Числа a_{ij} (i = 1, …, m; j = 1, …, n) называются **элементами** матрицы; индекс i указывает номер строки, индекс j – номер столбца.

В языке Си матрицы хранятся в виде двумерных массивов. В описании двумерного массива за именем массива следует две пары квадратных скобок: в первой паре задается количество строк матрицы, во второй паре – количество колонок. Например, объявление матрицы A размера 3×5, содержащей целые чисел, имеет следующий вид:

```
int A[3][5];
```

Элементы массива запоминаются в последовательных возрастающих адресах памяти. Элементы матрицы запоминаются друг за другом построчно: сначала запоминаются элементы первой строки, затем второй и т.д.

Если число строк матрицы равно числу ее столбцов, то матрица называется **квадратной**. Если квадратная матрица имеет размер $n \times n$, то число n называется **порядком** матрицы.

Две матрицы A и B называются **равными**, если они имеют одинаковое число строк и столбцов и $a_{ij} = b_{ij}$ для i = 1, …, m и j = 1, …, n (т.е. совпадают значения элементов, стоящих на соответствующих местах).

Основными арифметическими операциями над матрицами являются умножение матрицы на число, сложение матриц и умножение матриц.

Произведением числа λ и **матрицы** A называется матрица B, элементы которой вычисляются по правилу $b_{ij} = \lambda a_{ij}$.

Суммой двух матриц A и B, имеющих равные числа строк и столбцов, называется матрица S, элементы которой вычисляются по правилу $s_{ij} = a_{ij} + b_{ij}$.

Аналогично, **разностью** матриц A и B называется матрица C, составленная из разностей соответствующих элементов матриц A и B.

Для выполнения операций с матрицами в языке Си используются вложенные циклы for: внешний цикл – по строкам, внутренний – по столбцам. Для ввода и вывода массивов и матриц также используются циклы for.

Чтобы получить доступ к отдельному элементу двумерного массива, нужно в первой паре квадратных скобок, следующих после имени массива, указать индекс строки, в которой хранится элемент, а во второй паре скобок – индекс столбца.

Рассмотрим в качестве примера программу, которая вычисляет матрицу C, являющуюся суммой матриц A и B:

```
#include <stdio.h>
void main()
{
  int i, j, k, C[3][3];
  int A[3][3]={{1,2,3},{4,5,6},{7,8,9}};
  int B[3][3]={{11,12,13},{14,15,16},{17,18,19}};
  // Вычисление суммы матриц
  for(i=0;i<3;i++) for(j=0;j<3;j++) C[i][j]=A[i][j]+B[i][j];
  // Печать результата
  for(i=0;i<3;i++)
    printf("%d\t%d\t%d\n",C[i][0],C[i][1],C[i][2]);
}
```

Квадратные матрицы A и B, имеющие размер 3×3, в данном примере заданы при помощи инициализаторов.

Результирующая матрица C печатается по строкам. В данном примере размер матрицы известен заранее, поэтому для печати достаточно одного цикла (по строкам). Для того, чтобы при выводе матрицы колонки печатались ровно, можно либо использовать табуляцию для разделения соседних значений (как в приведенном примере), либо задавать в форматной строке для каждого значения фиксированную ширину поля вывода.

Если задана матрица A размера $m \times n$ и матрица B размера $n \times k$, то **произведением** матриц A и B называется матрица C размера $m \times k$, каждый элемент которой вычисляется по правилу

$$c_{ij} = \sum_{r=1}^{n} a_{ir} b_{rj}$$

т.е. элемент, стоящий в i-й строке и j-м столбце матрицы C получается в результате умножения i-й строки матрицы A на j-й столбец матрицы B.

Как видно из формулы, для вычисления произведения матриц требуется **три** вложенных цикла: цикл по строкам матрицы C, цикл по столбцам матрицы C и цикл накопления суммы. Перед началом внутреннего цикла значение элемента матрицы C, в котором будет накапливаться сумма, следует обнулить.

Элементы a_{11}, a_{22}, …, a_{nn} квадратной матрицы порядка n называются **диагональными элементами**. Говорят также, что элементы a_{11}, a_{22}, …, a_{nn} стоят на **главной диагонали.**

Квадратная матрица, у которой $a_{ij} = 0$ для всех $i \neq j$, называется **диагональной** (рис. 4).

$$\begin{bmatrix} a_{11} & 0 & \dots & 0 \\ 0 & a_{22} & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & a_{mn} \end{bmatrix}$$

Рис. 4. Диагональная матрица

Квадратная матрица, все диагональные элементы которой равны 1, а остальные – нули, называется **единичной** и обозначается символом *E* (рис. 5).

$$\begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 1 & \dots & 0 \\ \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & 1 \end{bmatrix}$$

Рис. 5. Единичная матрица

Пусть A – матрица размера $m \times n$. Матрица A^{T}, получающаяся из матрицы A путем замены строк столбцами ($a^{\mathrm{T}}_{\mathrm{ji}} = a_{\mathrm{ij}}$), имеет размер $n \times m$ и называется **транспонированной** (рис. 6).

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Рис. 6. Исходная и транспонированная матрицы

Квадратная матрица A называется **симметрической**, если $A^{\mathrm{T}} = A$. Пример симметрической матрицы приведен на рис. 7.

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}$$

Рис. 7. Пример симметрической матрица

Рассмотрим пример программы, которая вводит матрицу A размером 3×2, строит транспонированную матрицу A^T и выводит результат на экран:

```
#include <stdio.h>
void main()
{
  int i, j, k, A[3][2], AT[2][3];
  // Ввод матрицы
  for(i=0;i<3;i++) scanf("%d %d", &A[i][0], &A[i][1]);
  // Транспонирование
  for(i=0;i<2;i++) for(j=0;j<3;j++) AT[i][j] = A[j][i];
  // Печать результата
  for(i=0;i<2;i++)
    printf("%d\t%d\t%d\n", AT[i][0], AT[i][1], AT[i][2]);
}
```

12. Перестановка элементов массива

Если необходимо поменять местами элементы массива, для промежуточного хранения данных требуется ввести вспомогательную переменную, имеющую такой же тип, какой имеют элементы массива.

Операция перестановки элементов массива с индексами *i* и *j* выполняется в следующем порядке:

1) значение элемента с индексом *i* записывается в вспомогательную переменную;
2) элемент с индексом *j* копируется в элемент с индексом *i*;
3) значение вспомогательной переменной записывается в элемент с индексом *j*.

Пример перестановки элементов *i* и *j* в одномерном массиве *A*:

```
int i, j, b, A[10];
...
b = A[i];
A[i] = A[j];
A[j] = b;
```

Перестановка элементов выполняется, например, при транспонировании квадратной матрицы, когда транспонированную матрицу записывают на место исходной (с целью экономии памяти). В этом случае переставляются местами числа, лежащие под главной диагональю и над ней, а сама главная диагональ служит осью симметрии (числа на этой диагонали остаются на своих местах – их переставлять не требуется).

Для выполнения перестановки используются два вложенных цикла. Введем следующие обозначения: *n* – порядок матрицы, *i* – номер строки, *j* – номер колонки. Тогда во внешнем цикле номер строки *i* должен изменяться от 1 до *n*-1, а номер колонки j во внутреннем цикле – от 0 до *i*-1.

13. Основные операции обработки изображений

Фотографии, чертежи, графики и другие изображения представлены в памяти компьютера как матрицы чисел и операции обработки изображений – это операции с матрицами. В 8-разрядных графических режимах каждой точке изображения соответствует один байт данных и изображение хранится как массив типа unsigned char размера $m \times n$. В 16-разрядных режимах каждой точке соответствует слово (тип unsigned), в 32-разрядных – двойное слово (тип unsigned long), однако изображение по-прежнему представляет собой матрицу $m \times n$.

Когда для обработки массива используются подпрограмма, в качестве параметра в нее передается только указатель на массив, т.е. многомерные массивы передаются как одномерные. При работе с двумерными массивами программисту приходится выполнять в подпрограмме дополнительные индексные вычисления для преобразования индексов двумерного массива в индекс одномерного.

Обозначим номер строки двумерного массива символом *y*, номер колонки – символом *x*, количество колонок массива (длину строки изображения) – символом *n*, номер позиции в соответствующем одномерном массиве – символом *i*. Тогда значения индексов одномерного и двумерного массивов связаны следующей формулой:

$$i = y \times n + x .$$

Однако такое преобразование требуется только в том случае, если в самой подпрограмме массив обрабатывается как двумерный.

Рассмотрим в качестве примера подпрограмму, которая преобразует позитивное черно-белое изображение с 256 градациями яркости в негативное:

```
void Negative(unsigned char *Image, int m, int n)
{
  int i;
  for(i=0; i<m*n; i++) Image[i] = ~Image[i];
}
```

В качестве параметров в подпрограмму передается указатель на массив Image, содержащий изображение, а также размеры этого массива – количество строк *m* и столбцов *n*. Изображение является двумерным массивом, но в подпрограмму передается как одномерный массив, и обрабатывается как одномерный массив, поэтому преобразования индексов не требуется.

Сама операция получения негативного изображения заключается в том, что прямой код числа, являющегося кодом яркости точки изображения, преобразуется в обратный код. Яркие точки в результате становятся темными и наоборот.

Операции отражения (слева направо и сверху вниз) и поворота (на 90° или 180°) сводятся к перестановке элементов матрицы изображения, для чего требуется организовать два вложенных цикла – по строкам (внешний) и по столбцам (внутренний).

Так как для хранения изображений требуется большой объем оперативной памяти, при выполнении отражения или поворота на 180° можно (с целью экономии) записывать результат в область памяти, в которой хранилось исходное изображение. Составление соответствующих алгоритмов требует определенной аккуратности: как и в случае транспонирования матрицы, нужно следить за тем, чтобы не переставить дважды одну и ту же пару элементов.

При отражении слева направо левая половина изображения меняется местами с правой (рис. 8), поэтому цикл по столбцам продолжается до $n/2$: колонка с номером x меняется местами с колонкой с номером $(n-x-1)$. Если изображение имеет нечетное количество колонок, колонку, лежащую на оси симметрии, переставлять не требуется.

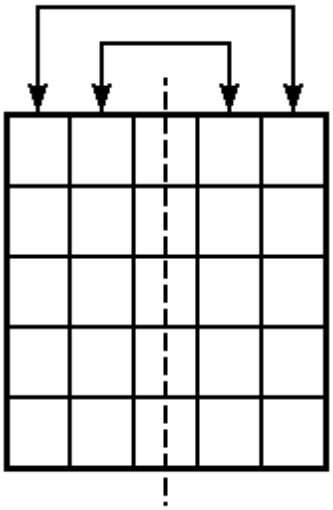

Рис. 8. Перестановка столбцов при отражении изображения слева направо

Пример подпрограммы, которая выполняет отражение изображения слева направо:

```
void LeftToRight(unsigned char *Image, int m, int n)
{
  int x, y;
  unsigned char b;
  for(y=0; y<m; y++) for(x=0; x<n/2; x++)
    {
    // Перестановка элементов массива
    b = Image[y*n+x];
    Image[y*n+x] = Image[y*n+n-x-1];
    Image[y*n+n-x-1] = b;
    }
}
```

При отражении сверху вниз верхняя половина изображения меняется местами с нижней (рис. 9) , поэтому цикл по строкам продолжается до $m/2$: строка с номером y меняется местами с строкой с номером (m-y-1). Если изображение имеет нечетное количество строк, строку, лежащую на оси симметрии, переставлять не требуется.

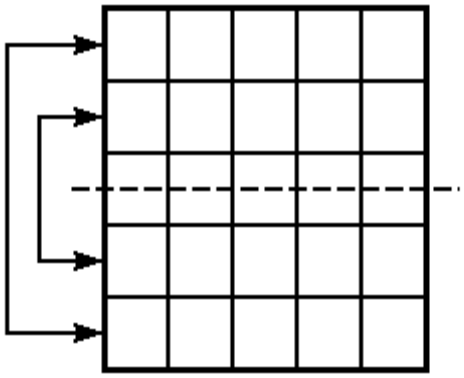

Рис. 9. Перестановка строк при отражении изображения сверху вниз

Поворот двумерного массива на 180° (рис. 10) эквивалентен отражению одномерного массива слева направо (рис. 11). Таким образом, с изображением можно работать, как с одномерным массивом: элемент с индексом i меняется местами с элементом с индексом $m \times n - i - 1$, а индекс i в цикле изменяется от 0 до $(m \times n)/2$.

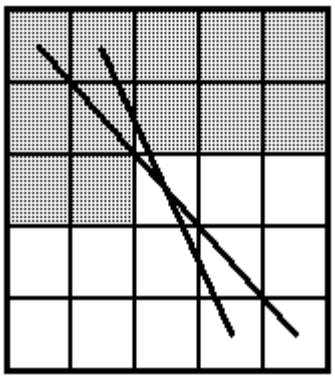

Рис. 10. Перестановка элементов изображения при повороте на 180°

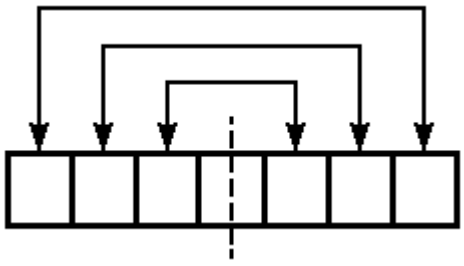

Рис. 11. Отражение одномерного массива

При повороте на 90° (рис. 12), если матрица не является квадратной (имеет размер $m \times n$), память сэкономить не удается – приходится создавать дополнительный массив размера $n \times m$ для сохранения результата. Индекс в этом случае последовательно пробегает все строки и столбцы исходного массива.

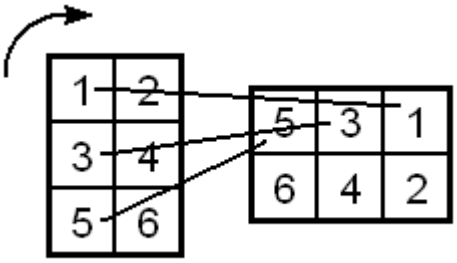

Рис. 12. Поворот изображения на 90° по часовой стрелке

Массив цветного изображения является трехмерным (первое измерение – строки изображения, второе – столбцы, третье – цветовые компоненты).

Для того, чтобы преобразовать цветное изображение из формата TrueColor в черно-белое с 256 градациями серого цвета, требуется для каждой точки вычислить усредненную яркость по всем цветовым компонентам.

Для вычисления средней яркости нужно вначале разделить яркость по каждой из трех цветовых компонент (R, G и B) на 3, а затем вычислить сумму. Поступать наоборот (вначале вычислить сумму, а затем разделить ее на 3) нельзя – может произойти переполнение разрядной сетки, что приведет к искажению изображения.

Масштабирование – это увеличение или уменьшение изображения.

Наиболее простая разновидность масштабирования – увеличение изображения в целое число раз. Для увеличения изображения в k раз каждая точка должна быть преобразована в квадрат размером k×k точек (рис. 13). Недостаток этого алгоритма – сильно выраженная ступенчатость изображения (лестничный эффект).

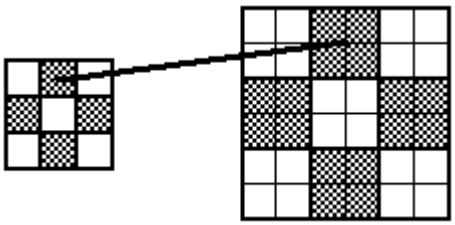

Рис. 13. Увеличение изображения в два раза

Чтобы уменьшить изображение в k раз, его требуется разбить на квадраты размером k×k точек (рис. 14). Яркость точки уменьшенного изображения вычисляется, как усредненная яркость всех точек соответствующего квадрата, т.е. сумма яркостей точек делится на k^2. Для цветного изображения усредненная яркость по каждой цветовой компоненте вычисляется отдельно. Недостаток этого алгоритма – теряются мелкие детали изображения.

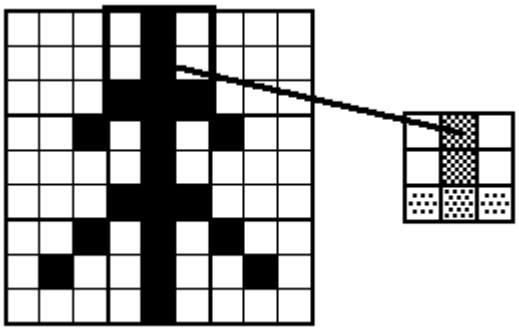

Рис. 14. Уменьшение изображения в три раза

При выполнении операции масштабирования требуется организовать **четыре** вложенных цикла. Для увеличения изображения нужно поочередно обработать все точки исходного изображения, т.е. требуются циклы по строкам и столбцам исходного изображения; кроме того, для копирования кода яркости точки нужны циклы по строкам и столбцам квадратика, в который эта точка преобразуется на увеличенном изображении. Для уменьшения изображения нужно вычислить яркость всех точек уменьшенного изображения, т.е. требуются циклы по строкам и столбцам уменьшенного изображения; кроме того, для определения яркости очередной точки уменьшенного изображения нужны циклы по строкам и столбцам квадратика исходного изображения, для которого вычисляется средняя яркость.

Вывод изображения на экран сводится к копированию данных из массива, содержащего изображение, в некоторую область видеопамяти. Для выполнения копирования требуется два вложенных цикла: внешний – по строкам изображения, внутренний – по столбцам. Видеопамять с точки зрения видеоконтроллера представляет собой двумерный массив, с точки зрения центрального процессора – одномерный массив, поэтому в процессе копирования потребуется выполнение преобразования координат двумерного массива в координаты одномерного.

14. Функций для генерации случайных чисел

Функции генерации случайных чисел определены в файле **stdlib.h**.

Функция **rand** использует мультипликативный генератор случайных чисел с периодом 2^{32}, чтобы получать числа в диапазоне от 0 до RAND_MAX ($2^{15} - 1$). Функция объявлена следующим образом:

```
int rand(void);
```

Пример программы, которая печатает на экране 10 случайных целых чисел в диапазоне от 0 до 99:

```
#include <stdio.h>
#include <stdlib.h>
void main(void)
{
  int i;
  printf("10 случайных чисел от 0 до 99:\n\n");
  for (i=0; i<10; i++) printf("%d\n", rand()%100);
}
```

Макрос **random** возвращает случайное число в диапазоне от 0 до (n – 1) . Функция объявлена следующим образом:

```
int random(int n);
```

Макрос **randomize** инициализирует генератор случайных чисел значением текущего системного времени. Функция объявлена следующим образом:

```
void randomize(void);
```

Инициализация генератора необходима в тех случаях, когда при каждом запуске программы требуется получить новую последовательность случайных чисел (если инициализация не производится, генератор выдает одну и ту же последовательность).

Вызов randomize производится в программе **только один раз** (перед началом цикла генерации случайных чисел).

Так как randomize использует вызов функции time, при использовании этой функции следует подключить также файл time.h.

Пример программы, которая печатает на экране случайную десятичную цифру:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
void main(void)
{
  randomize();
  printf("%d\n", random(10));
}
```

При каждом запуске эта программа выводит на экране новую цифру, а без использования randomize всегда выводилась бы одна и та же цифра.

Функция **srand** инициализирует генератор чисел заданным значением seed. Функция объявлена следующим образом:

```
void srand(unsigned seed);
```

15. Пузырьковая сортировка

Сортировка представляет собой процесс упорядочения множества подобных друг другу информационных объектов в порядке возрастания или убывания их значений.

Наиболее известной, простой и медленной является сортировка пузырьковым методом. Пузырьковая сортировка использует метод обменной сортировки – она основана на выполнении в цикле операций сравнения и при необходимости обмена соседних элементов. Ее название происходит из-за подобия процессу движения пузырьков в резервуаре с водой: вверх всплывают более легкие элементы, а самые тяжелые опускаются вниз.

Функция для сортировки элементов массива по возрастанию содержит два вложенных цикла. Внутренний цикл выполняет сравнение j-го элемента массива с элементом с индексом (j-1), и выполняет перестановку элементов, если j-й меньше (j-1)-го.

Внешний цикл повторяется (n-1) раз, где n – количество элементов массива. После первой итерации в конец массива будет перемещен элемент, имеющий максимальное значение. В следующий раз внутренний цикл продолжается до (n-1)-го элемента, так как последний элемент проверять не требуется, и с каждой итерацией внешнего цикла число операций в внутреннем сокращается.

Пример функции, реализующей пузырьковую сортировку целых чисел по возрастанию значений:

```
void Bubble(int *A, int n)
{
  int i, j, b;
  for(i=n; i>1; i--)
  {
    for(j=1; j<i; j++)
    if (A[j-1]>A[j]) {b=A[j]; A[j]=A[j-1]; A[j-1]=b;}
  }
}
```

В качестве параметров функция получает указатель на массив и количество элементов в массиве.

16. Быстрая сортировка

Функция **qsort** сортирует данные, применяя быстрый алгоритм сортировки.

Функция объявлена в файле stdlib.h следующим образом:

```
void qsort(void *base, size_t nelem, size_t width,
           int(*fcmp)(const void *, const void *));
```

Функция qsort сортирует содержимое таблицы постоянно вызывая функцию сравнения, определяемой пользователем и адресуемой с помощью указателя fcmp.

Параметры функции qsort:

- **base** – указатель на нулевой элемент сортируемой таблицы;
- **nelem** – число элементов таблицы;
- **width** – размер каждого элемента таблицы в байтах;
- **fcmp** – указатель на функцию сравнения.

Функция сравнения *fcmp получает два аргумента elem1 и elem2, которые представляют собой указатели на элементы таблицы. Функция сравнивает элементы *elem1 и *elem2 между собой и возвращает в зависимости от результата сравнения целое число (если *elem1 < *elem2 – отрицательное число, если *elem1 = *elem2 – ноль, если *elem1 > elem2 – положительное число).

Рассмотрим пример использования функции qsort для сортировки массива целых чисел по возрастанию:

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

int sort_function(const void *a,const void *b)
{
```

```
   return(*(int *)a - *(int *)b);
}

void main()
{
  int i, A[10];
  randomize();
  for(i=0; i<10; i++) A[i]=rand();
  printf("Массив случайных чисел:\n");
  for(i=0; i<10; i++) printf("%d\n",A[i]);
  qsort(A, 10, 2, sort_function);
  printf("Отсортированный массив:\n");
  for(i=0; i<10; i++) printf("%d\n",A[i]);
}
```

17. Функция time

Функция **time** объявлена в файле time.h следующим образом:

```
time_t time(time_t *tloc);
```

Функция time присваивает переменной, на которую указывает tloc, целочисленное значение, равное количеству секунд, прошедших от полуночи 1 января 1970 г, и возвращает это значение в качестве результата (тип time_t эквивалентен типу long).

При вызове функции time можно либо передать указатель на переменную, в которой требуется запомнить значение времени, в качестве аргумента, либо передать в качестве аргумента значение NULL, а в переменную занести возвращенное функцией значение.

18. Измерение производительности

Чтобы измерить время выполнения операции, требуется засечь время до и после выполнения операции и вычесть первое из второго.

Счетчик времени переключается один раз в секунду. Быстродействие современных персональных компьютеров очень велико, за секунду выполняется множество операций, и разность времени до и после операции, полученная с помощью функции time, обычно является нулевой. Следовательно, требуется изменить порядок проведения эксперимента: замерить начальное время, выполнить несколько тысяч или миллионов однотипных операций, а затем замерить конечное время.

Рассмотрим в качестве примера программу, которая измеряет интервал времени, необходимый для того, чтобы миллиард раз выполнить пустой цикл for:

```
#include <stdio.h>
#include <time.h>
void main(void)
{
  time_t Before, After;
```

```
  long i, DeltaT;
  (void)time(&Before);
  for(i=0; i<10000000001; i++); //Цикл задержки
  (void)time(&After);
  DeltaT = After - Before;
  printf("Результат: %ld секунд.\n", DeltaT);
}
```

В данном примере мы использовали первый способ обращения к функции time.

Производительность работы различных программных или аппаратных компонентов обычно оценивается по скорости выполнения каких-либо операций. Для измерения производительности требуется разделить количество выполненных операций на разность конечного и начального значений времени. Таким образом, производительность измеряется количеством операций в секунду.

Изменим предыдущий пример так, чтобы измерить производительность работы цикла for:

```
#include <stdio.h>
#include <time.h>
void main(void)
{
  time_t t0, t1;
  long i, dt;
  t0 = time(NULL);
  for(i=0; i<10000000001; i++); //Цикл задержки
  t1 = time(NULL);
  dt = t1 - t0;
  printf("Результат: %ld циклов в секунду.\n", 10000000001/dt);
}
```

В данном примере был использован второй способ обращения к функции time.

19. Структуры в языке Си

Не следует путать понятия структуры данных и структуры языка Си. Эти понятия связаны между собой следующим образом: структуры Си используются для построения различных структур данных.

Структуры Си – это наборы логически связанных переменных, объединенных под одним именем. Описание структуры начинается с ключевого слова **struct** и заканчивается **точкой с запятой**. В качестве примера приведем описание структуры, предназначенной для хранения координат точки изображения x и y:

```
struct Point {int x,y;};
```

На основе простых структур можно строить более сложные. Создадим в качестве примера структуру, описывающую отрезок прямой линии:

```
struct Line {
  unsigned char Color; // цвет линии
  struct Point p1, p2; // координаты начала и конца отрезка
  };
```

Для обеспечения доступа к отдельным элементам структуры используется **выражение выбора структурного элемента**, которое имеет две синтаксических формы:

```
выражение.идентификатор
выражение->идентификатор
```

В первой синтаксической форме выражение представляет собой структуру, во второй форме – указатель на структуру. Идентификатор именует элемент структуры.

Рассмотрим в качестве примера функцию, которая заполняет структуру Line набором определенных значений:

```
void NewLine(struct Line *L, unsigned char LineColor,
             int x1, int y1, int x2, int y2)
{
  L->Color = LineColor;
  L->p1.x = x1;
  L->p1.y = y1;
  L->p2.x = x2;
  L->p2.y = y2;
}
```

Параметры функции NewLine: L – указатель на структуру, описывающую отрезок; LineColor – цвет отрезка; x1, y1 – координаты начала отрезка; x2, y2 – координаты конца отрезка.

Для доступа к структурам p1 и p2 по указателю L в этом примере используется вторая форма выражения выбора структурного элемента, а для доступа к элементам структур – первая форма.

20. Обработка списков

Списком называется последовательность элементов x_1, …, x_n, где $n \geq 0$. В случае $n = 0$ список называется **пустым.**

Элементы списка могут иметь произвольную структуру. В частности, любой элемент может также представлять собой список, называемый в этом случае **подсписком**. Если список содержит подсписки, его называют **списковой структурой**. Список без подсписков называют **последовательным списком.**

Определенные позиции списка имеют специальное назначение. Например, первая позиция называется **заголовком** списка.

Метод программирования, в основе которого лежат операции с списками, называется **обработкой списков.**

Существует два основных вида списков: распределенные и связные.

Связным списком (рис. 15) называется список, элементы которого не обязательно расположены в памяти последовательно. Доступ к элементам организуется с помощью имеющегося в каждом элементе указателя, который содержит адрес следующего элемента в списке. Последний элемент списка имеет специальный «пустой указатель» (NULL), сигнализирующий о том, что других элементов в списке нет.

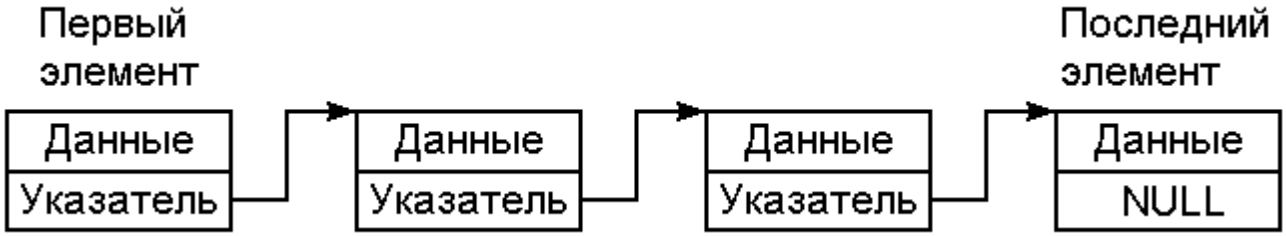

Рис. 15. Связный список

Обычно элемент связного списка содержит указатель на данные и указатель на следующий элемент. Иногда вместо указателя на данные в элементе списка могут содержаться непосредственно сами данные.

Связный список обеспечивает простоту выполнения операции вставки новых элементов. Операция вставки сводится к перенастройке указателей. Процедуре вставки передаются в качестве аргументов указатель на элемент, после которого производится вставка, и указатель на добавляемый элемент.

Допустим, что элемент С должен быть вставлен в список после элемента А. Действия должны выполняться в следующем порядке (рис. 16):

1) указатель на элемент В копируется из элемента А в элемент С;
2) адрес элемента С записывается в элемент А.

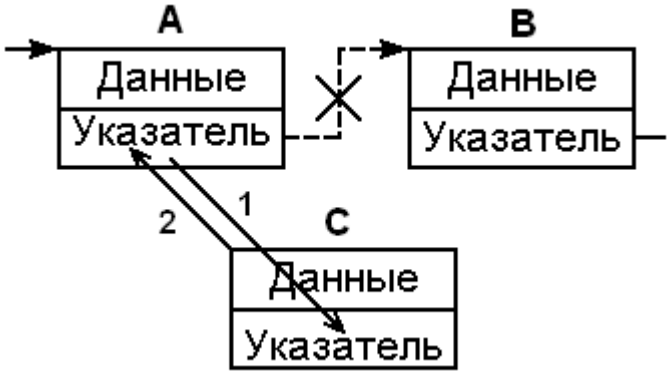

Рис. 16. Операция вставки элемента в связный список

Если А был последним элементом списка, в качестве указателя на следующий элемент в С будет скопировано значение NULL, и элемент С станет концом списка

Для выполнения вставки и удаления элементов обычно создаются специальные подпрограммы, а в качестве параметров передаются указатели на элементы.

Пример вызова процедуры, выполняющей вставку элемента в список:

```
#include <stdio.h>
// Структура, описывающая элемент связного списка
struct ListEl {
  char *Text;           // указатель на данные
  struct ListEl *Next; // указатель на следующий элемент
  };
// Вставить элемент El2 в список после элемента El1
void InsElement(struct ListEl *El1, struct ListEl *El2)
```

```
{
  El2->Next = El1->Next;
  El1->Next = El2;
}
// Главная функция
void main(void)
{
  struct ListEl El1, El2;
  char s1[]="Строка 1", s2[]="Строка 2";
  // Формируем первый элемент списка
  El1.Text = s1;
  El1.Next = NULL;
  // Формируем второй элемент списка
  El2.Text = s2;
  El2.Next = NULL;
  // Присоединяем второй элемент к первому
  InsElement(&El1, &El2);
}
```

Недостатком связного списка является сложность удаления элементов из списка: поиск предыдущего элемента приходится проводить последовательно от начала списка до текущего элемента.

Двунаправленный список (рис. 17) – усовершенствованный вариант связного списка, каждый элемент которого содержит по два указателя – на предыдущий и следующий элементы. У первого элемента списка указатель на предыдущий элемент является пустым, у последнего элемента списка указатель на следующий элемент является пустым.

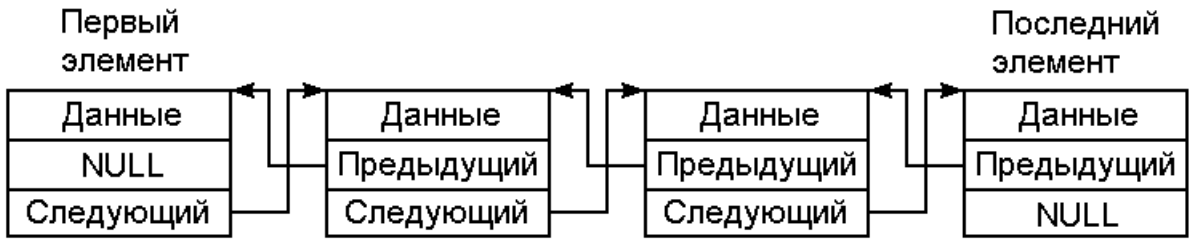

Рис. 17. Двунаправленный список

Например, для работы с текстом часто используют двунаправленные списки строк или абзацев. Элемент такого списка имеет следующую структуру:

```
struct ListEl {
  char *Text;          // указатель на данные
  struct ListEl *Prev; // указатель на предыдущий элемент
  struct ListEl *Next; // указатель на следующий элемент
  };
```

Как видно из описания элемента списка, он включает три указателя – указатель на массив, содержащий текстовую строку, указатель на предыдущий элемент Prev и указатель на следующий элемент Next.

Двунаправленный список позволяет легко добавлять и удалять элементы. Операция вставки сводится к перенастройке указателей (рис. 18).

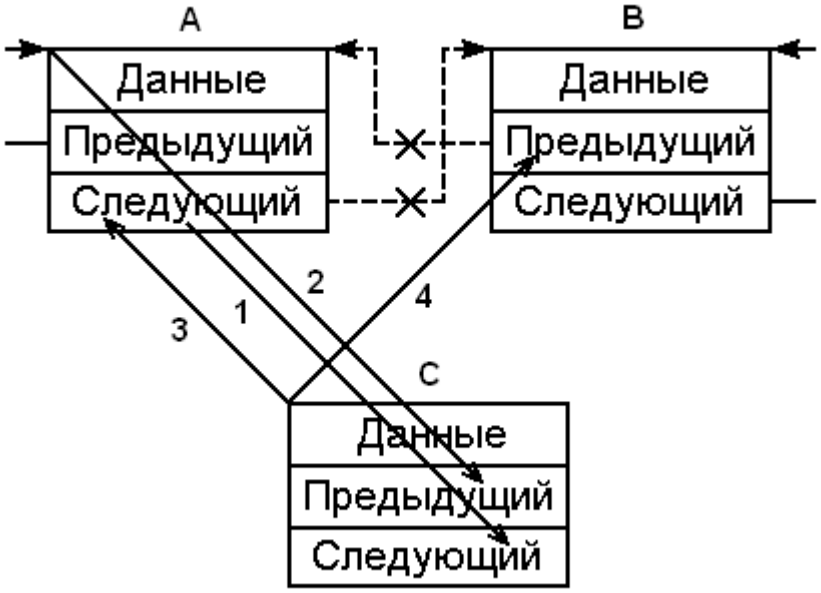

Рис. 18. Операция вставки элемента в двунаправленный список

Допустим, что элемент C должен быть вставлен в двунаправленный список после элемента A (рис. 17). Действия должны выполняться в следующем порядке:

1) указатель на элемент B копируется из элемента A в элемент C (элемент B становится следующим после C);
2) адрес элемента A записывается в элемент C (элемент A становится предыдущим для элемента C);
3) адрес элемента C записывается в элемент A (элемент C становится следующим после A);
4) адрес элемента C записывается в элемент B (элемент C становится предыдущим для элемента B);

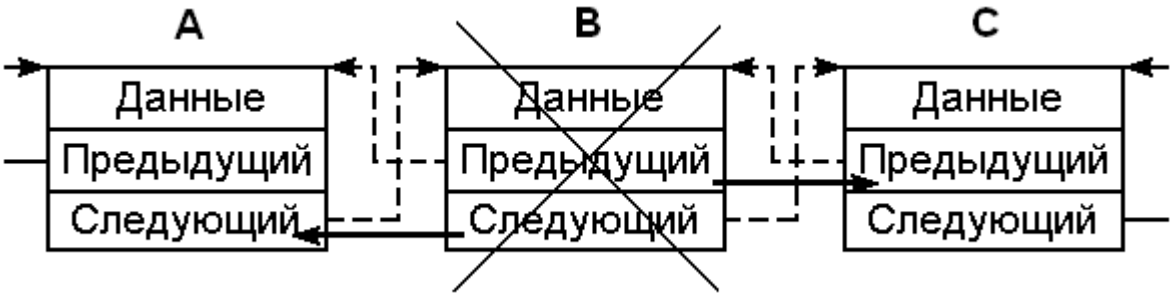

Рис. 19. Операция удаления элемента из двунаправленного списка

Если требуется удалить элемент B из двунаправленного списка (рис. 19), то указатель на предыдущий элемент копируется в элемент C, а указатель на следующий элемент – в элемент A (переписывать указатель нужно только в том случае, если он не равен NULL).

21. Очередь

Очередь – это список, в котором все дополнения вносятся в один конец, а выборка данных осуществляется с другого конца. Доступ к данным организован по принципу FIFO – «первым пришел, первым вышел».

Один из вариантов организации очереди – **кольцевой буфер** (рис. 20).

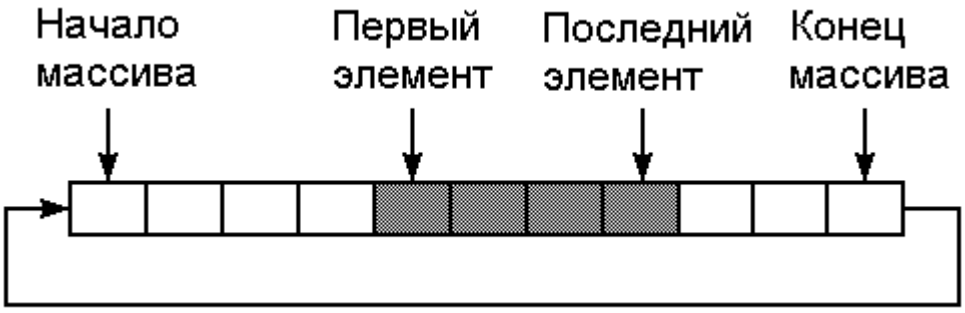

Рис. 20. Кольцевой буфер

Кольцевой буфер часто применяется в устройствах ввода-вывода для промежуточного хранения принимаемых или передаваемых данных. Кольцевой буфер, предназначенный для хранения целых чисел, можно представить, например, с помощью структуры следующего вида:

```
struct RingStruct {
  int First;      // индекс первого элемента
  int Last;       // индекс последнего элемента
  int Array[100]; // буфер
  } Ring;
```

Массив Array представляет собой собственно буфер, предназначенный для хранения чисел. Переменная First содержит номер первого элемента в очереди, а переменная Last – номер последнего элемента плюс 1 (т.е. указывает на первый свободный элемент кольцевого буфера).

Если очередь используются в программе в единственном экземпляре, ее объявляют как глобальную переменную (в данном случае переменной типа RingStruct присвоено имя Ring).

Инициализация очереди сводится к обнулению индексов первого и последнего элементов:

```
void InitRing(void)
{
  Ring.First=0;
  Ring.Last=0;
}
```

Так как Ring является глобальной переменной, передавать параметры в функцию InitRing не требуется.

При добавлении элемента число записывается в Array[Last] и значение Last увеличивается на единицу. Если достигнута верхняя граница массива (конец буфера), то следующий элемент заносится в начало буфера (значение Last

обнуляется) – поэтому буфер и называется кольцевым. Если после изменения значения Last оно становится равным значению First, то фиксируется ситуация переполнения буфера и выдается соответствующее сообщение об ошибке.

Рассмотрим пример программы, которая добавляет элемент в очередь:

```
void InsToRing(int a)
{
  Ring.Array[Ring.Last] = a; // Вставить элемент в буфер
  Ring.Last++;               // Передвинуть указатель
  // Если указатель вышел за границу буфера, обнулить его
  if(Ring.Last>=100) Ring.Last=0;
  // Проверка на переполнение
  if(Ring.First==Ring.Last)
  {
    printf("Переполнение буфера!\n");
    exit(1);                 // Аварийный выход из программы
  }
}
```

Функция InsToRing имеет только один аргумент – значение вставляемого в очередь элемента.

Использованная в конце примера функция **exit** служит для экстренного завершения работы программы. Функция описана в файле **stdlib.h** следующим образом:

```
void exit(int status);
```

Аргумент status передается операционной системе. Если он равен нулю, то считается, что программа завершилась успешно, иначе завершение считается аварийным. Для различных аварийных ситуаций можно использовать разные коды, чтобы по возвращаемому программой результату можно было определить причину ошибки.

В приведенном примере завершение в случае переполнения очереди пользователю выдается текстовое сообщение, после чего программа завершает работу аварийно с кодом 1.

При извлечении элемента число считывается из Array[First] и значение First увеличивается на единицу. Если достигнута верхняя граница буфера, то следующий элемент считывается из начала буфера (значение First обнуляется). Если значения First и Last равны, то очередь является пустой (не содержит данных): при попытке извлечения элемента из пустой очереди должно вырабатываться сообщение об ошибке.

Рассмотрим пример программы, которая извлекает элемент из очереди:

```
int GetFromRing(void)
{
  int a; // Вспомогательная переменная
  // Контроль опустошения буфера
  if(Ring.First==Ring.Last) exit(2); // Аварийный выход
  a = Ring.Array[Ring.First];        // Извлечение элемента
  Ring.First++;                      // Перемещение указателя
  // Если указатель вышел за границу буфера, обнулить его
```

```
  if(Ring.First>=100) Ring.First=0;
  // Вернуть извлеченное значение
  return a;
}
```

В случае возникновения ошибки опустошения буфера эта функция вызовет аварийное завершение программы с кодом 2.

22. Стек

Стек (рис. 21) – это линейный список, все записи в который вставляются и выбираются с одного конца, называемого **вершиной стека**. Доступ к данным организован по принципу LIFO – «последним пришел, первым вышел», т.е. последний вставленный в список элемент первым удаляется из списка. Вставка называется также проталкиванием элемента в стек, удаление – выталкиванием из стека.

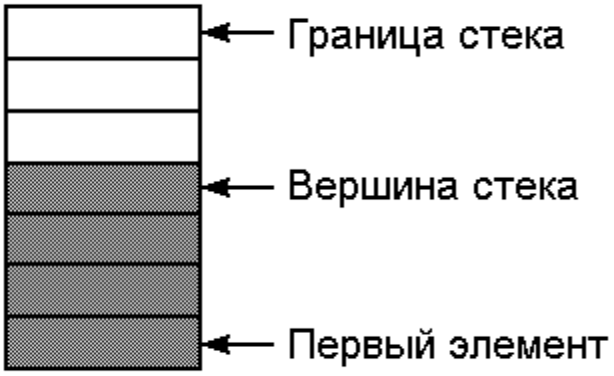

Рис. 21. Стек

Стек, предназначенный для хранения целых чисел, можно представить, например, с помощью структуры следующего вида:

```
struct StackStruct {
  int n;           // счетчик элементов
  int Array[100];  // буфер
  } Stack;
```

Переменная *n* хранит количество использованных элементов стека и указывает на первый свободный элемент (вершиной стека, таким образом, является элемент с номером *n*-1). Массив Array представляет собой собственно стек. Размер массива, таким образом, ограничивает размер стека.

Если стек используются в программе в единственном экземпляре, его объявляют как глобальную переменную (в данном случае переменной типа StackStruct присвоено имя Stack).

Если требуется вставить некоторое значение в стек, оно записывается в элемент Array[n], после чего *n* увеличивается на единицу. Если требуется извлечь значение из стека, то *n* уменьшается на единицу, после чего считывается значение Array[n].

При вставке элемента в стек необходимо контролировать верхнюю границу, а при извлечении – нижнюю (при выходе за границы стека должно выдаваться сообщение об ошибке).

23. Двоичные деревья

Дерево (рис. 22) – структура данных, состоящая из одного или нескольких **узлов** (вершин), связанных друг с другом отношением «предок-потомок». Каждая вершина может иметь несколько потомков, но только одного предка. Узел, не имеющий потомков, называется **листом**. Дерево имеет один особый узел, именуемый **корнем**, который является начальным и не имеет предков. Деревья принято рисовать в перевернутом виде (корень изображен сверху):

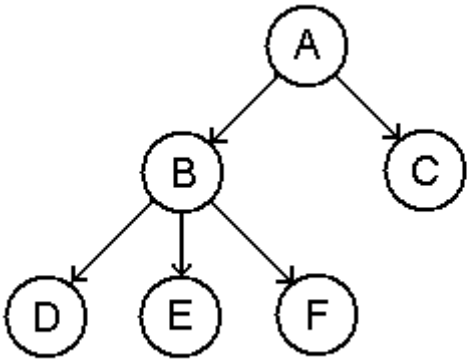

Рис. 22. Дерево

Соединения между узлами называют **ребрами** или **ветвями**. Дерево называется **упорядоченным**, если ветви, исходящие из каждой вершины, упорядочены по какому-либо правилу.

Уровень узла определяется следующим образом: корень имеет уровень 0, а уровень любого другого узла на 1 больше уровня его предка.

Глубина дерева – это максимальный уровень листа. Глубина равна длине пути от корня дерева до самого дальнего листа.

Дерево, вершины которого имеют не более двух потомков, называется **бинарным** (двоичным).

Обычно для описания вершин двоичного дерева используется структура вида:

```
struct TreeNode {
  char *Text;               // указатель на данные
  struct TreeNode *Father; // предок
  struct TreeNode *Left;   // левый потомок
  struct TreeNode *Right;  // правый потомок
  };
```

Полное бинарное дерево уровня *n* – это дерево, в котором каждый узел уровня *n* является листом, а каждый узел уровня меньше n имеет непустые левое и правое поддеревья.

Для работы с деревьями удобно использовать рекурсивные функции. **Рекурсивной** называют функцию, которая вызывает саму себя. Компилятор Си допускает любое число рекурсивных вызовов, но на практике оно ограничено размером стека. Так как каждый рекурсивный вызов требует дополнительной стековой памяти, то слишком много вызовов могут переполнить стек.

Рекурсию очень удобно применять, например, для определения глубины дерева или поиска данных в нем. Порядок обхода двоичного дерева при использовании рекурсивных вызовов функции поиска показан на рис. 23.

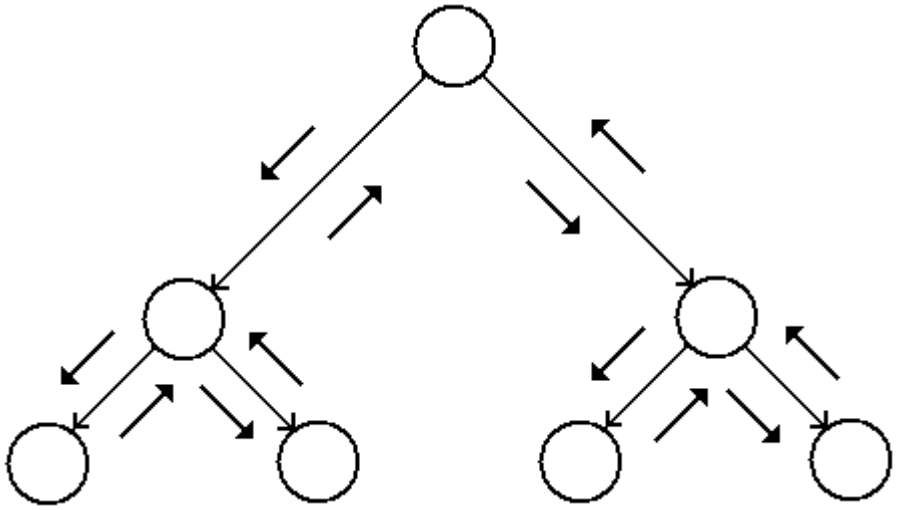

Рис. 23. Порядок обхода двоичного дерева

Рассмотрим функцию Deep, которая определяет глубину двоичного дерева:

```
void Deep(struct TreeNode *Node, int *k, int *d)
{
  // Если уровень находится ниже сохраненного значения
  // максимальной глубины, присвоить глубине новое значение
  if(*k>*d) *d = *k;
  // Опуститься на уровень ниже
  (*k)++;
  // Проверить левого потомка
  if(Node->Left!=NULL) Deep(Node->Left, k, d);
  // Проверить правого потомка
  if(Node->Right!=NULL) Deep(Node->Right, k, d);
  // Подняться обратно на верхний уровень
  (*k)--;
}
```

Функция Deep получает в качестве аргументов указатель на вершину Node, указатель на номер текущего уровня k и указатель на максимальную достигнутую глубину d.

Вначале функция сравнивает значения текущего уровня и максимальной глубины. Если текущий уровень оказывается ниже, то его значение присваивается в качестве нового значения глубины.

Далее производится спуск на следующий уровень. Функция работает только с указателями на потомков, игнорируя прочие данные. Если существует левый потомок, функция вызывается рекурсивно, а в качестве первого аргумента передается указатель на левого потомка. Когда функция получает управление обратно, она проверяет второго потомка. Если существует правый потомок, функция вызывается рекурсивно, а в качестве первого аргумента передается указатель на правого потомка. Затем выполняется возврат на текущий уровень.

Указатели k и d функция передает сама себе при каждом рекурсивном вызове.

При первом вызове функции в качестве первого аргумента передается указатель на вершину дерева; прежде чем передавать этой функции указатели на текущий уровень и глубину, значения соответствующих переменных в вызывающей программе нужно обнулить.

24. Распределение свободной памяти

Выделение памяти для элементов списка часто осуществляют динамически, т.е. память выделяется не сразу, а по мере добавления новых элементов. Если элемент списка содержит указатель на данные, а не сами данные, то прежде, чем добавить новый элемент в список, нужно также выделить в свободной памяти компьютера и место *для размещения данных*.

Выделение памяти осуществляется с помощью функций malloc и farmalloc, описанных в заголовочном файле **alloc.h**:

```
void *malloc(int size);
void far *farmalloc(unsigned long nbytes);
```

Функция **malloc** выделяет блок памяти размером size байт из локальной области памяти программы и возвращает указатель на выделенный блок. Если для размещения блока недостаточно памяти, функция возвращает NULL. Функция **farmalloc** выделяет блок размером nbytes в глобальной области памяти. Различие между функциями заключается в том, что локальная область памяти программы невелика (по сравнению с глобальной).

После завершения использования выделенного блока памяти его следует **освободить**. Для освобождения блоков, выделенных с помощью функций malloc и farmalloc используются функции **free** и **farfree** соответственно:

```
void free(void *block);
void farfree(void far *block);
```

Пример:

```
#include <stdio.h>
#include <alloc.h>
void main(void)
{
  char *str;
  str = malloc(100); // Выделить память под строку
  gets(str);         // Ввести строку с клавиатуры
  puts(str);         // Вывести строку
  free(str);         // Освободить память
```

```
}
```

При выделении памяти для структуры можно использовать оператор **sizeof** для вычисления ее размера в байтах:

```
struct ListEl *Elem;
Elem = malloc(sizeof(struct ListEl));
```

Рассмотрим в качестве следующего примера функцию, которая создает новый узел двоичного дерева, используя для выделения памяти функцию malloc:

```
void *Create(char *s)
{
  struct TreeNode *Node;
  // Выделить память для узла
  Node = malloc(sizeof(struct TreeNode));
  // Записать указатель на данные
  Node->Data = s;
  // Обнулить указатели на предка и потомков
  Node->Father = NULL;
  Node->Left = NULL;
  Node->Right = NULL;
  return Node;
}
```

В качестве аргумента функция Create получает указатель на данные, которые должны быть записаны в новом узле. Функция резервирует для нового узла некоторую область памяти, запоминает указатель на данные, обнуляет остальные указатели, после чего возвращает указатель на созданный узел в вызывающую программу.

Список использованной литературы

1. Абраш М. Таинства программирования графики. – К: ЕвроСИБ, 1996. – 384 с.: ил.
2. Белецкий Я. Энциклопедия языка Си: Пер. с польск. – М: Мир, 1992. – 687 с.: ил.
3. Керниган Б. Ритчи Д. Язык программирования Си: Пер. с англ. / Под ред. В.С. Штаркмана. – 2-е изд. – М: Финансы и статистика, 1992. – 272 с.: ил.
4. Цыпкин А.Г. Справочник по математике для средних учебных заведений. / Под ред. С.А. Степанова. – 3-е изд. – М: Наука, 1983. – 480 с.

Учебное издание

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ СИ.
УЧЕБНОЕ ПОСОБИЕ ПО ДИСЦИПЛИНЕ «ИНФОРМАТИКА»

СОСТАВИТЕЛЬ:
КУЛАКОВ ВЛАДИМИР ГЕННАДЬЕВИЧ

Редактор
Технический редактор
http://www.miem.edu.ru/rio
rio@miem.edu.ru

Подписано в печать . Формат 60x84/16.
Бумага типографская №2. Печать - ризография.
Усл. печ.л. Уч-изд.л. Тираж 50 экз.
Заказ - . Бесплатно. Изд № .
Московский государственный институт электроники и математики
109028 Москва, Б. Трехсвятительский пер. 3.
Отдел оперативной полиграфии Московского государственного
института электроники и математики.
113054 Москва, ул. М. Пионерская, 12.