

ГЛАВА 6. МАССИВЫ И СТРОКИ

Массивы являются наиболее часто используемыми структурами данных, представляющими собой совокупности элементов одного и того же типа, проиндексированных неотрицательными целыми числами.

Наиболее часто в программах применяются одномерные массивы и двумерные массивы (матрицы), но язык C++ позволяет работать и с многомерными массивами.

6.1. Одномерные массивы

Одномерный массив представляет собой несколько однотипных переменных, совместно использующих одно имя (имя массива), при этом доступ к каждому элементу осуществляется по его порядковому номеру (индексу).

Объявить вектор можно следующим образом:
тип_элементов имя_массива [число_элементов];

При этом число элементов должно быть задано явно либо числом, либо константой, так как компилятор резервирует место под массив в момент компиляции и изменить его размер потом уже невозможно. Допустимо лишь частичное использование занятого под массив адресного пространства.

Пример 6.1. Объявление массивов различных типов.

```
int i_arr [10];           // целочисленный массив из 10 элементов
char liter [80];          // символьный массив из 80 элементов
double d_mas [100];        // массив из 100 вещественных чисел
                           //двойной точности
```

При объявлении массивов необходимо помнить, что индекс первого элемента всегда равен нулю. Допустимыми считаются значения индексов, находящиеся в диапазоне от 0 до **число_элементов – 1**.

Обращение к элементу массива, индекс которого не является допустимым, приводит к возникновению ошибок, вызванных обращением к области памяти, не принадлежащей к массиву.

Доступ к элементам одномерного массива удобнее всего осуществлять, пользуясь циклом for.

Пример 6.2. Организация ввода элементов массива.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;
const int M = 10;
int _tmain(int argc, _TCHAR* argv[])
```

```
{double x[M]; //объявление действительного массива из 10 элементов
for (int i= 0; i<M; i++)
{ cout << "x[" << i << "] : "; // ввод элемента массива с клавиатуры
cin >> x[i];
}
// вывод массива на экран:
for (int i= 0; i<M; i++) cout << setw(5) << x[i];
cout << endl;
getch();
return 0;
}
```

Можно организовать ввод массива случайным образом в нужном диапазоне значений. Для этого используют генератор случайных чисел.

Пример 6.3. Ввод элементов вектора случайным образом и вывод его на экран.

```
#include "stdafx.h"
#include <iostream>
#include <time.h>
#include <conio.h>
using namespace std;
const int N = 100;
int _tmain(int argc, _TCHAR* argv[])
{
arr[N];
time t;
srand((int)time(&t)); // инициализация генератора случайных чисел
for (int i=0; i<N; i++) { arr[i]=rand()%100;
cout << arr[i] << ' ';
}
getch();
return 0;
}
```

Здесь формируется массив arr, состоящий из N элементов, пронумерованных от 0 до N–1 и содержащий целые значения от 0 до 99. При необходимости получения с равной вероятностью положительных и отрицательных значений элементов матрицы из генерированного значения вычитают число, равное половине аргумента функции rand() – random(100) – 50;

Пример 6.4. Дан 30-элементный целочисленный массив. Создать новый массив, расположив в нем сначала четные элементы исходного массива, затем нечетные.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;
const int M = 10;
int _tmain(int argc, _TCHAR* argv[])
```

```

const int M = 30;
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{time_t t;
srand((int)time(&t));
int i, j=0;
int mas1[M];           // исходный массив
int mas2[M];           // новый массив
for (i=0; i<M; i++) { mas1[i] = rand()%10;
    cout << mas1[i] << ' ';
}
cout << endl;
for (i=0; i<M; i++)
    if (mas1[i]%2==0) { mas2[j]=mas1[i];
        j++;
    }
for (i=0; i<M; i++)
    if (mas1[i]%2 != 0) { mas2[j]=mas1[i];
        j++;
    }
for (i=0; i<M; i++) cout << mas2[i] << ' ';
cout << endl;
_getch();
return 0;
}

```

Пример 6.5. Найти минимальный элемент и его индекс для 20-элементного массива.

```

#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
const int M = 20;
int _tmain(int argc, _TCHAR* argv[])
{ setlocale(LC_ALL, "Russian");
int i, min, i_min, mas[M];
for (i=0; i<M; i++) { mas[i] = rand()%10;
    cout << mas[i] << ' ';
}
cout << endl;
min = mas[0]; i_min = 0;
for (i=0; i<M; i++)
    if (mas[i] < min) { min = mas[i];
        i_min = i;
    }
cout << "min = " << min << ", индекс = " << i_min << endl;
_getch();
return 0;
}

```

6.2. Инициализация массива

Инициализация элементов массива осуществляется указанием его начальных значений в фигурных скобках:

```
int digits[10]={0,1,2,3,4,5,6,7,8,9};
```

Если при инициализации задано меньше начальных значений массива, чем их общее количество, остальные инициализируются нулевыми значениями. Другими словами, объявление

```
int digits[10]={0,1,2,3,4};
```

создает массив из 10 элементов целого типа, первые 5 из которых проинициализированы указанными значениями, а остальные – нулевыми значениями. Этим эффектом можно воспользоваться для инициализации элементов массива нулевыми значениями:

```
int digits[10]={0};
```

Однако если задать значений больше, чем число, указанное в квадратных скобках, компилятор генерирует сообщение об ошибке.

Во время инициализации можно задавать количество элементов массива таким образом, что невозможно будет впоследствии выйти за допустимые границы массива. Это достигается следующим образом.

Пусть MyMas – имя некоторого массива. Тогда sizeof(MyMas) – количество байтов, занимаемое этим массивом, а sizeof(MyMas[0]) – количество байтов, занимаемое одним элементом. Количество элементов в массиве можно определить как

```
int MyMas []={1,3,7,9,5,2,4,8,0,4,3,6}
#define N ((sizeof (MyMas)/sizeof (MyMas [0]))
```

Если использование #define нежелательно, можно выполнить те же действия путем определения константы:

```
#define int N=((sizeof (MyMas)/sizeof (MyMas [0]))
```

При необходимости можно задать массив констант следующим образом:

```
#define int ConstArr[]={5,4,3,2,1};
```

Тогда ни один из элементов массива изменить будет нельзя.

6.3. Методы сортировки массивов

6.3.1. Сортировка простым выбором

Наиболее простой из алгоритмов сортировки. В сортируемом массиве находится самый маленький элемент и обменивается местами с элементом в начале массива. Далее оставшаяся часть массива рассматривается как самостоятельный массив. В нем, в свою очередь, производится поиск минимального элемента, который меняется местами с первым эле-

ментом этого укороченного массива. Такие действия повторяются до тех пор, пока массив не укоротится до одного элемента.

Пример 6.6. Пусть дан массив 142, 23, 97, 19, 2, 4. Отсортируем его.

1-й проход: 2, 23, 97, 19, 142, 4
 2-й проход: 2, 4, 97, 19, 142, 23
 3-й проход: 2, 4, 19, 97, 142, 23
 4-й проход: 2, 4, 19, 23, 142, 97
 5-й проход: 2, 4, 19, 23, 97, 142

Программа сортировки:

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <time.h>
#include <conio.h>
using namespace std;
const int N = 6;
int _tmain(int argc, _TCHAR* argv[])
{ time_t t;
    srand((int)time(&t));
    int mas[N];
    int i, j, j_min, min;
    for (i = 0; i < N; i++) { mas[i]=rand()%100-50;
        cout << setw(4) << mas[i];
    }
    cout << endl;
    for (i=0; i<N-1; i++)
    { min = mas[i];
        j_min = i;
        for (j = i+1; j < N; j++) //перебор в уменьшенном массиве
            if (mas[j] < min) { min = mas[j];
                j_min = j;
            }
        mas[j_min] = mas[i];
        mas[i] = min;
    }
    for (i = 0; i < N; i++) cout << setw(4) << mas[i];
    cout << endl;
    _getch();
    return 0;
}
```

6.3.2. Метод пузырьковой сортировки

Под пузырьковой сортировкой понимают целый класс алгоритмов сортировки. В своем простейшем варианте пузырьковая сортировка выполняется крайне медленно, поэтому обычно применяют пузырьковую сортировку с элементами оптимизации. Однако все алгоритмы пузырько-

вой сортировки имеют общую особенность – обмен элементов массива производится между двумя соседними элементами массива.

Пример 6.7. Отсортируем данный массив 142, 23, 97, 19, 2, 4 пузырьковым методом.

1-й проход: 23, 142, 97, 19, 2, 4
 23, 97, 142, 19, 2, 4
 23, 97, 19, 142, 2, 4
 23, 97, 19, 2, 142, 4
 23, 97, 19, 2, 4, 142
 2-й проход: 23, 19, 97, 2, 4, 142
 23, 19, 2, 97, 4, 142
 23, 19, 2, 4, 97, 142
 3-й проход: 19, 23, 2, 4, 97, 142
 19, 2, 23, 4, 97, 142
 19, 2, 4, 23, 97, 142
 4-й проход: 2, 19, 4, 23, 97, 142
 2, 4, 19, 23, 97, 142

Программа сортировки:

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <time.h>
#include <conio.h>
const int N = 6;
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ time_t t;
    srand((int)time(&t));
    int mas[N], i, j, wrk;
    for (i = 0; i < N; i++) { mas[i]=rand()%100-50;
        cout << setw(4) << mas[i];
    }
    cout << endl;
    for (i = 0; i < N-1; i++) // i - счетчик проходов
        for (j = 0; j < N-1; j++) if (mas[j] > mas[j+1])
            { wrk = mas[j];
                mas[j] = mas[j+1];
                mas[j+1] = wrk;
            }
    for (i = 0; i < N; i++) cout << setw(4) << mas[i];
}
```

6.3.3. Метод пузырьковой сортировки с оптимизацией

Можно уменьшить количество проходов сортировки, выполняя их не $(N - 1)^2$ раз, а пока массив не будет отсортирован. Определить этот факт очень просто: если массив уже отсортирован, то в процессе прохода в нем не происходит никаких перестановок. Перед началом просмотра нужно установить признак (флаг) отсутствия перестановок. В случае, если производится хотя бы одна перестановка, флаг изменяет свое значение. Если к моменту завершения прохода значение флага осталось первоначальным, значит, массив отсортирован и дальнейшие проходы не нужны.

Пример 6.8. Сортировка массива 142, 23, 97, 19, 2, 4 пузырьковым методом с оптимизацией по количеству проходов.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <time.h>
#include <conio.h>
const int N = 6;
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ time_t t;
srand((int)time(&t));
int mas[N], i, j, wrk, flag;
for (i = 0; i < N; i++) { mas[i]=rand()%100-50;
cout << setw(4) << mas[i];
}
cout << endl;
flag = 1; // этот оператор нужен для входа в цикл
while (flag != 0)
{ flag = 0;
for (j = 0; j < N-1; j++)
{ if (mas[j] > mas[j+1])
{ flag++;
wrk = mas[j];
mas[j] = mas[j+1];
mas[j+1] = wrk;
}
}
for (j = 0; j < N; j++) cout << setw(4) << mas[j];
cout << endl;
_getch();
return 0;
}
```

Для оптимизации метода пузырьковой сортировки по времени выполнения каждого прохода можно использовать тот факт, что после первого прохода наибольший элемент окажется в конце массива, т. е. в предназначенном ему месте. В процессе выполнения второго прохода же самое происходит со следующим по величине элементом и т. д. По-

этому при последующих проходах можно уменьшать длину просмотра массива, что существенно сокращает общее время выполнения алгоритма. Если объединить этот метод оптимизации с проверкой признака завершения сортировки, получим алгоритм, называемый *обменной сортировкой с признаком завершения*.

Пример 6.9. Обменная сортировка массива 142, 23, 97, 19, 2, 4 с признаком завершения.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <time.h>
#include <conio.h>
const int N = 6;
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ time_t t;
srand((int)time(&t));
int mas[N], i, j, wrk, flag, k;
for (i = 0; i < N; i++) { mas[i]=rand()%100-50;
cout << setw(4) << mas[i];
}
cout << endl;
k = 1; // k - уменьшение длины просмотра
flag = 1; // оператор нужен для входа в цикл
while (flag != 0)
{ flag = 0;
for (j = 0; j < N-k; j++)
if (mas[j] > mas[j+1]) { flag++;
wrk = mas[j];
mas[j] = mas[j+1];
mas[j+1] = wrk;
}
for (j = 0; j < N; j++) cout << setw(4) << mas[j];
cout << endl;
}
return 0;
}
```

6.4. Двумерные массивы

Двумерные массивы – *матрицы* – представляют собой массивы одномерных массивов и задаются двумя числами в квадратных скобках:

двоеточие имя_массива [размер1] [размер2];

размер1 – количество строк; размер2 – количество столбцов.

Пример 6.10. Объявление матриц.

```
double matr [100][10];
int i_matrix [10][20];
```

В памяти ЭВМ матрицы занимают последовательно расположенные ячейки в следующем порядке: сначала располагается первая строка, за ней – вторая, третья и т. д.

Заполнение и обработку многомерных массивов обычно производят, пользуясь вложенными циклами. При этом массивы оказываются расположеными в памяти ЭВМ таким образом, что медленнее всего изменяется крайний левый индекс, а быстрее всего – крайний правый. Другими словами, при использовании вложенных циклов для обработки многомерных массивов самый внутренний цикл соответствует крайнему правому индексу, а самый внешний – крайнему левому индексу.

Пример 6.11. Ввод целочисленной матрицы с клавиатуры.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;
const int ROW = 5;
const int COL = 4;
int _tmain(int argc, _TCHAR* argv[])
{ int i, j;
  int x[ROW][COL];
  for (i=0; i< ROW; i++) // ROW - количество строк
    {for (j=0; j< COL; j++) // COL - количество столбцов
      { cout << "[" << i << "]" << j << "] " ;
       cin >> x[i][j];
     }
  }
_getch();
return 0;
}
```

Пример 6.12. Формирование матрицы с помощью генератора случайных чисел.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;
const int ROW = 5;
const int COL = 4;
int _tmain(int argc, _TCHAR* argv[])
{ int i, j;
  int x[ROW][COL];
  for (i = 0; i < ROW; i++)
    {for (j = 0; j < COL; j++)
      }
```

```
    { x[i][j] = rand()%10 - 5;
      cout << setw(4) << x[i][j];
    }
  cout << endl;
}
_getch();
return 0;
```

Вывод матриц также происходит с помощью вложенных циклов.

Пример 6.13. Вывод на экран двумерного массива.

```
for (i = 0; i < ROW; i++)
  {for (j = 0; j < COL; j++) cout << setw(4) << x[i][j];
  cout << endl;
}
```

Инициализация многомерных массивов осуществляется аналогично инициализации одномерных массивов. При этом инициализирующие значения должны совпадать с последовательностью хранения элементов массива в памяти.

Пример 6.14. Инициализация двумерного массива.

```
int x[5][3] = {{ 1, 2, 3}, // строка 1
                { 4, 5, 6}, // строка 2
                { 7, 8, 9}, // строка 3
                {10,11,12}, // строка 4
                {13,14,15}}; // строка 5.
```

Типичной ошибкой, не вызывающей сообщения компилятора, является объявление вида

```
int arr[i,j];
```

и использование arr[i,j] в выражениях. Компилятор воспринимает запятую между индексами как операцию «запятая» и игнорирует индекс i.

6.5. Обработка матриц

При работе с матрицами часто приходится решать задачи одного из трех типов:

работа с матрицей в целом;

работа со строками (столбцами) матрицы;

работа с диагональными элементами матрицы.

6.5.1. Работа с матрицей в целом

При работе с матрицей в целом вся подготовительная и инициализирующая работа выполняется один раз перед внешним циклом.

Пример 6.15. Поиск максимального элемента матрицы.

```
int arr[n][k], max;
max = arr[0][0];
for (i = 0; i<n; i++)
    for (j = 0; j<k; j++)
        if (max<arr[i][j]) max = arr[i][j];
```

6.5.2. Работа со строками/столбцами матрицы

При работе со строками инициализирующие действия производятся внутри внешнего цикла и до начала внутреннего цикла.

Пример 6.16. Поиск максимальных элементов каждой строки матрицы.

```
int arr[n][k];
int max[n];
for (i = 0; i<n; i++)
{ max[i] = arr[i][0];
    for (j = 0; j<k; j++) if (max[i]<arr[i][j]) max[i] = arr[i][j];
}
```

В результате этих действий формируется столбец, размерность которого равна числу строк в обрабатываемой матрице.

Работа со столбцами матрицы отличается от работы со строками только тем, что цикл обработки столбцов становится внешним и результирующий массив (строка) имеет размерность, равную количеству столбцов.

Пример 6.17. Поиск максимальных элементов каждого столбца матрицы.

```
int arr[n][k], max[k];
for (j = 0; j<k; j++)
{ max[j] = arr[0][j];
    for (i = 0; i<n; i++) if (max[j]<arr[i][j]) max[j] = arr[i][j];
}
```

6.5.3. Диагональные элементы матриц

При работе с диагональными элементами матрицы предполагают, что матрица квадратная (рис. 6.1).

Диагональ, соединяющая левый верхний угол матрицы с ее правым нижним углом, называется *главной*, а соединяющая правый верхний угол с левым нижним – *побочной*.

a ₀₀	a ₀₁	a ₀₂	a ₀₃	a ₀₄
a ₁₀	a ₁₁	a ₁₂	a ₁₃	a ₁₄
a ₂₀	a ₂₁	a ₂₂	a ₂₃	a ₂₄
a ₃₀	a ₃₁	a ₃₂	a ₃₃	a ₃₄
a ₄₀	a ₄₁	a ₄₂	a ₄₃	a ₄₄

Рис. 6.1. Квадратная матрица

Для элементов, стоящих на главной диагонали, выполняется равенство $i = j$. Если выполняется условие $i > j$, то элемент находится под главной диагональю. Если $i < j$ – над главной диагональю.

Элемент находится на побочной диагонали, если выполняется условие

$$i + n - j = 1,$$

где n – порядок матрицы; $i > (n - j - 1)$ – условие нахождения элемента на побочной диагональю; $i < (n - j - 1)$ – условие нахождения элемента на побочной диагональю.

Пример 6.18. Найти среднее арифметическое отрицательных элементов, расположенных под главной диагональю квадратной действительной матрицы 9×9 .

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <time.h>
#include <conio.h>
const ROW = 9;
const COL = 9;
namespace std;
int argc, _TCHAR* argv[];
if (argc != 2)
{
    cout << "Usage: <file>" << endl;
    exit(0);
}
if (fopen(argv[1], "r") == NULL)
{
    cout << "File " << argv[1] << " not found" << endl;
    exit(0);
}
double average = 0;
for (int i = 0; i < ROW; i++)
    for (int j = 0; j < COL; j++)
        if (i > j)
            if (mas[i][j] < 0)
                average += mas[i][j];
cout << "Average: " << average / (ROW * COL) << endl;
```

```

for (i=0; i<ROW; i++)
    for (j=0; j<i; j++) if (mas[i][j]<0)
        { n++;
        }
if (n>0)
    { average=average/n;
    cout <<"\nСреднее арифметическое= " << average << endl;
    }
    else cout <<"\nВсе элементы матрицы положительные" << endl;
_getch();
return 0;
}

```

Пример 6.19. Найти сумму четных элементов, расположенных под главной диагональю квадратной целочисленной матрицы 5×5 и над ее побочной диагональю (не включая диагональные элементы).

Для решения этой задачи необходимо анализировать одновременно три условия:

- расположение элемента под главной диагональю,
- расположение элемента над побочной диагональю,
- четность элемента.

С этой целью воспользуемся логической операцией «И» (`&&`). Двум первым условиям удовлетворяют элементы $a_{10}, a_{20}, a_{21}, a_{30}$ матрицы, представленной на рис. 6.1.

```

#include "stdafx.h"
#include <iomanip>
#include <stdlib.h>
#include <iostream>
#include <conio.h>
using namespace std;
const int RANG = 5 ;
int _tmain(int argc, _TCHAR* argv[])
{ int i, j, n=0, summa=0, mas [RANG][RANG];
for (i=0; i<RANG; i++)
    { for (j=0; j<RANG; j++) { mas[i][j]=rand()%20-10;
        cout << setw(6) << mas[i][j];
        }
    cout << endl;
    }
for (i=0; i<RANG; i++)
    for (j=0; j<RANG; j++)
        if ((i>j) && (i<RANG-j-1) && (mas[i][j]%2==0)) summa+=mas[i][j];
cout << "\nSumma = " << summa << endl;
_getch();
return 0;
}

```

В силу особенностей расположения матриц в памяти ЭВМ часто применяют способ обработки матриц с использованием лишь одного индекса, являющегося текущим индексом матрицы. Вычисляют его следующим образом. Если обозначить горизонтальный размер матрицы как N_x , абсциссу текущей точки массива как i_x , а ординату текущей точки массива как i_y , то текущий индекс матрицы i_a можно вычислить по формуле $i_a=N_x*i_y+i_x$.

6.6. Стока как массив символов

6.6.1. Ввод и вывод строк

Строка представляет собой массив значений типа `char`, завершающийся нулевым байтом.

Доступ к элементам строки осуществляется так же, как доступ к элементам любого массива:

```
char sym = str[2]; // переменная sym будет содержать 3-й элемент строки
// (счет идет с нуля)
```

Инициализация строки происходит следующим образом:

```
char HDD[]="Western Digital";
```

Только строка `HDD` имеет длину 16 байт, а не 15 – по числу символов, так как строки имеют невидимый нулевой завершающий байт. Поэтому, используя традиционные методы инициализации массивов, программы необходимо помнить о завершающем нулевом байте.

Таким образом, объявление

```
char MyABCs[]="ABC";
```

является точным эквивалентом объявления

```
char MyABCs[]={'A','B','C',0};
```

Для определения константы, равной длине инициализированной строкой переменной, можно воспользоваться функцией `sizeof()`:

```
int len_HDD = sizeof(HDD);
```

`HDD` – строка, инициализированная как «Western Digital»; `len_HDD` – константа, равная размеру массива в байтах, включая завершающий нулевой символ.

Пример 6.20. Объявление и ввод строки.

```

#include "stdafx.h"
#include <iomanip>
#include <stdlib.h>
#include <iostream>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ int i, j, n=0, summa=0, mas [5][5];
for (i=0; i<5; i++)
    for (j=0; j<5; j++)
        if ((i>j) && (i<5-j-1) && (mas[i][j]%2==0)) summa+=mas[i][j];
cout << "\nSumma = " << summa << endl;
_getch();
return 0;
}

```

```
{ const int MAX_LEN = 80; // максимальная длина строки
char str[MAX_LEN];
cout << "Input string : ";
cin >> setw(79) >> str;
cout << "Inputed string :" << str << endl;
_getch();
return 0;
}
```

Использование манипулятора setw(79) позволяет избежать переполнения буфера, отведенного под строку, поскольку ограничивает количество вводимых символов числом в скобках.

Операция >> вводит строку из (MAX_LEN-1) символов, не содержащую пробелов. Значащих символов в строке может быть не более, чем (MAX_LEN-1), так как строка обязательно содержит завершающий нулевой байт '\0', который служит признаком завершения строки. Особенность операции >> состоит в том, что пробел она воспринимает как нулевой символ, т. е. как признак конца строки.

Если строка содержит пробелы, то для ее считывания используется метод класса istream cin.get(). Мы пока не будем останавливаться на подробностях использования методов классов – речь об этом пойдет в разделе, связанном с методами классов. Для работы со строками достаточно знать, что оператор cin.get(str,MAX_LEN);

позволяет ввести с клавиатуры строку str, не превышающую (MAX_LEN-1) символов.

Пример 6.21. Ввод строки, содержащей пробелы.

```
#include "stdafx.h"
#include <iomanip>
#include <stdlib.h>
#include <iostream>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ const int MAX_LEN = 80; // максимальная длина строки
char str[MAX_LEN];
cout << "Input string : ";
cin.get(str,MAX_LEN);
cout << "Inputed string :" << str << endl;
_getch();
return 0;
}
```

Для ввода строки можно также воспользоваться методом getline.

Пример 6.22. Ввод строки, содержащей пробелы.

```
#include "stdafx.h"
#include <iomanip>
```

```
#include <stdlib.h>
#include <iostream>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ const int MAX_LEN = 80; // максимальная длина строки
char str[MAX_LEN];
cout << "Input string : ";
cin.getline(str,MAX_LEN);
cout << "Inputed string :" << str << endl;
_getch();
return 0;
}
```

Методы работают почти одинаково. Различие состоит в том, что метод getline считывает из входного потока не более (MAX_LEN-1) символов и записывает их в указанную строковую переменную, заменяя символ перевода строки \n нулевым байтом. Символ перевода строки при этом из потока удаляется.

Метод get оставляет символ перевода строки в потоке. Поэтому при попытке ввода нескольких строк, как показано в примере 6.23, будет введена лишь первая строка. Вместо всех остальных окажутся пустые строки, так как символ \n, оставленный в потоке методом get при вводе первой строки, будет обнаружен следующим методом get и интерпретирован как ввод пустой строки. При этом сам символ \n так и остается в потоке.

Пример 6.23. Попытка ввода нескольких строк.

```
#include "stdafx.h"
#include <stdlib.h>
#include <iostream>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ const int MAX_LEN = 80; // максимальная длина строки
char str[MAX_LEN];
cout << "Input string : ";
getline(str,MAX_LEN); // или так:
// cin.getline(str,MAX_LEN);
cout << "Inputed string :" << str << endl;
cout << "Input string : ";
getline(str,MAX_LEN);
cout << "Inputed string :" << str << endl;
cout << "Input string : ";
getline(str,MAX_LEN);
cout << "Inputed string :" << str << endl;
```

```
_getch();
return 0;
}
```

Удаление символа \n из потока может быть осуществлено вызовом метода get() без параметров.

Можно в примере 6.23 воспользоваться методом getline вместо метода get. Тогда эффекта потери строк не возникает и нет необходимости специально удалять символ \n из потока.

Современные версии языка C++ позволяют вводить несколько строк, пользуясь методом get с тремя аргументами: именем строковой переменной, длиной строки и символом завершения ввода. Тогда символ перевода строки не будет служить признаком завершения ввода строки. Теперь строки можно вводить до тех пор, пока не будет введен символ, объявленный признаком завершения ввода, или пока не будет превышен максимальный размер строки.

Пример 6.24. Ввод нескольких строк.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ setlocale(LC_ALL, "Russian");
const int MAX_LEN = 180; // максимальная длина строки
char str[MAX_LEN];
cout << "Введите строку: ";
cin.get(str,MAX_LEN,'*');
cout << "Введенная строка:\n" << str << endl;
_getch();
return 0;
}
```

В результате получится следующее:

Введите строку:
Якорь выбран чугунный,
И в открытых лагунах
Наши старые шхуны
Снова лижет прибой.
*

Введенная строка:
Якорь выбран чугунный,
И в открытых лагунах
Наши старые шхуны
Снова лижет прибой.

6.6.2. Функции для работы со строками

Пример 6.25. Поэлементное копирование строки.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
```

```
#include <string.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{const int MAX_LEN = 80; // максимальная длина строки
unsigned int i;
setlocale(LC_ALL, "Russian"); // русификатор
char s1[]="Ты навсегда в ответе за тех, кого приучил.";
char s2[MAX_LEN];
for (i = 0; i<strlen(s1); i++) s2[i] = s1[i];
// strlen(s1) - функция определения длины строки
s2[i] = '\0';
cout << s2 << endl;
_getch();
return 0;
}
```

Пример 6.26. Копирование строки с помощью функции strcpy().

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <string.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{const int MAX_LEN = 80; // максимальная длина строки
setlocale(LC_ALL, "Russian");
char s1[]="Нет в мире совершенства!";
char s2[MAX_LEN];
strcpy (s2,s1); // функция копирования строки s1 в строку s2
cout << s2 << endl;
_getch();
return 0;
}
```

Пример 6.27. Объединение строк с помощью функции strcat().

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <string.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{const int MAX_LEN = 80; // максимальная длина строки
setlocale(LC_ALL, "Russian");
char s1[]="Блестит над ледником холодная звезда, ";
char s2[]="но нет теплее на земле картины";
strcat (s1,s2); // функция объединения строк s1 и s2
cout << s1 << endl;
}
```

Пример 6.28. Поиск подстроки в строке с помощью функции strstr().

```
#include "stdafx.h"
#include <iomanip>
#include <iostream>
#include <conio.h>
#include <string.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{const int MAX_LEN = 80; // максимальная длина строки
setlocale(LC_ALL, "Russian");
char s1[]="Поиск подстроки в строке";
char s2[] = "строк";
char *s3 = strstr(s1,s2); // функция поиска в строке s1 подстроки s2
if (s3!=NULL) cout << s3 << endl;
else cout << "Подстрока не найдена" << endl;
_getch();
return 0;
}
```

Контрольные вопросы

1. Дать определение одномерного массива и матрицы.
2. Почему при объявлении статического массива его размер должен быть объявлен как константа?
3. Что происходит при обращении к элементу массива вне допустимого диапазона индексов?
4. Как выполнить инициализацию одномерного массива? Есть ли принципиальные отличия при инициализации матрицы?
5. Перечислите известные методы сортировки массива.
6. В чем заключается и чем обусловлена необходимость оптимизации пузырькового метода сортировки?
7. Что общего при работе с матрицей в целом и обработке ее отдельно взятых строк? В чем различия?
8. Как располагаются в оперативной памяти элементы массивов?
9. Что является условием нахождения элемента на главной диагонали матрицы? На побочной диагонали?
10. Дано:

```
const int MAX_ROW = 5;
const int MAX_COL = 4;
```

Все ли правильно в приведенных ниже фрагментах?

```
a) for (i = 0; i <= MAX_ROW; i++)
{for (j = 0; j <= MAX_COL; j++)
{x[i][j] = rand()%10 - 5;
}
}
```

- b) for (i = 0; i < MAX_ROW; i++)
{for (j = 0; j < MAX_COL; j++)
{x[i][j] = rand()%10 - 5;
}
}
- b) for (i = 1; i <= MAX_ROW; i++)
{for (j = 1; j <= MAX_COL; j++)
{x[i][j] = rand()%10 - 5;
}
}

Практические задания 6

1. Дан массив, содержащий 20 действительных чисел. Определить, сколько из них отличается от последнего элемента массива.
2. Найти максимальный и минимальный элементы 15-элементного действительного массива и их индексы.
3. Сдвинуть циклически элементы 20-элементного целочисленного массива на две позиции вправо.
4. Определить сумму и количество элементов массива, объявленного как int mas[20], попадающего в заданный с клавиатуры диапазон.
5. Не пользуясь дополнительным массивом, переписать массив float arr[10] с обратной стороны.
6. В действительной матрице 3×5 заменить нулями элементы, меньшие среднего арифметического значения этой матрицы, и единицы – большие.
7. Данна матрица 5×7 , элементами которой являются значения символьного типа. Составить одномерный массив, содержащий количество символов т в каждом из ее столбцов. Подсчитать количество символов а над третьей строкой матрицы и общее количество символов в и з под третьей строкой.
8. Данна действительная матрица 7×7 . Найти минимальное значение среди элементов, стоящих над главной диагональю, и максимальное значение среди элементов, находящихся ниже главной диагонали, а также их местоположение.
9. В квадратной действительной матрице 9-го порядка определить количество отрицательных элементов над побочной диагональю и сумму положительных элементов под главной диагональю.
10. Данна действительная матрица 6×8 . Найти сумму элементов каждой строки верхней половины матрицы и произведение элементов каждой строки ее нижней половины. Определить значение и местоположение максимального элемента верхней половины матрицы и минимального элемента ее нижней половины.
11. Данна целочисленная матрица 4×7 . Определить минимальное и максимальное значение матрицы и их местоположение. Найти среднее арифметическое значение положительных элементов и модуль от-

- рицательных элементов в каждом столбце матрицы. Результаты вычислений записать в одномерные массивы.
12. Данна вещественная квадратная матрица порядка 5. Получить новую матрицу путем прибавления к элементам каждой строки матрицы наименьшего значения элементов этой строки.

ГЛАВА 7. ТИПЫ ДАННЫХ,

СОЗДАЕМЫЕ ПОЛЬЗОВАТЕЛЕМ

(СТРУКТУРЫ, ОБЪЕДИНЕНИЯ, ПЕРЕЧИСЛЕНИЯ)

Несмотря на то что язык C++ предоставляет широкий спектр стандартных типов данных, иногда бывает удобно объединить переменные в более сложные конструкции, чем это предусмотрено языком программирования. Для этих целей в языке C++ существуют специальные средства, позволяющие программистам конструировать свои собственные типы данных. К этим средствам относятся:

- структуры,
- объединения,
- перечисления.

7.1. Структуры

Структура – это одна или несколько переменных одного или различных типов, которые для удобства работы сгруппированы в одну языковую конструкцию под одним именем. Структуры помогают в организации сложных данных, поскольку позволяют группу связанных между собой переменных (структурную) трактовать не как множество отдельных элементов, а как единое целое.

Для объявления *структур* (или *структурного шаблона*) используется ключевое слово *struct*, за которым следуют идентификатор, являющийся именем типа структуры, и список полей или членов структуры, заключенных в фигурные скобки. Каждый член объявляемой структуры, как и структура в целом, заканчивается точкой с запятой:

```
struct имя_структурь { тип имя_поля;  
    тип имя_поля;  
};
```

Например:

```
struct date { int day;  
    int month;  
};
```

После того как объявлен структурный тип, компилятор «знает», что программа создан некоторый собственный тип данных. Однако пользоваться этим типом нельзя до того момента, пока не объявлена переменная этого типа.

Для того чтобы получить возможность использования структуры, следует сначала объявить переменную структурного типа. Это можно сделать двумя способами:

```
1) date today; // объявили переменную структурного типа today
2) struct date { int day;
    int month;
} today;
```

В языке С слово `struct` перед объявлением структурной переменной было обязательным и первый способ объявления имел вид `struct date today`.

Второй способ одновременно объявляет структуру (структурный шаблон) и выделяет память под переменную структурного типа. При объявлении нескольких структурных переменных для одной структуры их следует разделять запятыми:

```
date d1, d2, d3;
```

В момент создания структурной переменной компилятор резервирует под нее место в памяти ЭВМ, т. е. создается реальный объект, который может участвовать в работе программы.

Доступ к членам структуры осуществляется записью вида

имя_структурной_переменной.имя_поля

Другими словами, если объявлена структурная переменная `today`, то можно записать

```
today.day = 12;
today.month = 4;
```

Структуры могут содержать в качестве своих полей или членов переменные любых типов, в том числе массивы или другие структуры.

Пример 7.1

```
struct bigstr { float f;           // поле структуры – переменная типа float
    char st [80]; // поле структуры – массив из 80 символов
    double d [25]; // поле структуры – массив из 25
                    // действительных чисел двойной точности
} BigStr;
```

7.1.1. Присваивание значений структурным переменным

Полям структурных переменных можно присваивать значения одним из трех способов: воспользоваться оператором присваивания, ввести значения полей с клавиатуры в диалоговом режиме или проинициализировать объявляемую структурную переменную.

Пример 7.2. Пусть описана структура `Addr`, содержащая 4 компонента адреса: город, улицу, дом, квартиру.

```
struct Addr{ char town[10];
    char street[10];
```

```
int block;
int flat;
};

Addr Home;
// Далее следует присваивание значений членам структуры
Home.town = "Vilnius";
Home.street = "Atejtes";
Home.block = 35;
Home.flat = 305;
cout << "The address is " << Home.town << Home.street << Home.block <<
Home.flat << endl;
// Ввод данных с клавиатуры
cout << "Город :";
cin >> Home.town;
cout << "Улица :";
cin >> Home.street;
cout << "Дом :";
cin >> Home.block;
cout << "Квартира :";
cin >> Home.flat;
cout << "The address is " << Home.town << Home.street << Home.block <<
Home.flat << endl;
```

Проинициализировать переменную структурного типа можно прямо в объявлении:

```
Addr Home = {"Vilnius", "Atejtes", 35, 305};
cout << "The address is " << Home.town << Home.street << Home.block
<< Home.flat << endl;
```

Очевидно, что порядок следования инициализаторов (присваиваемых значений) соответствует порядку объявленных в структуре членов.

7.1.2. Псевдонимы структур

При объявлении структуры часто используется ключевое слово `typedef`, которое создает **псевдонимы** типов данных. Необходимо помнить, что `typedef` не определяет нового типа данных, а только связывает обявление типа данных с именем, называемым **псевдонимом**:

typedef объявление Псевдоним;

Можно создать псевдоним для переменной любого типа, в том числе стандартного.

Обычно псевдоним начинается с большой буквы, но это не безусловное требование языка, а лишь соглашение.

Например, `typedef int Counter;` делает псевдоним `Counter` эквивалентом `int`, после этого можно делать объявления: `Counter i;`

При создании псевдонима структурной переменной допустимо отсутствие идентификатора структуры.

Пример 7.3

```
typedef struct date { int day;
                     int month;
} Date;
```

```
typedef struct { int day;
                 int month;
} Date;
```

В любом случае псевдоним Date также представляет объявленный структурный тип данных.

7.1.3. Операции, допустимые над переменными структурного типа

Чаще всего над переменными структурного типа выполняют операцию присваивания. Присваивание значений одной структурной переменной другой возможно, только если эти переменные имеют один и тот же тип:

```
Date d1, d2, d3;
```

```
.....
```

```
d1=d2;
```

Недопустимо присваивание структур разных типов, даже в том случае, когда они идентичны по количеству, типу и размеру их полей.

Пример 7.4. Пусть объявлены два идентичных структурных типа date1 и date2 и введены переменные объявленных типов d1 и d2.

```
typedef struct { int day;
                 int month;
} date1;
```

```
typedef struct { int day;
                 int month;
} date2;
```

```
date1 d1;
date2 d2;
d1=d2; // Нельзя!
        // Разные
        // типы
```

Несмотря на возможность присваивания значений однотипных структурных переменных, сравнивать их нельзя:

```
Date d1, d2;
```

```
...
```

```
if (d1==d2) оператор; // Нельзя: ошибка компиляции!
```

Побитовое сравнение не является надежным, так как различные компиляторы ведут себя по-разному. Единственный доступный способ достоверно сравнить структурные переменные – это их поэлементное сравнение:

```
if ((d1.day==d2.day)&&(d1.month==d2.month)) оператор;
```

7.1.4. Вложенные структуры

Если полем структуры является другая структура, то получается вложенная структура.

Пример 7.5. Объявим структуру для запоминания времени.

```
typedef struct time { int hour;
                      int minute;
                      int second;
```

```
} Time;
```

и структуру для запоминания даты:

```
typedef struct date { int day;
                      int month;
                      int year;
} Date;
```

Теперь можно построить вложенную структуру DateTime:

```
typedef struct datetime { Date today;
                           Time now;
                           } DateTime;
```

Для доступа к вложенным полям структур используют число точек, достаточное для осуществления такого доступа. Например:

```
DateTime dt;
```

тогда можно записать:

```
dt.today.year = 2003;
dt.now.minute = 27;
```

При этом слева от точки располагается имя структурной переменной, справа – имя поля структуры: dt – структурная переменная, today – поле структурной переменной dt; today – структурная переменная, year – ее поле; now – структурная переменная, minute – ее поле.

Структура не может объявить в качестве члена структуру своего типа, т. е. структура не может объявить членом саму себя, кроме случаев использования указателей.

Инициализация вложенных структур осуществляется с помощью конструкций вида

```
DateTime birthtime = {{17, 06, 1970}, {17, 30, 00}};
```

7.1.5. Массивы структур

Комбинация массивов и структур является довольно мощным средством для организации, например, баз данных.

Существует две основные комбинации:

- массивы структур;

- структуры с полями, являющимися массивами.

Описание массива структур совершенно аналогично описанию массива любого другого типа. Каждый элемент массива представляет собой структуру объявленного типа.

При определении элементов массива структур мы применяем те же правила, которые используются для отдельных структур: сопровождаем структуры операцией получения элемента и именем элемента:

```
Home[10];
```

Индекс массива присоединяется к имени массива структурного типа, а не к концу конструкции, осуществляющей доступ к полям структуры:

```
Homes.town[2]; // неправильно
Homes[3].street; // правильно
```

В то же время Homes[2].street[4] – вполне допустимая запись с точки зрения компилятора, так как street – тоже массив. При этом мы получаем доступ к 5-му элементу массива, street, который, в свою очередь, является членом 3-го элемента массива Homes. Таким образом, мы рассмотрели использование массива в качестве элемента структуры.

Полями структур могут быть также массивы и других типов. Можно даже объявить в качестве полей массивы других структур. Однако при этом следует помнить, что такие массивы занимают фиксированный объем памяти и любая неиспользуемая позиция массива приводит к большим и напрасным затратам памяти.

Пример 7.6. Дан массив структур, содержащий сведения об успеваемости по информатике группы из 20 студентов. Структура содержит следующие сведения: фамилию и инициалы студента, 4 оценки, отражающие его текущую успеваемость в течение семестра. Вывести на экран список неуспевающих студентов (имеющих хотя бы одну оценку <2>).

```
#include "stdafx.h"
#include <stdlib.h>
#include <iostream>
#include <conio.h>
using namespace std;
const int N = 20;
const int MAX_LEN=80;
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian"); //подключение русификатора
int i, flag;
struct student { char FAM[MAX_LEN];
    int inform[ 4 ];
} Gruppa[N];
cout << "Введите данные" << endl; //запись массива данных
for (i=0; i<N; i++)
{cout << "имя: ";
cin.getline(Gruppa[i].FAM, MAX_LEN);
for (int j=0; j<4; j++)
{ cout << j+1 << "-я оценка";//1,2,3,4
    cin >> Gruppa[i].inform[j];
}
cin.get(); //убирает перевод строки из потока ввода
}
// Теперь можно обработать введенный массив записей.
for (i=0; i<N; i++)
{flag=0; //считаем, что студент хороший (нет двоек)
for (int j=0; j<4; j++)
{if (Gruppa[i].inform[j]<2) flag++;
}
if (flag!=0) cout << Gruppa[i].FAM << " - " << flag << endl;
}
getch();
return 0;
}
```

```
if (Gruppa[i].inform[j]==2) flag++;
if (flag!=0) cout << Gruppa[i].FAM << " - " << flag << endl;
}
getch();
return 0;
}
```

7.1.6. Структуры с битовыми полями

Рассмотрим структуру с битовыми полями (рис. 7.1).

```
struct person
{
    unsigned int age;           // возраст
    unsigned int married;       // семейное положение
    unsigned int children;      // количество детей
    unsigned int ownflat;       // наличие собственной квартиры
}; // занимает по 2 байта на каждое поле
// (а в некоторых системах и по 4 байта).
```

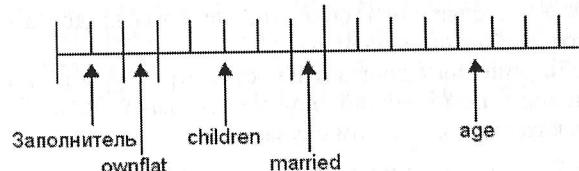


Рис. 7.1. Распределение памяти в структуре с битовыми полями

Запись структуры с битовыми полями позволяет выделить необходимое число битов на каждое поле структуры, указывая его через двоеточие после имени поля структуры. Неиспользуемое поле служит для выравнивания структуры до границы слова и называется **заполнителем**. Заполнителей может не быть, а может быть несколько в рамках одной структуры, если требуется, чтобы поля начинались с определенных битов.

```
struct person
{
    unsigned int age: 8;          // 1 – состоит в браке, 0 – нет
    unsigned int married: 1;       // 1 – есть своя квартира, 0 – нет.
    unsigned int children: 4;      // не используется
    unsigned int ownflat: 1;       // 1 – не используется
    unsigned: 2; // не используется
};
```

Следует обратить внимание на два момента:

- 1) программа, использующая структуру с битовыми полями, скорее окажется непереносимой с одного типа компьютера на другой;
- 2) использование структур с битовыми полями может вызвать снижение быстродействия работы программы.

7.2. Объединения

Весьма полезными могут оказаться средства, позволяющие экономить память за счет использования одной и той же области для хранения в различные моменты времени разных компонентов. В языке C++ для решения этой задачи используется конструкция, называемая *объединением*.

Объединения объявляются с помощью ключевого слова `union` и подобно структурам объединяют несколько полей. Но в структурах поля следуют друг за другом, а в объединении поля перекрывают друг друга, так как располагаются по одному и тому же адресу.

Синтаксис:

```
union имя_объединения { тип имя_поля;
    тип имя_поля;
    ...
};
```

Компилятор выделяет под объединение память, достаточную для размещения его наибольшего элемента.

Обычно объединения включают в состав структуры и задают специальный ее элемент, помогающий определить, какое поле объединения применяется в каждом конкретном случае.

Пример 7.7. Объявление и использование объединений.

```
union un { int i;
    float f;
    char c;
} my_u;

if (flag=='i'){
    my_u.i=25;
    cout << my_u.i << endl;
}
else if (flag=='f'){
    my_u.f=37.82;
    cout << my_u.f << endl;
}
else if (flag=='c'){
    my_u.c='*';
    cout << my_u.c << endl;
}
else
    cout << "Некорректно введен тип" << endl;
```

Объединения могут входить в структуры, и наоборот:

```
struct {
    char *string;
    char flag;
    union un { int i;
        float f;
        char c;
    } my_u;
} my_s;
```

Доступ к полям объединения осуществляется следующим образом: `my_s.my_u.f`

Подобно структурам объединения можно инициализировать в объявлении, но при этом можно задать значения только для первого члена объединения, т. е. в отличие от структуры нельзя проинициализировать сразу все значения объединения. Так как поля объединения начинаются по одному и тем же адресам, присваивание значения первому элементу одновременно является присваиванием значений всем другим элементам, перекрываемым этим значением. Однако это утверждение справедливо для объединений с полями одного и того же размера.

Анонимные объединения

C++ имеет особый тип объединений, называемый *анонимным объединением*. Анонимное объединение сообщает компилятору, что все переменные, являющиеся полями такого объединения, располагаются по одному и тому же адресу в памяти ЭВМ. Но, поскольку анонимные объединения не имеют имени созданного типа, нет возможности впоследствии объявить переменную такого типа. Поэтому обращение к полям анонимного объединения происходит без использования точки.

Пример 7.8

```
union { char sym;
    int i;
    float f;
    double d;
};

sym = '$';
i = 125;
f = 6.351;
d = 5.2e-8;
```

Фактически, анонимное объединение просто позволяет располагать несколько переменных по одному адресу.

7.3. Перечисления

Перечислимый тип данных создается пользователем и используется в тех случаях, когда переменная может принимать заранее известные значения и количество этих значений невелико.

Синтаксис объявления перечислимого типа имеет вид:

1) `стит перечислимый_тип {константы_перечислимого_типа};`

2) в следующем объявлении переменных перечислимого типа:

перечислимый_тип переменная1, переменная2, ..., переменнаяN;

3) `стит перечислимый_тип {константы_перечислимого_типа} список_переменных;`

Пример 7.9. Объявление перечислимого типа.

```
enum month {Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec}
birthdays;
```

C++ позволяет связать перечислимые идентификаторы с целыми числами:

```
enum cats {Cat = 1, Puma = 2, Lion = 3, Tiger = 10};
enum birds {stork = 1, lark, swallow, eagle};
```

Здесь константе lark будет присвоено значение 2, swallow – 3, eagle = 4.

Пример 7.10. При вводе номера месяца программа должна вывести дни рождения друзей, которые родились в этом месяце.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian");// подключение русификатора
typedef enum month {Jan = 1, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct,
Nov, Dec} Month;
int birth_mon;
cout << "Введи номер месяца :";
cin >> birth_mon;
switch (birth_mon)
{case Jan: cout << "Январь:" << endl;
cout << "17 - Анна Воронова" << endl;
cout << "22 - Олег Седов" << endl;
break;
case Feb: cout << "Февраль:" << endl;
cout << "2 - Станислав Алешкин" << endl;
break;
case Mar: cout << "Март:" << endl;
cout << "15 - Лариса Ткаченко" << endl;
cout << "30 - Инна Ярошенко" << endl;
break;
case Apr: cout << "Апрель:" << endl;
cout << "12 - Марина Андрианова" << endl;
break;
case May: cout << "Май:" << endl;
cout << "У меня нет друзей, родившихся в мае" << endl;
break;
case Jun: cout << "Июнь:" << endl;
cout << "1 - Игорь Егоров" << endl;
break;
case Jul: cout << "У меня нет друзей, родившихся в июле" << endl;
break;
case Aug: cout << "У меня нет друзей, родившихся в августе" << endl;
break;
case Sep: cout << "Сентябрь:" << endl;
```

```
cout << "13 - Наташа Гончарова" << endl;
cout << "30 - Татьяна Афанасьева" << endl;
break;
case Oct: cout << "Октябрь:" << endl;
cout << "10 - Юля Коротаева" << endl;
cout << "21 - Дима Волошин" << endl;
break;
case Nov: cout << "У меня нет друзей, родившихся в ноябре" << endl;
break;
case Dec: cout << "У меня нет друзей, родившихся в декабре" << endl;
break;
default: cout << "Некорректный ввод." << endl;
}
getch();
return 0;
}
```

Контрольные вопросы

1. Каким образом можно описать элемент данных типа структуры?
2. Как описать переменную структурного типа?
3. Как обратиться к члену структуры, если используются вложенные структуры?
4. Верно ли, что все поля структуры должны быть различных типов?
5. Что такое структура с битовыми полями? В чем преимущества и недостатки использования структур с битовыми полями?
6. Каков максимальный уровень вложенности структур?
7. Дать определение объединения.
8. Перечислить общие и отличные черты структур и объединений.
9. Может ли быть вложено объединение в структуру? А структура в объединение?
10. Что такое перечисление? Для чего оно используется?
11. Описать структуру для представления следующего понятия:
 - a) цена в рублях и копейках;
 - b) время в часах, минутах, секундах;
 - c) дата – число, месяц, год;
 - d) адрес – город, улица, дом, квартира.
12. Определить комбинированный тип (структурку) для представления анкеты школьника, включающей в себя его Ф.И.О., возраст, номер школы, класс, оценки по пяти каким-либо предметам. Описать переменную данного типа и присвоить ей значения, соответствующие следующей анкете: Петров Сергей Ильич, 15 лет, класс 8 «Б», оценки 5, 3, 4, 5, 5.
13. Придумать структурный шаблон, который будет содержать название месяца, трехбуквенную аббревиатуру месяца, количество дней в месяце и номер месяца. Определить массив, состоящий из 12 структур-

ных переменных объявляемого типа и инициализировать его для не-
високосного года.

Практические задания

1. Дан массив из 30 переменных структурного типа {житель: фамилия, город, улица, дом, квартира}. Написать программу, которая напечатает информацию обо всех однофамильцах, живущих:
 - а) в одном городе;
 - б) в разных городах.
2. Пусть структура описывает метеорологические данные за апрель 2001 г. и содержит следующую информацию:
 - область;
 - массив дневных температур;
 - массив ночных температур;
 - массив сведений об осадках (были или нет).
 Вывести на экран список дней, когда среднесуточная температура была положительной.
3. Данна ведомость (массив из 100 элементов), содержащая следующие сведения о сотрудниках кафедры:
 - фамилия, имя, отчество;
 - дата рождения;
 - стаж работы (год поступления на работу);
 - ученая степень;
 - ученое звание;
 - должность;
 Вывести на экран:

а) список сотрудников, возраст которых больше (меньше) чем введенно с клавиатуры;

б) список ветеранов (стаж больше 20 лет);

в) фамилии сотрудников, имеющих ученую степень и ученое звание, совпадающие с введенными с клавиатуры;

г) список сотрудников, занимающих определенную должность.

4. Дан массив из 50 элементов структурного типа, описывающего расписание отправления самолетов:

- номер рейса;
- пункт назначения;
- время вылета (часы и минуты), оформленное в виде вложенной структуры;
- время прибытия;
- цена билета;
- количество свободных мест.

Выбрать все рейсы, прибывающие в нужный пункт назначения с 10 до 12 ч, на которые есть билеты.

ГЛАВА 8. МАГНИТНЫЕ НОСИТЕЛИ ДАННЫХ.

ПОНЯТИЕ О ФАЙЛОВЫХ СИСТЕМАХ

8.1. Структура диска

В настоящее время наиболее распространенными носителями информации являются флоппи-диски (дискеты), жесткие диски (винчестеры) и лазерные диски (CD). Мы остановимся на рассмотрении магнитных носителей.

Независимо от вида накопителя (дискета или винчестер) хранение информации возможно благодаря наличию на нем магнитного слоя, расположенного в виде концентрических окружностей вокруг центра диска. Разница заключается в том, что дискета имеет две поверхности для записи информации, а в жестком диске пластины, напоминающие магнитный диск дискеты, объединены в пакеты таким образом, что имеют общую ось (рис. 8.1).

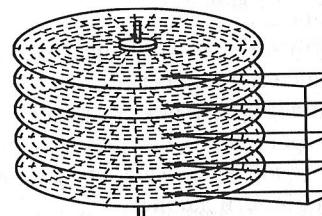


Рис. 8.1. Расположение головок чтения/записи
относительно магнитных поверхностей жесткого диска

Каждая из концентрических окружностей представляет собой дорожку. Дорожки делятся на сегменты, называемые секторами. В каждом секторе содержится 512 байт информации.

Диски вращаются вокруг оси, а головки чтения/записи движутся от внешнего края к центру, останавливаясь над дорожками. Достигнув нужной дорожки, головки чтения/записи останавливаются и ждут, когда под ними окажется нужный сектор, и затем приступают к операции чтения или записи.

В пакете магнитные поверхности нумеруются, как правило, начиная с 0.

0 – первая пластина,

1 – вторая пластина и т. д.

Головки располагаются одна над другой и жестко закреплены в этом пакете, поэтому перемещаются параллельно и одновременно оказы-

ваются над дорожкой с одним и тем же номером. DOS минимизирует медленные операции, такие, как перемещение головок, и стремится записать как можно большую часть файла при их фиксированном положении. Например, запись файла начинается с 10-й дорожки. При этом будет заполнена дорожка 10 на сторонах 0 и 1, затем – 2 и 3 и т. д. Когда 10-я дорожка на всех поверхностях окажется заполненной, головки будут передвинуты на 11-ю дорожку и процесс записи продолжится.

Все поверхности дорожки с номером N образуют цилиндр номер N.

Плотность цилиндра представляет собой число секторов, содержащихся в цилиндре, и равна числу секторов на дорожке, умноженному на число сторон пластин.

Плотность дорожек – это число концентрических дорожек на дюйм радиуса диска.

Очевидно, что при одинаковом количестве секторов плотность записи на сектор увеличивается при приближении к центру диска. Чтобы скомпенсировать эту неравномерность, используют метод зонно-секционной записи, заключающийся в том, что вся поверхность диска делится на зоны и во внешних зонах дорожки делятся на большее число секторов, чем во внутренних.

Следует понимать, что прежде, чем диск окажется способным осуществлять процедуры записи и чтения, его необходимо отформатировать. В процессе форматирования, кроме поиска дефектных блоков, происходит запись служебной информации и формирование следующих областей: загрузочной записи, одной или, чаще, двух таблиц размещения файлов, корневого каталога и области данных.

Файл (в данной интерпретации) – это цепочка секторов, заполненных данными.

8.2. Файловая система DOS.

Таблица размещения файлов

DOS содержит на диске списки файлов, называемые *каталогами*. Каталог содержит только адрес начала файла на диске, все остальное содержится в *таблице размещения файлов* (File Allocation Table, FAT). Это понятие тесно связано с понятием кластера.

Контроллер диска не может читать части секторов, поэтому DOS предоставляет дисковое пространство целыми секторами (блоками). Если размер файла находится между 1 и 512 байтом, ему отводится ровно 512 байт, если от 513 до 1024, то 1024, т. е. 2 сектора и т. д.

По одному сектору разделяют только некоторые гибкие диски, в то время как остальные диски выделяют пространство по 2, 4 или 8 секторов разделя. Эти минимальные единицы выделения дискового пространства называются *кластерами* (рис. 8.2).

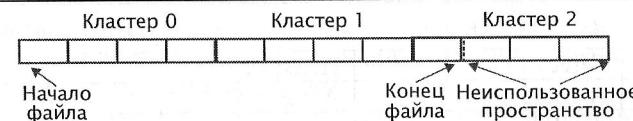


Рис. 8.2. Кластеры

Размер кластера не является свойством, присущим диску, это лишь метод, которым в DOS организуются данные. Размер кластера зависит от размера логического диска: чем больше диск, тем крупнее кластер. Поэтому очень большие диски лучше разбивать на более мелкие, так как иначе возникает эффект «потери дискового пространства», особенно если записывается большое количество маленьких файлов.

Каталог, как и файл, – это последовательность секторов, содержащих данные. Разница заключается в архитектуре этих данных и в их содержании. На каждую запись в каталоге отводится 32 байта, т. е. в каждом секторе помещается 16 записей каталога, или ячеек.

Каждая запись имеет структуру, показанную в табл. 8.1.

Таблица 8.1. Структура записи каталога

Байты	Содержимое
1–8	Имя файла
9–11	Расширение имени (если есть)
12	Атрибут файла
13–22	Зарезервированы
23–24	Время последней модификации файла
25–26	Дата последней модификации файла
27–28	Начальный кластер файла
29–32	Размер файла (в байтах)

Начальный кластер дает номер первого (и возможно, единственного) сектора, отведенного файлу. FAT указывает путь к следующему кластеру, занимаемому файлом, что очень актуально вследствие фрагментарного способа записи файла. Двенадцать байт – это байт атрибутов. Он может содержать следующую информацию (табл. 8.2).

Таблица 8.2. Атрибуты файлов

	Обычный файл
Read-Only	Только для чтения
Hidden	Скрытый
SysFile	Системный
Volume ID	Метка диска

Окончание табл. 8.2		
10h	Directory	Подкаталог
20h	Archive	Архивный
3Fh	AnyFile	Сумма всех предыдущих

FAT – самое важное, что есть на диске, поэтому обычно создается две такие таблицы – для надежности. Пользовательские программы могут иметь доступ к области FAT, но подобным программам не гарантируется совместимость с последующими версиями системы.

FAT – это таблица чисел, причем каждая позиция в таблице соотносится с кластером дискового пространства. Первая позиция представляет собой кластер 0, вторая – кластер 1 и т. д. Каждая позиция содержит номер того кластера, где продолжается файл. Например, файл начинается в кластере 100. DOS читает секторы, содержащиеся в кластере 100, затем в 100-й позиции FAT ищет номер следующего кластера. Пусть это будет кластер 105. Тогда DOS читает продолжение файла в кластере 105 и смотрит в позицию 105 FAT и т. д. (рис. 8.3). Когда DOS доходит до позиции FAT, содержащей специальный код, она знает, что достигнут конец файла. Коды элементов FAT представлены в табл. 8.3.

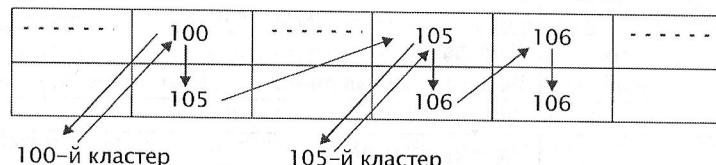


Рис. 8.3. FAT

Таблица 8.3. Коды элементов FAT

FAT		Описание
12-битовая	16-битовая	
000h	0000h	Кластер свободен
001h	0001h	Не используется
002h-FEFh	0002h-FFEFh	Номер следующего кластера
FF0h-FF6h	FFF0h-FFF6h	Резерв
FF7h	FFF7h	Сбойный кластер
FF8h-FFFh	FFF8h-FFFFh	Последний кластер файла

FAT располагается сразу за загрузочной записью и занимает разное число секторов. Первый элемент таблицы содержит дескриптор, используемый для идентификации диска. Для жестких дисков он равен F8h.

Второй байт обычно зарезервирован.

Третий элемент FAT содержит номер кластера начала области данных (002). Запись на диск выполняется необязательно последовательно. Первый свободный кластер, следующий за последним, выделенным

файла, будет следующим выделенным кластером независимо от его физического расположения на диске, т. е. при удалении файла просто освобождаются выделенные ему кластеры.

Каждый элемент FAT содержит 3 или 4 шестнадцатеричные цифры, в зависимости от элемента (12- или 16-разрядный); 16-разрядные FAT обычно используются на дисках, содержащих более чем 4 885 кластеров.

8.3. Файловая система NTFS

ОС Windows NT используют файловую систему NTFS.

Раздел NTFS теоретически может быть почти любого размера. Максимальный размер раздела NTFS в данный момент ограничен лишь размерами жестких дисков.

8.3.1. Структура раздела

Как и любая другая система, NTFS делит все доступное дисковое пространство на кластеры. NTFS поддерживает почти любые размеры кластеров – от 512 байт до 64 Кбайт. Стандартным считается кластер размером 4 Кбайт.

Диск NTFS условно делится на две части (рис. 8.4). Первые 12 % диска отводятся под так называемую MFT-зону – пространство, в котором размещается метафайл MFT. Запись каких-либо данных в эту область невозможна. MFT-зона по возможности содержит пустой. Это делается для того, чтобы самый главный служебный файл (MFT) не фрагментировался в процессе своего формирования. Физически фрагментирование метафайла MFT возможно, хотя и нежелательно.

Остальные 88 % диска представляют собой обычное пространство для хранения файлов.

Свободное место диска включает в себя всё физически свободное место – незаполненные участки MFT-зоны туда тоже включаются. Механизм использования MFT-зоны таков: когда файлы уже нельзя записывать в обычное пространство, MFT-зона просто сокращается (в текущих версиях ОС ровно в два раза), освобождая, таким образом, место для записи файлов. При освобождении места в обычной области MFT-зона может снова расширяться. При этом не исключена ситуация, когда в этой зоне окажутся и обычные файлы, но это не влечет за собой никаких проблем в работе ОС.

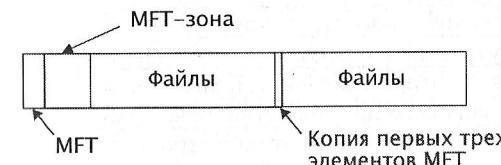


Рис. 8.4. Структура раздела NTFS

8.3.2. Структура MFT

Каждый элемент файловой системы NTFS представляет собой файл, в том числе служебная информация. Самый главный файл на NTFS называется MFT или Master File Table – общая таблица файлов. Именно он размещается в MFT-зоне и представляет собой централизованный каталог всех файлов диска. MFT поделен на записи фиксированного размера (обычно 1 Кбайт), и каждая запись соответствует какому-либо файлу. Первые 16 файлов недоступны ОС – они называются метафайлами, причем самый первый метафайл – сам MFT. Эти первые 16 элементов MFT – единственная часть диска, имеющая фиксированное положение. Существует вторая копия первых трех записей (для надежности), поскольку они очень важны. Она хранится ровно посередине диска. Остальной MFT-файл может располагаться, как и любой другой файл, в произвольных местах диска – восстановить его положение можно с помощью первого элемента MFT.

Метафайлы являются служебными файлами. Каждый из них отвечает за какой-либо аспект работы системы. Преимущество такого модульного подхода заключается в поразительной гибкости – например, физическое повреждение диска непосредственно в самой области FAT фатально для функционирования всего диска, а NTFS может сместить (даже фрагментировать по диску) все свои служебные области, обойдя любые неисправности поверхности, кроме первых 16 элементов MFT.

Метафайлы находятся в корневом каталоге NTFS диска, их имена начинаются с символа \$, хотя получить какую-либо информацию о них стандартными средствами сложно. В табл. 8.4 приведены используемые в современных ОС метафайлы и их назначение.

Таблица 8.4. Метафайлы и их назначение

Файл	Описание
\$MFT	Собственно MFT
\$MFTmirr	Копия первых 16 записей MFT, размещенная посередине диска
\$LogFile	Файл поддержки журналирования
\$Volume	Служебная информация – метка тома, версия файловой системы и т. д.
\$AttrDef	Список стандартных атрибутов файлов на томе
\$.	Корневой каталог
\$Bitmap	Карта свободного места тома
\$Boot	Загрузочный сектор (если раздел загрузочный)
\$Quota	Файл, в котором записаны права пользователей на применение дискового пространства (начал работать лишь в NT5)
\$Upcase	Файл – таблица соответствия строчных и прописных букв в именах файлов на текущем томе

8.3.3. Файлы и потоки

Понятие файла на NTFS включает в себя:

- Обязательный элемент – запись в MFT. В этом месте хранится вся информация о файле, за исключением собственно данных: имени файла, размера, положения на диске отдельных фрагментов и т. д. Если для информации не хватает одной записи MFT, то используются несколько, причем не обязательно подряд.
- Опциональный элемент – потоки данных файла. Если файл не имеет данных, на него не расходуется свободное место диска. Если файл имеет не очень большой размер, то содержимое файла хранится прямо в MFT, в оставшемся от основных данных месте в пределах одной записи MFT. Файлы, занимающие порядка 100 байт, обычно не имеют своего «физического» воплощения в основной файловой области – все данные такого файла хранятся в одном месте – в MFT.

Имя файла может содержать любые символы, включая полный набор национальных алфавитов, так как данные представлены в Unicode – 16-битовом представлении, которое дает 65 535 разных символов. Максимальная длина имени файла – 255 символов.

8.3.4. Каталоги

Каталог на NTFS представляет собой специфический файл, хранящий ссылки на другие файлы и каталоги. Таким образом формируется иерархия данных на диске. Файл каталога поделен на блоки, каждый из которых содержит имя файла, базовые атрибуты и ссылку на элемент MFT, который содержит полную информацию об элементе каталога. Внутренняя структура каталога представляет собой бинарное дерево. В отличие от линейного каталога (как у FAT) бинарное дерево располагает имена файлов таким образом, чтобы поиск файла осуществлялся более быстрым способом, называемым *дихотомией* (т. е. двоичным поиском).

Контрольные вопросы

- Каким образом выравнивают плотность записи на внешних и внутренних краях диска?
- Как избежать потери дискового пространства, вызванного слишком большим размером кластеров?
- Опишите алгоритм использования FAT при доступе к файлу.
- Почему операции чтения/записи организованы таким образом, что заполнение диска происходит по цилиндрам?
- Как вычислить плотность дорожек?
- Как размер кластера соотносится с размером магнитного носителя?

- пропорционален размеру носителя;
 - обратно пропорционален размеру носителя;
 - не зависит от размера носителя?
7. Сколько FAT обычно располагается на жестком диске?
8. Что такое NTFS?
9. Опишите структуру MFT. Зачем нужна копия первых записей MFT?

ГЛАВА 9. ОРГАНИЗАЦИЯ ВВОДА И ВЫВОДА. ФАЙЛОВАЯ СИСТЕМА

9.1. Стандартные файлы ввода и вывода

Чаще всего под *файлом* понимают именованную целостную совокупность данных на внешнем носителе. Другим часто встречающимся определением файла является его представление как последовательности или множества однотипных записей [5].

Однако понятие файла может быть и гораздо шире. В системе MS-DOS файлами считают коммуникационные порты, устройства печати и т. д. Операции ввода-вывода в языке С осуществляются через потоки. *Поток* – это логическое устройство, выдающее и принимающее информацию [25]. Клавиатура и экран дисплея называются *стандартными потоками ввода-вывода* и интерпретируются как текстовые файлы.

С потоком связано понятие внутреннего указателя, который определяет позицию, с которой начинается следующая операция чтения или записи. При каждой операции чтения или записи происходит автоматическое перемещение указателя.

В языке С (C++) формат стандартных файлов ввода-вывода описан в заголовочном файле stdio.h. Имена стандартных файлов ввода-вывода для языка С (C++) представлены в табл. 9.1. В момент начала выполнения программы на языке С (C++) автоматически открываются три потока: stdin, stdout, stderr.

Таблица 9.1. Потоки, определяемые в языке С (C++)

Стандартный файл	Описание
stdin	Последовательный ввод-вывод
stderr	Выходной поток ошибок
stdin	Стандартный ввод
stdout	Стандартный вывод
stderr	Вывод на принтер

C++ поддерживает всю систему ввода-вывода С и добавляет к ней дополнительные возможности, связанные в основном с вводом-выводом объектов. Описание средств для создания потоков в C++ представлено в заголовочном файле iostream.h. Когда начинает работать программа на C++, открываются потоки, приведенные в табл. 9.2.

Таблица 9.2. Потоки, определяемые в языке C++

Стандартный файл	Описание
cin	Стандартный ввод – клавиатура
cout	Стандартный вывод – экран
cerr	Стандартная ошибка – экран
clog	Буферизованная версия cerr – экран

Файловая система языков С и С++ состоит как бы из двух уровней: логических файлов и физических файлов, с которыми логические файлы всегда связаны.

Логический файл описывается как указатель на открываемый поток FILE * и служит средством взаимодействия с физическим файлом. Имя физического файла появляется в программе всего один раз, в тот момент, когда происходит открытие файла, осуществляемое функцией fopen() и одновременно его связывание с логическим файлом.

Основными действиями, производимыми над файлами, являются их открытие, обработка и закрытие. Обработка файлов может заключаться в считывании блока данных из потока в оперативную память, запись блока данных из оперативной памяти в поток, считывание определенной записи данных из потока, занесение определенной записи данных в поток. При этом необходимо помнить, что понятие файла в памяти ЭВМ не определено и приобретает смысл только после его связи с внешним физическим файлом.

9.2. Текстовые файлы

Тип FILE определяется в заголовочном файле stdio.h и обычно представляет собой структуру, содержащую параметры реализации потока, такие, как адреса буферов, указатели позиций потока, маркеры ошибок потока и т. д. При работе с дисковыми файлами в момент их открытия следует задать режим доступа, чтобы определить, к какому файлу осуществляется доступ: к текстовому или к двоичному, а также способ доступа: чтение или запись. Все это выполняется функцией fopen(), имеющей синтаксисы:

fopen("имя_файла", "режим_доступа") – для компиляторов более ранних версий;

fopen_s(&файловая_переменная, "имя_файла (путь к файлу)", "режим_доступа") – для современных компиляторов.

Режимы доступа к файлам для функции fopen() приведены в табл. 9.3.

Таблица 9.3. Режимы доступа к файлам

Режим	Описание
r	Открыть файл только для чтения, модификации файла запрещены
w	Создать новый файл только для записи. При попытке открыть таким способом существующий файл происходит перезапись файла. Чтение данных из файла запрещено
a	Открыть файл для дозаписи. Если файла с указанным именем не существует, он будет создан
r+	Открыть существующий файл для чтения и записи
w+	Создать новый файл для чтения и записи
a+	Открыть существующий файл для дозаписи и чтения

Для работы с текстовым файлом можно к режиму доступа добавить строчную латинскую букву t, но делать это не обязательно, так как файлы открываются в текстовом виде по умолчанию.

Таким образом, чтобы открыть текстовый файл, например для чтения, нужно произвести следующие действия:

```
FILE *ft; // объявили указатель на файловый поток
ft = fopen("inp_f.txt", "r"); // открыли файл inp_f.txt
```

При попытке открыть существующий файл можно допустить ошибку в его имени или в указании пути к нужному файлу. Это вызывает ошибку неполнения программы. Следует предвидеть подобные ситуации и проводить проверку возможности открытия файла. Такую проверку осуществлять довольно легко, так как функция fopen() возвращает значение указателя в случае успешного его открытия и значение NULL, если доступ к файлу невозможен. Поэтому достаточно написать

```
if (ft = fopen("inp_f.txt", "r")) != NULL
    // обработка файла
```

Файлы, открываемые в режиме записи, также нуждаются в аналогичной проверке.

Текстовый файл состоит из последовательности символов, разбитой на строки путем использования управляющего символа \n. На диске текстовые файлы хранятся в виде сплошной последовательности символов, разделение на строки становится заметным лишь в момент вывода на экран или на печать, так как именно при выводе управляющие символы начинают выполнять свои функции. Текстовые файлы легко переносятся с одного типа компьютера на другой лишь в случаях, когда они содержат управляющие символы, принадлежащие к стандартному набору символов.

При работе с текстовыми файлами возможна их посимвольная или строчная обработка.

9.3. Основные методы обработки текстовых файлов

Файловые функции ввода-вывода `fprintf()` и `fscanf()` работают аналогично функциям `printf()` и `scanf()`, но имеют дополнительный аргумент, являющийся указателем на файловый поток.

Пример 9.1. Чтение одного элемента из файла, обработка и запись результата в текстовый файл.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian");           //подключение русификатора
FILE *f; // объявили файловую переменную
int dat;
fopen_s(&f, "d:\\input.txt", "r");        // открывает файл для чтения
if ((f != NULL)// если файл открылся
{fscanf_s(f, "%d", &dat);               //считать значение dat из файла
fclose(f);                            // закрыть входной файл
fopen_s(&f, "d:\\outp.txt", "w");        // открывает файл для записи
printf("Мы прочитали число %d", dat);   // на экран
fprintf(f, "Мы прочитали число %d", dat); // в файл
fclose(f); // закрыть выходной файл
}
else printf("File Not Opened\n");
_getch();
return 0;
}
```

Мы использовали один и тот же указатель на файловый поток дважды, так как прежде, чем обращаться к нему вторично, закрыли связанный с ним файл и освободили таким образом указатель.

В приведенном примере имя файла было записано непосредственно в операторе открытия файла. Но можно сообщить имя открываемого файла, введя его с клавиатуры.

Пример 9.2. Написать программу, которая сжимает содержимое файла, записывая в выходной файл лишь каждый третий символ из входного файла.

```
#include "stdafx.h"
#include <iostream>
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{FILE *f_in, *f_out; // объявили две файловые переменные
```

```
char ch;                                // счетчик элементов
int count = 0;                           // открыть файл для чтения
f_in=fopen("d:\\input.txt", "r");         // если файл открыт корректно
if((f_in != NULL)
{fopen_s(&f_out, "d:\\outp.txt", "w"); // открыть файл для записи
while((ch = fgetc(f_in)) != EOF)
    if (count++ % 3 == 0) fputc(ch, f_out); // записывать в выходной файл
    // каждый третий символ
fclose(f_in);                          // закрыть входной файл
fclose(f_out);                         // закрыть выходной файл
}
else printf("File Not Opened\n");
_getch();
return 0;
```

При работе с текстовыми файлами возможна не только поэлементная обработка файлов, но и построчная.

Пример 9.3. Построчное чтение информации из входного файла и вывод ее на экран как на стандартное устройство вывода [16].

```
#include "stdafx.h"
#include <iostream>
#include <stdio.h>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{FILE *f_in; . . .
char buffer[256]; // максимальная длина строки – 255 символов
char name[80]; // имя входного файла
cout<<"Input File Name";
cin>>name;
if ((f_in = fopen(name, "r")) != NULL)
{ while (fgets(buffer, 255, f_in) != NULL)
    { fputs(buffer, stdout);
    fputc('\\n', stdout);
    }
fclose(f_in);
}
else cout<<"File Not Opened\n";
return 0;
```

В цикле `while` присутствует две файловые функции работы со строками `fgets()` для чтения строки символов в буфер и `fputs()` для записи содержимого буфера в файл.

Закрыть файл не менее важно, чем открыть его, так как в этот момент происходит заполнение ячейки таблицы размещения файлов значением, которое является признаком завершения файла, и установка атрибутов файла.

Кроме того, вывод текстового файла буферизован. Это значит, что в тот момент, когда работает оператор записи в файл, фактическая запись может и не происходить, поскольку реально сначала происходит заполнение данными текстового буфера, а потом его содержимое записывается на диск. Запись буфера происходит как только он окажется полностью заполненным или при выполнении специальных команд принудительной записи на диск. Процесс записи содержимого буфера на диск называется *флешированием* и обычно выполняется с помощью функции `fflush(f_out)`. При необходимости завершить работу сразу со всеми открытыми файлами пользуются функцией `flushall()`.

Закрытие файла посредством функции `fclose(f_out)` также включает процесс флеширования, т. е. перенос информации из буфера на диск.

Доступ к элементам текстовых файлов возможен только в последовательном режиме как при записи файла, так и при его чтении.

Пример 9.4. Записать в текстовый файл таблицу значений x и y . Считать из файла находящиеся там данные и умножить считанные значения x на 2, значения y – на 3. Результат умножения вывести на экран.

```
#include "stdafx.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    int i, x, y;
    FILE *fp;
    fopen_s(&fp, "dat.txt", "w+"); // создать файл в режиме записи и чтения
    for (i = 0; i < 10; i++)
    {
        printf("X=%d\n");
        scanf_s("%d", &x);
        printf("\n Y=%d\n");
        scanf_s("%d", &y);
        printf("%d %d \n", x, y);
        fprintf(fp, "%5d, %5d\n", x, y); //запись данных в файл
    }
    fclose(fp); //закрыть файл
    printf("Number lists\n");
    fopen_s(&fp, "dat.txt", "r+"); //открыть файл в режиме чтения и записи
    for (i = 0; i < 10; i++) //читать файл (в цикле)
    {
        fscanf_s(fp, "%5d, %5d", &x, &y); //поместить данные
        // из файла в переменные x и y
        fscanf_s(fp, "\n");
        x*=2; y*=3;
        printf("%5d%5d\n", x, y);
    }
}
```

```
_getch();
fclose(fp);
return 0;
}
```

9.4. Двоичные файлы

Текстовые файлы не являются единственным возможным способом хранения информации на диске. Можно запоминать информацию и в двоичном виде.

Способ хранения двоичных данных в файлах имеет два важных преимущества:

- увеличивается скорость извлечения из файла данных, не носящих чисто текстового характера, так как не требуется действий по преобразованию их из текстового формата в двоичный;
- имеет место экономия памяти и дискового пространства, поскольку двоичная кодировка более компактна и нет необходимости в использовании управляющих символов.

Принципиально обработка двоичных файлов не очень сильно отличается от обработки текстовых файлов. В любом случае, прежде чем работать с файлом, следует связать его физическое имя с логическим именем (с файловым потоком) и открыть файл, указав режим доступа. После работы с файлом его необходимо закрыть. Тем не менее, отличия обработки двоичных файлов и текстовых все же существуют. Рассмотрим их.

Открыть файл для двоичной обработки можно посредством вызова функции `fopen()`, но ко всем режимам доступа добавляют строчную латинскую букву `b`. Режимы доступа одинаковы для текстовых и двоичных файлов.

```
FILE *fb = fopen( "Bin_fil.dat", "wb");
```

откроет файл для записи в двоичном режиме. А чтобы открыть его для чтения и записи следует написать

```
FILE *fb = fopen( "Bin_fil.dat", "r+b");
```

Если после вызова функции `fopen()` указатель на файловый поток `fb` равен нулю, его можно использовать в последующих обращениях к функциям работы с двоичными файлами, таким, как `fread()` и `fwrite()`. Закрывают двоичные файлы, как и текстовые, функцией `fclose()`.

Не следует использовать функции обработки текстовых файлов применительно к двоичным файлам и наоборот.

Существует два способа доступа к элементам двоичных файлов: последовательный и произвольный.

9.5. Последовательный доступ к элементам двоичных файлов

Последовательный доступ к элементам файла особенно эффективен, если нужно перебрать все данные, хранящиеся в нем. Кроме того, если файл открыт для записи, но еще не содержит данных (пуст), то заполнение его возможно лишь в последовательном режиме.

Пример 9.5. Записать 100 целых, случайно выбранных чисел в двоичный файл.

```
#include "stdafx.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "Russian");
    FILE *f_out;
    int number;
    fopen_s (&f_out, "Int.dat", "wb");
    if (!f_out) { puts("Нельзя создать файл !\n");
        exit(1); }
    for (int i = 0; i < 100; i++) // цикл формирования данных и записи их в файл
        {number = rand()%100 - 50;
        fwrite(&number, sizeof(int), 1, f_out);
    }
    fclose(f_out);
    fopen_s (&f_out, "Int.dat", "rb");
    if (!f_out) { puts("Нельзя открыть файл !\n");
        exit(1); }
    for (int i = 0; i < 100; i++) // цикл чтения данных
        // из файла и вывода их на экран
        { fread(&number, sizeof(int), 1, f_out);
            cout<<number<<' ';
        }
    fclose(f_out);
    return 0;
}
```

Все использованные в программе функции встречались ранее, кроме функции `fwrite()`, производящей запись данных в поток. Прототип ее находится в заголовочном файле `stdio.h`. Синтаксическое описание функции имеет вид [17]:

```
size_t fwrite (const void *ptr, size_t site, size_t n, FILE *stream);
```

Параметры функции:

`const void *ptr` – указатель на исходные данные, записываемые в файл;

`size_t size` – размер в байтах одного элемента данных;
`size_t n` – число записываемых в файл элементов данных;
`FILE *stream` – указатель на файловый поток, открытый в двоичном режиме.

Поскольку функция `fwrite()` имеет в качестве одного из своих аргументов количество записываемых элементов, можно применять ее и для записи в файл сразу целого массива, который к моменту записи в файл должен быть сформирован и заполнен данными.

Пример 9.6. Записать в двоичный файл массив целых, случайно выбранных чисел.

```
#include "stdafx.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    FILE *f_out;
    int number[100]; // создали массив данных
    fopen_s (&f_out, "Int.dat", "wb");
    if (!f_out) { puts("Нельзя создать файл !\n");
        exit(1); }
    for (int i = 0; i < 100; i++) // заполнение массива
        {number[i] = rand()%100 - 50;
        cout<<number[i]<<' ';
        fwrite(number, sizeof(int), 100, f_out); // запись массива в файл
    }
    fclose(f_out);
    cout<<endl; cout<<endl;
    fopen_s (&f_out, "Int.dat", "rb");
    if (!f_out) { puts("Нельзя открыть файл !\n");
        exit(1); }
    fread(&number, sizeof(int), 100, f_out);
    for (int i = 0; i < 100; i++) cout<<number[i]<<' ';
    fclose(f_out);
    return 0;
}
```

Это оптимальный способ доступа с точки зрения скорости записи информации на диск.

Пример 9.7. Определить максимальное из целых чисел, записанных в двоичном файле, и его порядковый номер.

```
#include "stdafx.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <io.h> // для работы с дескриптором
```

```

using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{FILE *f_inp;
int cnt_max=0; // номер максимального элемента
int handle; // дескриптор файла
int value, max;
int number[20];
fopen_s (&f_inp, "d:\Int.dat", "wb");
if (!f_inp) { puts("Нельзя создать файл !\n");
    exit(1);
}
for (int i = 0; i < 20; i++)
{number[i] = rand()%100 - 50;
 cout<<number[i]<<' ';
}
fwrite(number, sizeof(int), 20, f_inp);
fclose(f_inp);
cout<<endl;
cout<<endl;
fopen_s (&f_inp, "d:\Int.dat", "rb"); // чтение в двоичном режиме
if (!f_inp) { puts("File not opened!\n" );
    exit(1);
}
handle = _fileno(f_inp); // преобразовать открытый
// файловый поток в дескриптор файла
fread(&max, sizeof(int), 1, f_inp);
cnt_max = 0;
for (unsigned int i=1; i<_filelength(handle)/sizeof(int);i++)
{ fread(&value, sizeof(int), 1, f_inp);
    if (value > max) {max = value; cnt_max = i; }
}
printf("Max = %5d, number=%4d", max, cnt_max);
fclose(f_inp);
_getch();
return 0;
}

```

Функция `filelength(f_inp)` в качестве аргумента принимает дескриптор файла, открытого функцией `fopen()`, и возвращает размер этого файла в байтах. *Дескриптор (handle)* – это число, которое уникальным образом идентифицирует определенный объект, в данном случае – файл. Для получения дескриптора, используемого для идентификации файла, служит функция преобразования файлового потока в дескриптор `fileno()`. Аргументом функции `fileno(FILE *stream)` является открытый файловый поток. Отрицательное значение дескриптора служит признаком ошибки.

Заметим, что, как и в массивах, первый элемент, находящийся в файле, имеет номер 0, второй – 1 и т. д.

Функция чтения значений с диска `fread()` имеет тот же синтаксис, что и функция записи `fwrite()`, только первым аргументом здесь является ад-

рес приемника, в который эта функция должна скопировать байты с диска. При использовании этой функции следует убедиться в том, что размер приемника достаточен для копирования количества байтов. Прототип функции `fread()` находится в заголовочном файле `stdio.h`.

Функция `fread()` может загрузить больше одного значения с диска в один прием. Тогда в качестве приемника должен выступать массив элементов соответствующего типа и достаточного размера. Этот массив может быть статическим, если заранее известна длина файла, или динамическим, если размер файла определяется в процессе выполнения программы.

```

float arr_f[100];
...
fread (&arr_f, sizeof(float), 100, f_inp);

```

Это самый быстрый способ загрузки из файла большого количества чисел.

9.6. Организация произвольного доступа к элементам двоичных файлов

Организация произвольного доступа к компонентам файла позволяет считывать значения из любой позиции в файле, а также записывать новую информацию в любое место в файле. Но к файлам с произвольным доступом предъявляется одно жесткое требование: их компоненты должны иметь одинаковую длину. Двоичные файлы позволяют обеспечить удовлетворение этого требования. О том, чтобы данные, которые будут находиться в файле произвольного доступа, имели одинаковый размер, следует позаботиться в момент создания файла.

Еще раз напомним о том, что первичная запись в файл возможна только в режиме последовательного доступа.

Для организации произвольного доступа к элементам файла используют функцию `fseek()`, прототип которой описан в заголовочном файле `stdio.h`. Синтаксическое описание функции [17]:

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

Функция `fseek()` перемещает внутренний указатель файлового потока, изменяя таким образом место в файле, с которого начинается следующая операция чтения или записи. В случае успешного завершения функция возвращает 0, в случае ошибки – ненулевое значение.

Параметры функции:

`FILE *stream` – указатель на открытый файловый поток, аналогичный возвращаемому функцией `fopen()`;

long offset – число байтов, на которое нужно переместить файловый указатель в направлении, указанном параметром whence. Для перемещения файлового указателя в обратном направлении (в сторону начала файла) следует устанавливать offset равным отрицательному значению;

int whence – указывает положение точки отсчета файлового указателя, от которой будет происходить его перемещение. Значения аргумента whence представлены в табл. 9.4.

Таблица 9.4. Значения аргумента whence

Значение	Описание
SEEK_SET	Перемещение файлового указателя происходит относительно начала файла
SEEK_END	Перемещение файлового указателя происходит относительно конца файла
SEEK_CUR	Перемещение файлового указателя происходит относительно текущей позиции файлового указателя

При использовании функции fseek() следует соблюдать осторожность, так как из-за ограничений DOS попадание за пределы файла чаще всего не приводит к генерации ошибки, поэтому программисту самому следует принимать меры для предотвращения обращения к диску за пределами известных границ файла.

При организации произвольного доступа используется функция ftell(), осуществляющая навигацию внутри файла. Прототип функции описан в stdio.h. Данная функция возвращает внутренний указатель файлового потока, равный смещению в байтах от начала двоичного файла до байта, с которого начинается следующая операция ввода-вывода. Это значение можно передать функции fseek() или использовать каким-либо другим образом.

Синтаксическое описание функции:

long ftell(FILE *stream).

Единственным параметром функции является указатель на открытый файловый поток.

Следующие операторы демонстрируют возможности функции fseek(), а именно перемещают файловый указатель:

fseek(f, sizeof(t), SEEK_CUR); – с текущей позиции на следующую;
fseek(f, -sizeof(t), SEEK_CUR); – на предыдущую позицию;
fseek(f, 0, SEEK_END); – на конец файла.

Пример 9.8. Прочитать из файла, содержащего целые числа, его 15-й элемент.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
```

```
#include <stdlib.h>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "Russian");
    FILE *inp_f;
    int value;
    fopen_s (&inp_f, "d:\\Int.dat", "rb");
    if (!inp_f) { puts ("Невозможно открыть файл\\n"); exit(1); }
    fseek(inp_f, 15 * sizeof(int), SEEK_SET);
    fread(&value, sizeof(int), 1, inp_f);
    printf("Прочитанное число = %d\\n", value);
    fclose(inp_f);
    getch();
    return 0;
}
```

В любом случае при работе с двоичными файлами не следует забывать добавлять букву b при указании режима доступа функции fopen(), но нужно быть особенно внимательным, если предполагается, что будет осуществляться произвольный доступ к компонентам данного файла.

Пример 9.9. Записать нуль на место максимального значения в файле Test.dat.

```
#include "stdafx.h"
#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <iomanip>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    FILE *f_inp; // номер максимального элемента
    int i, max; // дескриптор файла
    int handle;
    int value, max, zero = 0, number[20];
    fopen_s (&f_inp, "Int.dat", "wb");
    if (!f_inp) { puts("Нельзя создать файл !\\n"); exit(1); }
    for (int i = 0; i < 20; i++)
        number[i] = rand()%100 - 50;
    cout<<number[i]<<' ';
    fwrite(number, sizeof(int), 20, f_inp);
    free(f_inp);
    cout<<endl;
    cout<<endl;
    fopen_s (&f_inp, "Int.dat", "r+b");
    if (!f_inp) { puts("File not opened !\\n"); exit(1); }
```

```

handle = _fileno(f_inp);
fread(&max, sizeof(int), 1, f_inp);
i_max = 0;
for (int i = 1; i < _filelength(handle); i++) // 0 элемент считали раньше
{ fread(&value, sizeof(int), 1, f_inp);
  if (value > max) {max = value; i_max = i;}
}
cout << max << " " << i_max << endl;
fseek(f_inp, i_max * sizeof(int), SEEK_SET);
fwrite(&zero, sizeof(int), 1, f_inp);
fclose(f_inp);
fopen_s(&f_inp, "Int.dat", "r+b");
if (!f_inp) {puts("File not opened !\n"); exit(1);}
}
fread(number, sizeof(int), 20, f_inp);
for (int i = 0; i < 20; i++) cout << number[i] << ' ';
cout << endl;
fclose(f_inp);
return 0;
}

```

Контрольные вопросы

- Что включает в себя понятие файла? Как оно связано со стандартными потоками ввода-вывода?
- Перечислить стандартные потоки ввода-вывода, открываемые при запуске программы на языках С и С++.
- Как связаны между собой понятия логического и физического файлов?
- Как работает функция fopen()?
- Что такое режим доступа? Перечислить возможные режимы доступа при работе с текстовыми файлами.
- Почему при работе с файлами не рекомендуется использовать оператор цикла с последующим условием?
- Каков механизм действий, связанных с закрытием файла? Почему необходимо закрывать файлы по окончании работы с ними?
- Как открыть двоичный файл?
- Какие функции осуществляют чтение и запись при работе с двоичными файлами?
- В чем состоят особенности чтения массивов из двоичных файлов и их записи в двоичные файлы?
- Что такое последовательный и произвольный доступ к компонентам файла?
- В чем заключается опасность использования функции fseek()?
- Как «подойти» к предпоследнему элементу файла? Привести все возможные варианты.
- Для чего используется функция ftell()?

Практические задания 9

- Дана целочисленная матрица размера 5×5 . Получить новую матрицу путем умножения всех элементов на наименьший по модулю элемент. Сформировать одномерный массив из максимальных элементов каждой строки полученной матрицы. Результат работы программы поместить в текстовый файл.
- Из данного текстового файла получить другой, в котором все строки исходного файла будут пронумерованы.
- Дан текстовый файл произвольного формата. Преобразовать его так, чтобы каждая следующая строка содержала на 5 элементов больше, чем текущая. Первая строка состоит из трех элементов, например:

wdcvfdgjdhkjhk
nkfgkkfjlkjlflkjkblskjgkfjgldfkj
fljfgjlfkgjl
fdlgk;kdgk;
toiuartjigr
968r0itrptosfgkkgf

wdc
vfdgdjh
kihknkfgkkfgl
kjlijflkjkblskjgkfj
gldfkjfljfgjlfkgjl;fdl
gk;kdgk;toiuartjigr968r0it
rptosfgkkgf

- Дана база данных магазина игрушек, реализованная в виде текстового файла:
 - название игрушки,
 - цена,
 - артикул,
 - количество,
 - возраст детей, для которых она предназначена.

Составить программу, позволяющую выбрать игрушки для ребенка указанного возраста, стоимость каждой из которых не превышает имеющихся в наличии денег. Результат работы программы вывести в текстовый файл.

Записать в динамический массив данные из файла типа double.

Заменить в файле типа int минимальное значение суммой предшествующих элементов, а максимальное – суммой последующих элементов. Особо учесть случаи, когда минимальное и/или максимальное значение стоят на первом или на последнем месте.

В файле типа int заменить четные элементы на 0, а нечетные на 1.

В файле типа double поменять местами максимальный и минимальный элементы.

В файле типа float заменить максимальный и минимальный элементы средним арифметическим значением элементов этого файла.

В конец целочисленного файла дописать нечетные значения, содержащиеся в этом файле.

11. Упорядочить значения двоичного файла, содержащего данные типа `char`, по алфавиту.
12. Из файла, содержащего как положительные, так и отрицательные вещественные числа, сформировать два других, в один из которых поместить значения, большие чем число, введенное с клавиатуры, в другой – меньшие. Определить количество элементов в обоих файлах.

ГЛАВА 10. ОБОБЩЕННАЯ АРХИТЕКТУРА ПРОЦЕССОРА IBM PC

10.1. Понятие об адресном пространстве

Практически все микропроцессоры содержат следующие элементы: арифметико-логическое устройство (АЛУ), регистры, дешифратор команд, блок управления и синхронизации, внутренние шины цепей управления, связанные сшинами адресов и данных, входы и выходы управления.

АЛУ выполняет арифметические и логические операции, а также операции сдвига.

Регистры – это внутренние запоминающие устройства процессора для временного хранения обрабатываемой или управляющей информации. Регистры IBM PC группируются следующим образом:

- 1) регистры общего назначения (РОН) – восемь 16-разрядных регистров, которые могут произвольно использоваться программистами: AX, DX, CX, BX, BP, SI, DI, SP. В качестве РОН могут применяться 32-битовые регистры: EAX, EDX, ECX, EBX, EBP, ESI, EDI, ESP;
- 2) регистры сегментов: CS (регистр сегмента кодов), SS (регистр сегмента стека), DS (регистр сегмента данных), ES (регистр дополнительного сегмента);
- 3) регистр системных флагов: FLAGS;
- 4) управляющий регистр (счетчик команд): IP.

Дешифратор команд интерпретирует (декодирует) содержимое регистра команд, выполняет необходимые преобразования и активизирует работу блока управления.

Важнейшей характеристикой любого микропроцессора является разрывность его внутренних регистров и внешних шин адресов и данных. Совместимые микропроцессоры имеют 16-разрядную внутреннюю архитектуру и 16-разрядную шину данных. Регистры внутри процессора имеют ширину 16 бит. Таким образом, максимальное целое число (данное или адрес), которым может работать микропроцессор, составляет $2^{16} - 1 = 65\,535$ (65 Кбайт – 1). Однако адресная шина микропроцессора 8086 содержит 20 бит, что соответствует адресному пространству $2^{20} = 1$ Мбайт. Для того чтобы с помощью 16-разрядных адресов можно было обращаться в любую точку 20-разрядного адресного пространства, в микропроцессоре IBM PC используется сегментная адресация памяти, реализуемая с помощью четырех сегментных регистров: CS, SS, DS и ES.

Аппаратный интерфейс процессора и памяти во всех компьютерах одинаков: процессор выдает физический адрес нужной ячейки па-

мяти на шину адреса и формирует сигнал чтения или записи. В случае операции записи он помещает данные на шину данных. В случае операции чтения память возвращает считанные данные по шине данных. Процессор, имеющий n линий в шине адреса, может обращаться к 2^n ячейкам памяти. Схемы интерфейса действуют линейно, т. е. каждая из 2^n возможных комбинаций на линиях шины адреса выбирает однозначно определенную ячейку памяти. Такая модель памяти называется *плоской (flat)* или *сплошной памятью* и позволяет программам адресоваться к ячейкам памяти от адреса 0 и до адреса $2^n - 1$.

В PC-совместимых процессорах применяется сегментированная модель памяти, которая отличается от линейной аппаратной модели (рис. 10.1).

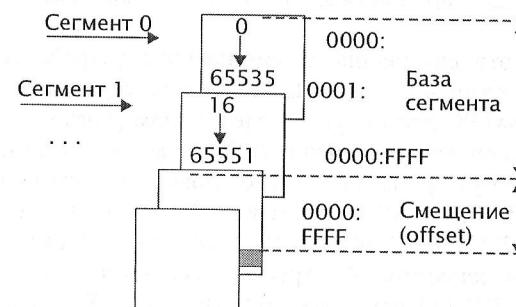


Рис. 10.1. Сегментная организация памяти IBM PC

С точки зрения программы адресное пространство разделено на блоки смежных адресов, называемые сегментами, а программа может обращаться только к данным, находящимся в этих сегментах.

Адресное пространство (address space) – это множество ячеек памяти, к которым может обращаться задача. Адресное пространство может представлять собой сплошной участок или состоять из нескольких сегментов. Система управления памятью отображает адресное пространство задачи на физическую память ЭВМ.

10.2. Система адресации в MS-DOS

Внутри сегментов применяется линейная адресация относительно начала сегмента, а физический адрес, например с программным адресом 0, по существу, программисту неизвестен.

Под *сегментом* понимается блок сплошных ячеек памяти (в адресном пространстве 1 Мбайт) с максимальным размером 64 Кбайт и начальным или базовым адресом, находящимся на 16-байтовой границе, называемой *параграфом*.

Перекрывающиеся сегменты начинаются каждые 16 байт и могут достигать 65 535 байт длины. Для обращения к памяти необходимо опре-

делить базу сегмента (16-разрядную) и смещение или относительный адрес (16-битовое расстояние от базы). Поэтому в PC-совместимых компьютерах адрес формируется в виде пары *сегмент:смещение*. Такое представление называется *логическим или виртуальным адресом*. Преобразование логического адреса в физический всегда однозначно, т. е. каждому логическому адресу соответствует единственный физический адрес. Однако каждому физическому адресу соответствует 4 Кбайт логических адресов.

Для получения физического (сплошного) адреса ячейки, отсчитанного от начала памяти 0000:0000, содержимое сегментного регистра нужно сдвигнуть на 4 бита влево (что эквивалентно умножению на 16), а затем сложить со смещением:

$$\text{физический адрес} = \text{сегмент} * 16 + \text{смещение}.$$

Пусть DS содержит значение 1234h, SI содержит значение 5678h. Литера h служит признаком того, что число представлено в шестнадцатеричной системе счисления. Тогда физический адрес будет равен:
 $(DS)*16+(SI)=1234h*16+5678h=179B8h$.

Иногда необходимо знать нормализованный адрес, т. е. адрес, приведенный к такому виду, что смещение находится в диапазоне от 0 до 16 (000Fh). Если вычислен сплошной адрес ячейки памяти, то его можно легко пересчитать в нормализованный формат:

- сегмент = сплошной адрес/16 (/ – целочисленное деление);
- смещение = сплошной адрес%16 (% – остаток от целочисленного деления).

Память, выделяемая для программы на C++, необходима:

- для размещения программного кода;
- размещения данных;
- динамического использования;
- резервирования компилятором на время выполнения программы.

10.3. Понятие о моделях памяти

Концептуальная модель распределения памяти в C++ представлена на рис. 10.2.

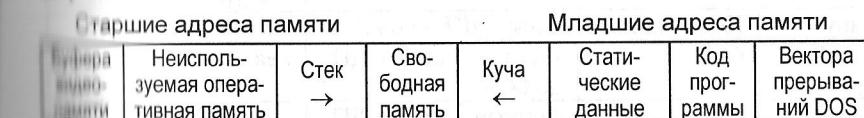


Рис. 10.2. Модель распределения памяти

Размер области памяти, занимаемой векторами прерываний, определяется архитектурой ЭВМ. Для IBM PC – это 256 ячеек памяти (с 0 до 255).

Область памяти, отводимая под код программы, в процессе работы программы остается неизменной.

Не изменяется и память, занимаемая статическими данными.

Куча – это специальным образом организованная область памяти, используемая для работы с динамическими структурами. Объем памяти для кучи зависит от того, сколько памяти запрашивается программой с помощью функции `alloc` (и подобных ей).

Стек (stack) – область памяти, организованная так, что позволяет добавлять и удалять элементы данных, но при этом доступен только последний добавленный элемент.

Размер использованной памяти стека изменяется в результате активизации локальных переменных в функциях, а также за счет того, что при вызовах функций в стек заносятся параметры функций.

В ОС MS-DOS для микропроцессоров семейства 8086 (PC-совместимых) размер указателя (число байтов, требуемых для размещения адреса памяти под указатель) зависит от модели памяти, задаваемой при компиляции программы.

В C++ программы можно компилировать в расчете на 6 моделей памяти:

`tiny` – крошечную;

`small` – маленькую (модель по умолчанию);

`medium` – среднюю;

`compact` – компактную;

`large` – большую;

`huge` – огромную, использующую нормализованные адреса;

`flat` – использующую 32-разрядные сегменты.

Близкие (*near*) указатели занимают 2 байта, дальние (*far*) – 4 байта: два под сегмент и два под смещение.

Модели памяти, используемые в C(C++), представлены в табл. 10.1.

Таблица 10.1. Модели памяти DOS

Модель	Код	Данные	Стек	Тип указателей данных и динамической памяти (ДП) по умолчанию
Tiny	Код + данные + стек + ДП до 64 К			near
Small	64 К	Данные + стек + ДП до 64 К		near
Medium	64 К	Данные + стек + ДП до 64 К		near
Compact	64 К	64 К		far
Large	Больше 64 К	Больше 64 К		far
Huge	Больше 64 К	Больше 64 К		far
Flat	Код + данные + стек + ДП до 1 МБ			near

Контрольные вопросы

- Перечислить основные составляющие процессора. Какие функции выполняет каждый из них?
- Почему разрядность внутренних регистров и внешних шин адресов и данных процессора является его важнейшей характеристикой?
- Что представляют собой плоская и сегментированная модели организации памяти? Какая из них однозначно соответствует физической реализации памяти?
- Дать определение адресного пространства.
- Что определяют понятия: сегмент, смещение, параграф?
- Как вычислить физический адрес ячейки памяти, если известен ее логический адрес?
- Что понимают под нормализованным адресом ячейки памяти и как можно его определить?
- В чем заключается разница в организации и функциональном назначении стека и кучи?
- Перечислить основные модели памяти и определить понятия ближнего и дальнего указателей.