Глава 6. Алгоритмизация и программирование

§ 38. Целочисленные алгоритмы

Во многих задачах все исходные данные и необходимые результаты – целые числа. При этом всегда желательно, чтобы все промежуточные вычисления тоже проводились с только целыми числами. На это есть, по крайней мере, две причины:

- процессор, как правило, выполняет операции с целыми числами значительно быстрее, чем с вещественными;
- целые числа всегда точно представляются в памяти компьютера, и вычисления с ними также выполняются без ошибок (если, конечно, не происходит переполнение разрядной сетки).

Решето Эратосфена

Во многих прикладных задачах, например, при шифровании с помощью алгоритма RSA, широко используются простые числа. Основные задачи при работе с простыми числами – это проверка числа на простоту и нахождение всех простых чисел в заданном диапазоне.

Пусть задано некоторое натуральное число N и требуется найти все простые числа в диапазоне от 2 до N. Самое простое (но неэффективное) решение этой задачи состоит в том, что в цикле перебираются все числа от 2 до N, и каждое из них отдельно проверяется на простоту. Например, можно проверить, есть ли у числа k делители в диапазоне от 2 до $\sqrt{k}$. Если ни одного такого делителя нет, то число k простое.

Описанный метод при больших N работает очень медленно, он имеет асимптотическую сложность $O(N\sqrt{N})$. Греческий математик Эратосфен Киренский (276-194 гг. до н.э.) предложил другой алгоритм, который работает намного быстрее, его сложность $O(N \log\log N)$:

1) выписать все числа от 2 до N;
2) начать с $k = 2$;
3) вычеркнуть все числа, кратные k ($2k, 3k, 4k$ и т.д.);
4) найти следующее не вычеркнутое число и присвоить его переменной k;
5) повторять шаги 3 и 4, пока $k < N$.

Покажем работу алгоритма при $N = 16$:

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Первое не вычеркнутое число – 2, поэтому вычеркиваем все чётные числа:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ 9 ~~10~~ 11 ~~12~~ 13 ~~14~~ 15 ~~16~~

Далее вычеркиваем все числа, кратные 3:

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~

А все числа, кратные 5 и 7 уже вычеркнуты. Таким образом, получены простые числа 2, 3, 5, 7, 11 и 13.

Классический алгоритм можно улучшить, уменьшив количество операций. Заметьте, что при вычеркивании чисел, кратных трем, нам не пришлось вычеркивать число 6, так как оно уже было вычеркнуто. Кроме того, все числа, кратные 5 и 7, к последнему шагу тоже оказались вычеркнуты.

Предположим, что мы хотим вычеркнуть все числа, кратные некоторому k, например, $k = 5$. При этом числа $2k, 3k$ и $4k$ уже были вычеркнуты на предыдущих шагах, поэтому нужно начать не с $2k$, а с k^2. Тогда получается, что при $k^2 > N$ вычеркивать уже будет нечего, что мы и увидели в примере. Поэтому можно использовать улучшенный алгоритм:

1) выписать все числа от 2 до N;
2) начать с $k = 2$;
3) вычеркнуть все числа, кратные k, начиная с k^2;
4) найти следующее не вычеркнутое число и присвоить его переменной k;
5) повторять шаги 3 и 4, пока $k^2 \le N$.

Чтобы составить программу, нужно определить, что значит «выписать все числа» и «вычеркнуть число». Один из возможных вариантов хранения данных – массив логических величин с индексами от 2 до N. Как и в учебнике 10 класса, в левой части страницы будем писать программы на языках C и C++, рассматривая оба варианта программы, если они отличаются.

Объявление переменных в программе будет выглядеть так (для $N = 100$):

```
const int N=100;
bool A[N+1];
int i, k;
```

Если число **i** не вычеркнуто, будем хранить в элементе массива **A[i]** истинное значение (**true**), если вычеркнуто – ложное (**false**). Поскольку нумерация элементов массива в C и C++ начинается с нуля, а нам удобнее использовать «естественную» нумерацию (с единицы), мы выделили на 1 элемент больше, и нулевой элемент массива задействовать не будем.

В самом начале нужно заполнить массив истинными значениями (все числа не вычеркнуты):

```
for ( i=2; i<=N; i++ )
  A[i]=true;
```

В основном цикле выполняется описанный выше алгоритм:

```
k=2;
while ( k*k<=N )
  {
  if ( A[k] )
    {
    i=k*k;
    while ( i<=N )
      {
      A[i]=false;
      i+=k;
      }
    }
  k++;
  }
```

Обратите внимание, что для того, чтобы вообще не применять вещественную арифметику, мы заменили условие $k \le \sqrt{N}$ на равносильное условие $k^2 \le N$, в котором используются только целые числа.

После завершения этого цикла не вычеркнутыми остались только простые числа, для них соответствующий элемент массива содержит истинное значение. Эти числа нужно вывести на экран:

```
for ( i=2; i<=N; i++ )
  if ( A[i] )
    printf ( "%d ", i );
```

```
for ( i=2; i<=N; i++ )
  if ( A[i] )
    cout << i << " ";
```

«Длинные» числа

Современные алгоритмы шифрования используют достаточно длинные ключи, которые представляют собой числа длиной 256 бит и больше. С ними нужно выполнять разные операции: складывать, умножать, находить остаток от деления. Вопрос состоит в том, как хранить такие числа в памяти, где для целых чисел отводятся ячейки значительно меньших размеров (обычно до 64 бит). Ответ достаточно очевиден: нужно «разбить» длинное число на части так, чтобы использовать несколько ячеек памяти.

Длинное число – это число, которое не помещается в переменную одного из стандартных типов данных языка программирования. Алгоритмы работы с длинными числами называют «длинной арифметикой».

Для хранения длинного числа удобно использовать массив целых чисел. Например, число 12345678 можно записать в массив с индексами от 0 до 9 таким образом:

	0	1	2	3	4	5	6	7	8	9
A	1	2	3	4	5	6	7	8	0	0

Такой способ имеет ряд недостатков:

1) нужно где-то хранить длину числа, иначе числа 12345678, 123456780 и 1234567800 будет невозможно различить;
2) неудобно выполнять арифметические операции, которые начинаются с младшего разряда;
3) память расходуется неэкономно, потому что в одном элементе массива хранится только один разряд – число от 0 до 9.

Чтобы избавиться от первых двух проблем, достаточно «развернуть» массив наоборот, так чтобы младший разряд находился в **A[0]**. В этом случае на рисунках удобно применять обратный порядок элементов:

	9	8	7	6	5	4	3	2	1	0
A	0	0	1	2	3	4	5	6	7	8

Теперь нужно найти более экономичный способ хранения длинного числа. Например, разместим в одной ячейке массива три разряда числа, начиная справа:

	9	8	7	6	5	4	3	2	1	0
A	0	0	0	0	0	0	0	12	345	678

Здесь использовано равенство

$$12345678 = 12 \cdot 1000^2 + 345 \cdot 1000^1 + 678 \cdot 1000^0.$$

Фактически мы представили исходное число в системе счисления с основанием 1000!

Сколько разрядов можно хранить в одной ячейке массива? Это зависит от ее размера. Например, если ячейка занимает 4 байта и число хранится со знаком (тип **long int** в языках C и C++), допустимый диапазон ее значений

$$\text{от } -2^{32} = -4\,294\,967\,296 \quad \text{до} \quad 2^{32} - 1 = 4\,294\,967\,295.$$

В такой ячейке можно хранить до 9 разрядов десятичного числа, то есть использовать систему счисления с основанием 1 000 000 000. Однако нужно учитывать, что с такими числами будут выполняться арифметические операции, результат которых должен «помещаться» в ячейку памяти. Например, если надо умножать разряды этого числа число на $k < 100$, и в языке программирования нет 64-битных целочисленных типов данных, можно хранить в ячейке не более 7 разрядов.

Задача 1. Требуется *вычислить точно значение факториала* $100! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot 99 \cdot 100$ и вывести его на экран в десятичной системе счисления (*это число состоит более чем из сотни цифр и явно не помещается в одну ячейку памяти*).

Для хранения длинного числа будем использовать целочисленный массив A. Определим необходимую длину массива. Заметим, что

$$1 \cdot 2 \cdot 3 \cdot \ldots \cdot 99 \cdot 100 < 100^{100}$$

Число 100^{100} содержит 201 цифру, поэтому число 100! содержит не более 200 цифр. Если в каждом элементе массива записано 6 цифр, для хранения всего числа требуется не более 34 ячеек:

```
const int N=33;
long int A[N+1];
```

Чтобы найти 100!, нужно сначала присвоить «длинному» числу значение 1, а затем последовательно умножать его на все числа от 2 до 100. Запишем эту идею на псевдокоде, обозначив через **[A]** длинное число, находящееся в массива **A**:

```
[A]=1;
for ( k=2; k<=100; k++ )
  [A]=[A]*k;
```

Записать в длинное число единицу – это значит присвоить элементу **A[0]** значение 1, а в остальные ячейки записать нули:

```
A[0]=1;
for ( i=1; i<=N; i++ )
  A[i]=0;
```

Таким образом, остается научиться умножать длинное число на «короткое» ($k \le 100$). «Короткими» обычно называют числа, которые помещаются в переменную одного из стандартных типов.

Попробуем сначала выполнить такое умножение на примере. Предположим, что в каждой ячейке массива хранится 6 цифр длинного числа, то есть используется система счисления с основанием d = 1 000 000. Тогда число **[A]=12345678901734567** хранится в трех ячейках

	2	1	0
A	**12345**	**678901**	**734567**

Пусть $k = 3$. Начинаем умножать с младшего разряда: **734567·3 = 2203701**. В нулевом разряде может находиться только 6 цифр, значит, старшая двойка перейдет в перенос в следующий разряд. В программе для выделения переноса r можно использовать деление на основание системы счисления d с отбрасыванием остатка. Сам остаток – это то, что остается в текущем разряде. Поэтому получаем

```
s = A[0]*k;
A[0] = s % d;
r = s / d;
```

Для следующего разряда будет всё то же самое, только в первой операции к произведению нужно добавить перенос из предыдущего разряда, который был записан в переменную **r**. Приняв в самом начале **r = 0**, запишем умножение длинного числа на короткое в виде цикла по всем элементам массива, от **A[0]** до **A[N]**:

```
r = 0;
for ( i = 0; i <= N; i++ )
  {
  s = A[i] * k + r;
  A[i] = s % d;
  r = s / d;
  }
```

В свою очередь, эти действия нужно выполнить в другом (внешнем) цикле для всех **k** от 2 до 100:

```
for ( k = 2; k <= 100; k++ )
  {
  ...
  }
```

После этого в массиве **A** будет находиться искомое значение 100!, остается вывести его на экран. Нужно учесть, что в каждой ячейке хранятся 6 цифр, поэтому в массиве

	2	1	0
A	**1**	**2**	**3**

хранится значение 1000002000003, а не 123. Кроме того, старшие нулевые разряды выводить на экран не надо. Поэтому при выводе требуется

1) найти первый (старший) ненулевой разряд числа;
2) вывести это значение без лидирующих нулей;
3) вывести все следующие разряды, добавляя лидирующие нули до 6 цифр.

Поскольку мы знаем, что число не равно нулю, старший ненулевой разряд можно найти в таком цикле[1]:

```
i = N;
while ( !A[i] )
  i --;
```

Старший разряд выводим обычным образом (без лидирующих нулей):

```
printf ( "%ld", A[i] );
```

```
cout << A[i];
```

Формат вывода **"%ld"** в программе на языке С предназначен для вывода длинных целых чисел (тип **long int**).

Для остальных разрядов будем использовать специальную процедуру **Write6**:

```
for ( k = i-1; k >= 0; k-- )
  Write6 ( A[k] );
```

Эта процедура последовательно выводит цифры десятичного числа, начиная с сотен тысяч и кончая единицами:

[1] Подумайте, что изменится, если выводимое число может быть нулевым.

```
void Write6 ( long int x )
{
  long int M=100000;
  while ( M>0 )
    {
    printf ( "%d", x/M );
    x%=M;
    M/=10;
    }
}
```

```
void Write6 ( long int x )
{
  long int M=100000;
  while ( M>0 )
    {
    cout << x/M;
    x%=M;
    M/=10;
    }
}
```

Для того, чтобы разобраться, как она работает, выполните «ручную прокрутку» при различных значениях **x** (например, возьмите **x** = **1**, **x** = **123** и **x** = **123456**).

Контрольные вопросы

1. Какие преимущества и недостатки имеет алгоритм «решето Эратосфена» в сравнении с проверкой каждого числа на простоту?
2. Что такое «длинные числа»?
3. В каких случаях необходимо применять «длинную арифметику»?
4. Какое максимальное число можно записать в ячейку размером 64 бита? Рассмотрите варианты хранения чисел со знаком и без знака.
5. Можно ли использовать для хранения длинного числа символьную строку? Какие проблемы при этом могут возникнуть?
6. Почему неудобно хранить длинное число, записывая первую значащую цифру в начало массива?
7. Почему неэкономично хранить по одной цифре в каждом элементе массива?
8. Сколько разрядов числа можно хранить в одной 16-битной ячейке?
9. Объясните, какие проблемы возникают при выводе длинного числа. Как их можно решать?
10. Объясните работу процедуры **Write6**.

Задачи

1. Докажите, что если у числа k нет ни одного делителя в в диапазоне от 2 до $\sqrt{k}$, то оно простое.
2. Напишите две программы, которые находят все простые числа в диапазоне от 2 до N двумя разными способами:
   а) проверкой каждого числа из этого диапазона на простоту;
   б) используя решето Эратосфена.
   Сравните число шагов цикла (или время работы) этих программ для разных значений **N**. Постройте для каждого варианта зависимость количества шагов от **N**, сделайте выводы о сложности алгоритмов.
3. Докажите, что в приведенной программе вычисления 100! не будет переполнения при использовании 32-битных целых переменных.
4. Можно ли в той же программе в одной ячейке массива хранить 9 цифр длинного числа?
5. Без использования программы определите, сколько нулей стоит в конце числа 100!
6. Соберите всю программу и вычислите 100!. Сколько цифр входит в это число?
7. Оформите вывод длинного числа на экран в виде отдельной процедуры. Учтите, что число может быть нулевым.
8. *Придумайте другой способ вывода длинного числа, использующий символьные строки.
9. Напишите процедуру для ввода длинных чисел из файла.
10. Напишите процедуры для сложения и вычитания длинных чисел.
11. *Напишите процедуры для умножения и деления длинных чисел.
12. **Напишите процедуру для извлечения квадратного корня из длинного числа.

§ 39. Структуры (записи)

Зачем нужны структуры?

Представим себе базу данных библиотеки, в которой хранится информация о книгах. Для каждой из них нужно запомнить автора, название, год издания, количество страниц, число экземпляром и т.д. Как хранить эти данные?

Поскольку книг много, нужен массив. Но в массиве используются элементы одного типа, тогда как информация о книгах разнородна, она содержит целые числа и символьные строки разной длины. Конечно, можно разбить эти данные на несколько массивов (массив авторов, массив названий и т.д.), так чтобы **i**-ый элемент каждого массива относился к книге с номером **i**. Но такой подход оказывается слишком неудобен и ненадежен. Например, при сортировке нужно переставлять элементы всех массивов (отдельно!) и можно легко ошибиться и нарушить связь данных.

Возникает естественная идея – объединить все данные, относящиеся к книге, в единый блок памяти, который в программировании называется структурой.

Структура – это тип данных, который может включать в себя несколько *полей* – элементов разных типов (в том числе и другие структуры).

Объявление структур

Как и любые переменные, структуры необходимо объявлять. До этого мы работали с простыми типами данных (целыми, вещественными, логическими и символьными), а также с массивами этих типов. Вы знаете, что при объявлении переменных и массивов указывается их тип, поэтому для того, чтобы работать со структурами, нужно ввести новый тип данных.

Построим структуру, с помощью которой можно описать книгу в базе данных библиотеки. Будем хранить в структуре только[2]

- фамилию автора (строка, менее 40 символов);
- название книги (строка, менее 80 символов);
- имеющееся в библиотеке количество экземпляров (целое число).

Объявление такого *составного* типа в языке С имеет вид:

```
typedef struct {
    char author[40];  // автор, строка
    char title[80];   // название, строка
    int count;        // количество, целое
    } TBook;
```

Объявления типов данных начинаются с ключевого слова **type** (англ. тип) и располагаются выше блока объявления переменных. Имя нового типа – **TBook** – это удобное сокращение от английских слов ***Type Book*** (*тип книга*), хотя можно было использовать и любое другое имя, составленное по правилам языка программирования. Слово **struct** означает, что этот тип данных – структура; далее в фигурных скобках перечисляются поля и указывается их тип.

Обратите внимание, что для строк **author** и **title** указан максимальный размер. Это сделано для того, чтобы точно определить, сколько места нужно выделить на них в памяти.

В языке С++ для хранения символьных строк можно использовать тип **string**:

```
typedef struct {
    string author;  // автор, строка
    string title;   // название, строка
    int count;      // количество, целое
    } TBook;
```

В результате такого объявления никаких структур в памяти не создается: мы просто описали новый тип данных, чтобы транслятор знал, что делать, если мы захотим его использовать.

Теперь можно использовать тип **TBook** так же, как и простые типы, для объявления переменных и массивов:

[2] Конечно, в реальной ситуации данных больше, но принцип не меняется.

```
const int N=100;
TBook B;
TBook Books[N];
```

Здесь введена переменная **B** типа **TBook** и массив **Books**, состоящий из 100 элементов того же типа.

Иногда бывает нужно определить размер одной структуры. Для этого используется стандартная функция **sizeof**, которой можно передать имя типа, а также переменную или массив:

```
printf ( "%d\n", sizeof(TBook) );
printf ( "%d\n", sizeof(B) );
printf ( "%d\n", sizeof(Books) );
```

Для структуры, которая хранит символьные строки как массивы символов, первые две команды выведут на экран размер одной структуры (124 байта, если целое число типа **int** занимает в памяти 4 байта), а последняя – размер выделенного массива из 100 структур.

Для структуры на языке C++, в которой для хранения полей **author** и **title** используется тип **string**, ситуация более сложная. Команды вывода

```
cout << sizeof(TBook) << endl;
cout << sizeof(B) << endl;
cout << sizeof(Books) << endl;
```

покажут на экране 12, 12 и 1200. Почему так?

Дело в том, что объект класса **string** фактически представляет собой указатель – ссылку на область памяти, где хранятся данные этого объекта (длина строки и все её символы). Поэтому функция **sizeof** определяет размер именно этого указателя, который равен 4 байтам. Таким образом, использование типа **string**, с одной стороны, значительно облегчает работу с символьными строками в языке C++, а с другой может привести к неожиданным проблемам из-за того, что внутреннее устройство таких строк скрыто от программиста.

Обращение к полю структуры

Для того, чтобы работать не со всей структурой, а с отдельными полями, используют так называемую *точечную нотацию*, разделяя точкой имя структуры и имя поля. Например, **B.author** обозначает «поле **author** структуры **B**», а **Books[5].count** – «поле count элемента массива **Books[5]**». Например, для определения размера полей в байтах, можно снова использовать функцию **sizeof**:

```
printf ( "%d\n", sizeof(B.author) );
printf ( "%d\n", sizeof(B.title) );
printf ( "%d\n", sizeof(B.count) );
```

Выполнив эти команды для варианта на языке C, мы увидим на экране числа 40, 80 и 4.

Если в структуре на языке C++ используются строки типа **string**, в результате выполнения операторов:

```
cout << sizeof(B.author) << endl;
cout << sizeof(B.title) << endl;
cout << sizeof(B.count) << endl;
```

мы увидим 4, 4 и 4. Причины такого поведения мы уже обсуждали.

С полями структуры можно обращаться так же, как и с обычными переменными соответствующего типа. Можно вводить их с клавиатуры (или из файла):

```
gets ( B.author );
gets ( B.title );
scanf ( "%d", &B.count );
```

```
cin >> B.author;
cin >> B.title;
cin >> B.count;
```

присваивать новые значения[3]:

```
strcpy ( B.author,"Пушкин А.С." );
strcpy ( B.title, "Полтава" );
B.count=1;
```

```
B.author="Пушкин А.С.";
B.title="Полтава";
B.count=1;
```

[3] Здесь и далее, приводя программный код для языка C++, мы подразумеваем, что символьные строки хранятся как объекты типа **string**.

использовать при обработке данных:

```
B.count--;          // одну книгу взяли
if( B.count==0)
   puts( "Этих книг больше нет!" );
```

и выводить на экран:

```
printf( "%s %s. %d шт.", B.author, B.title, B.count );
```

и на языке C++

```
cout << B.author << " " << B.title << ". " << B.count << " шт.";
```

Работа с файлами

В программах, которые работают с данными на диске, бывает нужно читать массивы структур из файла и записывать в файл. Конечно, можно хранить структуры в текстовых файлах, например, записывая все поля одной структуры в одну строчку и разделяя их каким-то символом-разделителем, который не встречается внутри самих полей.

Но есть более грамотный способ, который позволяет выполнять файловые операции проще и надёжнее (с меньшей вероятностью ошибки). Для этого нужно использовать *двоичные* файлы, в которых данные хранятся в том же формате, что и в оперативной памяти компьютера.

Поскольку при работе со структурами, содержащими поля типа **string**, возникают серьезные трудности, связанные со скрытием их реального устройства, мы будем рассматривать только вариант структур в стиле языка C, который можно применять также и в C++.

Указатель на файл объявляется обычным образом:

```
FILE *Fout;
```

Но при открытии двоичного файла после режима (**"r"** – чтение, **"w"** – запись, **"a"** – добавление в конец) добавляется еще буква **b** (от англ. *binary* – двоичный):

```
Fout=fopen( "books.dat", "wb" );
```

Для записи в двоичный файл используют функцию **fwrite**, ей нужно передать 4 аргумента:

1) адрес первого блока данных (структуры);
2) размер одной структуры;
3) количество структур;
4) указатель на файл.

Например, записать в файл структуру **B** можно так:

```
fwrite( &B, sizeof(B), 1, Fout );
```

Напомним, что знак & обозначает операцию взятия адреса объекта. А так записываются 12 первых структур из массива **Books**:

```
fwrite( &Books[0], sizeof(TBook), 12, Fout );
```

После окончания записи нужно закрыть файл:

```
fclose( Fout );
```

Прочитать из файла одну структуру и вывести её поля на экран можно следующим образом:

```
FILE *Fin;
Fin=fopen( "books.dat", "rb" );
fread( &B, sizeof(B), 1, Fin );
printf( "%s %s. %d шт.", B.author, B.title, B.count );
fclose( Fin );
```

Функция **fread** принимает те же аргументы, что и **fwrite**. Можно прочитать сразу несколько структур (в данном случае – 5) в массив:

```
fread( &Books[0], sizeof(TBook), 5, Fin );
```

Если же число структур неизвестно, можно использовать значение, которое возвращает функция **fread**: оно равно количеству фактически прочитанных структур. Чтение останавливается, когда прочитано столько структур, сколько требовалось, или закончился файл:

```
const int N=100;
int M;
...
M=fread( &Books[0], sizeof(TBook), N, Fin );
printf( "Прочитано %d структур.", M );
```

Здесь мы пытаемся прочесть из файла максимально возможное количество структур (**N**), но это может не получиться, если файл закончится раньше. В этом случае в переменную **M** запишется количество фактически прочитанных блоков.

В языке C++ можно работать с двоичными файлами через потоки, только при открытии потока нужно установить режим **ios::binary** (от англ. *binary* – двоичный). Этот поток открывается для записи двоичных данных:

```
#include <fstream>
...
ofstream Fout;
Fout.open ( "books.dat", ios::binary );
```

Теперь в него можно записать структуру **B**:

```
Fout.write ( (char*) &B, sizeof(TBook) );
```

Параметры метода **write** – это адрес блока памяти, откуда взять данные для записи, и количество байтов, которые нужно записать. Поскольку первый параметр метода **write** должен относиться к типу «указатель на символ», мы сразу преобразуем к этому типу адрес структуры.

А так записываются в файл 12 первых структур из массива **Books**:

```
Fout.write ( (char*) Books, 12*sizeof(TBook) );
```

Напомним, что вместо адреса начала массива (**&Books[0]**) можно указать просто имя массива (**Books**). После окончания записи нужно закрыть файл:

```
Fout.close ();
```

Прочитать из двоичного файла одну структуру и вывести её поля на экран можно следующим образом:

```
ifstream *Fin;
Fin.open ( "books.dat", ios::binary );
Fin.read ( (char*) &B, sizeof(B) );
cout << B.author << " " << B.title << ". " << B.count << " шт.";
Fin.close ();
```

Метод **read** принимает те же аргументы, что и **write**. Можно прочитать сразу несколько структур (в данном случае – 5) в массив:

```
Fout.read ( (char*) Books, 5*sizeof(TBook) );
```

Чтение останавливается, когда прочитано столько структур, сколько требовалось, или закончился файл.

Если нужно прочитать неизвестное количество структур (но не более 100), мы пытаемся читать все 100, а фактическое количество прочитанных структур определяем с помощью метода **gcount**, который возвращает число прочитанных байтов. Если разделить это число на размер одной структуры, получаем количество прочитанных структур:

```
const int N=100;
int M;
...
Fin.read ( (char*) Books, N*sizeof(TBook) );
M = Fin.gcount() / sizeof(TBook);
cout << "Прочитано " << M << " структур.";
```

Сортировка

Для сортировки массива структур применяют те же методы, что и для массива простых переменных. Структуры обычно сортируют по возрастанию или убыванию одного из полей, которое называют *ключевым полем* или *ключом*, хотя можно, конечно, использовать и сложные условия, зависящие от нескольких полей (составной ключ).

Отсортируем массив **Books** (типа **TBook**) по фамилиям авторов в алфавитном порядке. В данном случае ключом будет поле **author**. Предположим, что фамилия состоит из одного слова, а за ней через пробел следуют инициалы. Тогда сортировка методом пузырька на языке C++ выглядит так:

```
for ( i=0; i<N-1; i++ )
  for ( j=N-2; j>=i; j-- )
```

```
    if ( Books[j].author > Books[j+1].author )
      {
      B = Books[j];
      Books[j] = Books[j+1];
      Books[j+1] = B;
      }
```

Здесь **i** и **j** – целочисленные переменные, а **B** – вспомогательная структура типа **TBook**.

Как вы знаете из курса 10 класса, при сравнении двух символьных строк они рассматриваются посимвольно до тех пор, пока не будут найдены первые отличающиеся символы. Далее сравниваются коды этих символов по кодовой таблице. Так как код пробела меньше, чем код любой русской (и латинской) буквы, строка с фамилией «Волк» окажется выше в отсортированном списке, чем строка с более длинной фамилией «Волков», даже с учетом того, что после фамилии есть инициалы. Если фамилии одинаковы, сортировка происходит по первой букве инициалов, затем – по второй букве.

В языке С для сравнения строк используется функция **strcmp**, которая возвращает «разность» строк, то есть разность кодов первых несовпадающих символов этих строк. Если разность строк положительна («бóльшая» строка стоит перед меньшей), их нужно поменять местами:

```
for ( i = 0; i < N-1; i++ )
  for ( j = N-2; j >= i; j-- )
    if ( strcmp ( Books[j].author, Books[j+1].author ) > 0 )
      {
      B = Books[j];
      Books[j] = Books[j+1];
      Books[j+1] = B;
      }
```

Возможно, что структуры требуется отсортировать так, чтобы не перемещать их в памяти. Например, они очень большие, и многократное копирование целых структуры занимает много времени. Или по каким-то другим причинам перемещать структуры нельзя. При таком ограничении нужно вывести на экран или в файл отсортированный список. В этом случае применяют «сортировку по указателям», в которой используется дополнительный массив переменных специального типа – указателей (вспомните материал главы 8 учебника для 10 класса).

Для сортировки массива **Books** нужно разместить в памяти массив таких указателей и ещё одну вспомогательную переменную-указатель, которая будет использована при сортировке:

```
TBook *p[N], *p1;
```

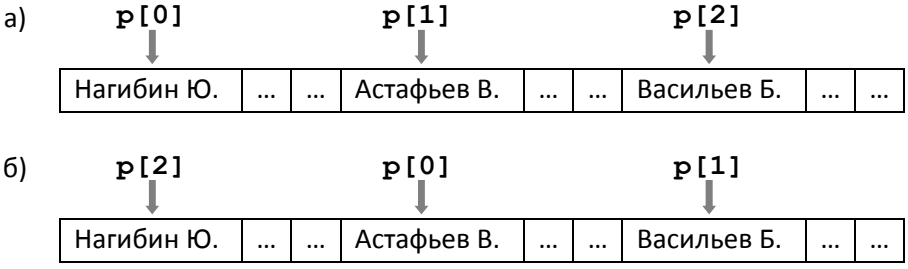
Следующий этап – расставить указатели так чтобы **i**-ый указатель был связан с **i**-ой структурой из массива **Books**:

```
for ( i = 0; i < N; i++ ) p[i] = &Books[i];
```

Знак & обозначает операцию взятия адреса, то есть в указатель записывается адрес структуры.

Для того, чтобы от указателя перейти к полю объекту, на который он ссылается, используют оператор **->**. Например, в нашем случае (после показанной выше начальной установки указателей) запись **p[i]->title** обозначает то же самое, что и **Books[i].title**.

Теперь можно перейти к сортировке. Рассмотрим идею на примере массива из трех структур. Сначала указатели стоят по порядку (рис. а). В результате сортировки нужно переставить их так, чтобы **p[0]** указывал на первую по счёту структуру в отсортированном списке, **p[1]** – на вторую и т.д. (рис. б).



Обратите внимание, что при этом сами структуры в памяти не перемещались.

При сортировке обращение к полям структур идет через указатели, и меняются местами тоже только указатели, а не сами структуры:

```
for ( i=0; i<M-1; i++ )
  for ( j=M-2; j>=i; j-- )
    if ( strcmp ( p[j]->author, p[j+1]->author ) > 0 )
      {
      p1=p[j]; p[j]=p[j+1];
      p[j+1]=p1;
      }
```

Здесь использован метод пузырька, а **p1** – это переменная-указатель на объект типа **TBook**, которая служит для перестановки указателей.

Теперь можно вывести отсортированные данные, обращаясь к ним через указатели (а не через массив **Books**):

```
for ( i=0; i<M; i++ )
  printf ( "%s, %s, %d\n",
           p[i]->author, p[i]->title, p[i]->count );
```

Контрольные вопросы

1. Что такое структура? В чём её отличие от массива?
2. В каких случаях использование структур дает преимущества? Какие именно?
3. Как объявляется новый тип данных в языках C и C++? Выделяется ли при этом память?
4. Как обращаются к полю структуры? Расскажите о точечной нотации.
5. Как определить, сколько байт памяти выделяется на структуру?
6. Какие проблемы могут возникнуть при использовании структур с символьными строками в C++? Чем они вызваны?
7. Что такое двоичный файл? Чем он отличается от текстового?
8. Как работать с типизированными файлами?
9. Как можно сортировать структуры?
10. В каких случаях при сортировке желательно не перемещать структуры в памяти?
11. Что такое указатель?
12. Как записать в указатель адрес переменной или элемента массива?
13. Как обращаться к полям структуры через указатель?
14. Как используются указатели при сортировке?

Задачи

1. Опишите структуру, в которой хранится информация о
   а) видеозаписи;
   б) сотруднике фирмы «Рога и Копыта»;
   в) самолёте;
   г) породистой собаке.
2. Постройте программу, которая работает с базой данных в виде двоичного файла. Ваша СУБД (система управления базой данных) должна иметь следующие возможности:
   а) просмотр записей;
   б) добавление записей;
   в) удаление записей;
   г) сортировка по одному из полей (через указатели).

§ 40. Динамические массивы

Что это такое?

Когда мы объявляем массив, место для него выделяется во время трансляции, то есть до выполнения программы. Такой массив называется *статическим*. В то же время иногда размер данных заранее неизвестен.

Например, в файле записан массив чисел, которые нужно отсортировать. Их количество неизвестно, но известно, что такой массив помещается в оперативную память. В этом случае есть два варианта: 1) выделить заранее максимально большой блок памяти, и 2) выделять память уже *во время выполнения программы* (то есть динамически), когда стал известен необходимый размер массива.

Другой пример – задача составления *алфавитно-частотного словаря*. В файле находится список слов. Нужно вывести в другой файл все различные слова, которые встречаются в файле, и определить, сколько раз встречается каждое слово. Здесь проблема состоит в том, что нужный размер массива можно узнать только тогда, когда все различные слова будут найдены и таком образом задача решена. Поэтому нужно сделать так, чтобы массив мог «расширяться» в ходе работы программы.

Эти задачи приводят к понятию «динамических структур данных», которые позволяют во время выполнения программы:

- создавать новые объекты в памяти;
- изменять их размер;
- удалять их из памяти, когда они не нужны.

Память под эти объекты выделяется в специальной области, которую обычно называют «*кучей*» (англ. *heap*).

Размещение в памяти

Задача 2. *Ввести с клавиатуры целое значение **N**, затем – **N** целых чисел, и вывести на экран эти числа в порядке возрастания.*

Поскольку для сортировки все числа необходимо удерживать в памяти, нужно заводить массив, в который будут записаны все элементы. Поэтому алгоритм решения задачи на псевдокоде выглядит так:

```
прочитать данные из файла в массив
отсортировать их по возрастанию
вывести массив на экран
```

Все эти операции для обычных массивов подробно рассматривались в курсе 10 класса, поэтому здесь мы остановимся только на главной проблеме: как разместить в памяти массив, размер которого до выполнения программы неизвестен?

Для выделения памяти во время работы программы в языках C и C++ используют указатели. например, если необходим новый массив целых чисел, нужно объявить указатель на целую переменную:

```
int *A;
```

Это еще не массив, поскольку память для его элементов не выделена. Чтобы выделить память на **N** элементов, в языке C используется функция **calloc** (для её использования нужно подключить заголовочный файл **stdlib.h** – стандартную библиотеку):

```
#include <stdlib.h>
...
A = (int*) calloc ( N, sizeof(int) );
```

Функции **calloc** передаётся два аргумента: количество элементов массива и размер одного элемента. Запись **(int*)** говорит о том, что результат, который вернет функция **calloc**, нужно преобразовать к типу «*указатель на целую переменную*».

В C++ аналогичную операцию выполняет оператор **new**:

```
A = new int[N];
```

После выделения памяти массив можно использовать так же, как обычный статический массив (элементы нумеруются с нуля):

```
for ( i=0; i<N; i++ )
  {
  A[i]=i;
  printf ( "%d ", A[i] );
  }
```

Такие массивы можно использовать в процедурах и функциях так же, как и статические массивы.

Как только динамический массив стал не нужен, можно освободить выделенную память: в языке С для этого применяется процедура **free**, а в C++ - оператор **delete**:

```
free ( A );
```

```
delete [] A;
```

Квадратные скобки после оператора **delete** в языке C++ обозначают, что удаляется массив, а не одна переменная.

В языке C++ удобно использовать класс **vector** из стандартной библиотеки шаблонов (STL = *Standard Template Library*), который фактически представляет собой динамический массив с возможностью расширения. Такой массив (вектор) для целых чисел (элементов типа **int**) можно объявить так:

```
vector <int> A;
```

Количество элементов в массиве можно определить с помощью метода **size**:

```
cout << A.size();
```

В самом начале в массиве нет ни одного элемента и его размер равен нулю. Метод **push_back** добавляет новый элемент в конец массива:

```
for ( i=0; i<N; i++ )
  A.push_back ( i+1 );
```

Работать с вектором можно так же, как и с обычным массивом:

```
for ( i=0; i<A.size(); i++ )
  cout << A[i] << " ";
```

Для работы с классом **vector** нужно подключить заголовочный файл **vector**:

```
#include <vector>
```

Динамические матрицы

Похожим образом можно работать и с динамическими матрицами. Матрица – это «массив массивов», каждую её строку можно рассматривать как линейный массив. Поэтому для выделения в памяти места под матрицу нужен массив указателей. Для удобства введём новый тип данных **pInt** – указатель на переменную типа **int**:

```
typedef int *pInt;
```

Тогда динамическая матрица объявляется как массив таких указателей:

```
pInt *A;
```

Сначала выделяем память под эти указатели (справа указан вариант для C++)

```
A=(pInt*)calloc ( N, sizeof(pInt) );
```

```
A=new pInt[N];
```

Дальше у нас есть выбор: можно выделить единый блок памяти на всю матрицу или выделять блоки отдельно для каждой строки. Рассмотрим сначала первый вариант. В матрице, у которой N строк и M столбцов, должно быть N·M элементов, поэтому выделим блок памяти такого размера и запишем его адрес в указатель A[0]:

```
A[0]=(int*)calloc ( N*M, sizeof(int));
```

```
A[0]=new int[N*M];
```

Теперь нужно расставить остальные указатели так, чтобы они указывали на первые элементы соответствующих строк. Например, указатель **A[1]** должен содержать адрес **A[0]+M**, то есть его нужно сдвинуть относительно начала блока на **M** элементов. Аналогично **A[1]** должен быть равен **A[0]+2*M** и т.д.:

```
for ( i=1; i<N; i++ )
  A[i]=A[i-1]+M;
```

Обратите внимание, что сдвиг указателя задаётся не в байтах, а в размерах переменной, на которую он указывает. В нашем случае **A[i]** – это указатели на переменные типа **int**, поэтому до-

бавление числа **M** к значению указателя означает увеличение хранящегося в нём адреса на **M*sizeof(int)**.

После выделения памяти можно «забыть» о том, что это динамическая матрица, и работать с неё стандартными способами:

```
for ( i=0; i<N; i++ )
  for ( j=0; j<M; j++ )
    A[i][j]=i+j;
```

Для удаления динамической матрицы и освобождения памяти нужно сначала удалить основной блок памяти (под элементы матрицы):

```
free ( A[0] );
```

```
delete [] A[0];
```

а затем удалить массив указателей:

```
free ( A );
```

```
delete [] A;
```

Во втором варианте память выделяется отдельно на каждую строку:

```
for ( i=0; i<N; i++ )
  A[i]=(int*) calloc(M, sizeof(int));
```

```
for ( i=0; i<N; i++ )
  A[i]=new int[M];
```

Соответственно и удалять эти блоки нужно с помощью цикла

```
for( i=0; i<N; i++ )
  free ( A[i] );
```

```
for ( i=0; i<N; i++ )
  delete [] A[i];
```

В случае, если число элементов в разных строках матрицы может быть различно, удобнее использовать именно этот способ.

В языке C++ для работы с динамической матрицей можно использовать тип **vector** из библиотеки STL. Так как матрица – это массив из массивов, можно составить вектор, элементами которого будут вектора (массивы):

```
typedef vector<int> vint;
vector <vint> A;
```

В первой строке введён новый тип данных **vint** – вектор целых чисел, а во второй объявляется вектор с именем **A**, состоящий из элементов типа **vint**.

Сначала с помощью метода **resize** (от англ. *resize* – изменить размер) нужно определить количество строк матрицы:

```
A.resize ( N );
```

а затем в цикле установить размеры каждой строки (обратите внимание, что они могут быть разные!):

```
for ( i=0; i<N; i++ )
  A[i].resize ( M );
```

После этого можно использовать такой объект как обычный массив:

```
for ( i=0; i<N; i++ )
  for ( j=0; j<M; j++ )
    A[i][j]=i+j;
```

Расширение массива

Задача 3. *С клавиатуры вводятся натуральные числа, ввод заканчивается числом 0. Нужно вывести на экран эти числа в порядке возрастания.*

Как и в предыдущей задаче, для сортировки необходимо предварительно сохранить все числа в оперативной памяти (в массиве). Но проблема в том, что размер этого массива становится известен только тогда, когда будут введены все числа. Что же делать?

В первую очередь приходит в голову такой вариант: при вводе каждого следующего ненулевого числа расширять массив на 1 элемент и записывать введённое число в последний элемент массива:

```
прочитать x
пока x != 0
  {
  расширить массив на 1 элемент
  записать x в последний элемент
```

```
    прочитать x
    }
```

Здесь **x** – это целая переменная. К счастью, при таком расширении массива значения всех существующих элементов сохраняются.

Для расширения массива в языке С используется функция **realloc** (от англ. *reallocate* – перераспределить память):

```
A = (int*) realloc( A, N*sizeof(int) );
```

Здесь **N** – это новая длина массива. К счастью, значения всех существующих элементов будут скопированы в новый массив. Таким образом, получаем:

```
N = 0;
scanf ( "%d", &x );
while ( x!= 0 )
   {
   N ++;
   A = (int*) realloc ( A, N*sizeof(int) );
   A[N-1] = x;
   scanf ( "%d", &x );
   }
```

Чем плох такой подход? Дело в том, что память в куче выделяется блоками. Поэтому при каждом увеличении длины массива последовательно выполняются три операции:

1) выделение блока памяти нового размера;
2) копирование в этот блок значений всех существующих элементов;
3) удаление «старого» блока памяти из кучи.

Видно, что «накладные расходы» очень велики, то есть мы заставляем компьютер делать слишком много вспомогательной работы.

Ситуацию можно немного исправить, если увеличивать массив не каждый раз, а скажем, после каждых 10 введённых элементов. То есть, когда все свободные элементы массива заполнены, к нему добавляется ещё 10 новых ячеек. Фактический размер массива будем хранить в отдельной переменной **length**:

```
length = 10;
A = (int*) calloc ( 10, sizeof(int) );
N = 0;
scanf ( "%d", &x );
while ( x!= 0 )
   {
   N ++;
   if ( N > length )
      {
      length += 10;
      A = (int*) realloc ( A, length*sizeof(int) );
      }
   A[N-1] = x;
   scanf ( "%d", &x );
   }
```

Здесь целая переменная **N** – это счётчик введенных чисел. В самом начале выделяется массив из 10 элементов.

В языке С++ простейший вариант решения этой задачи – использование типа **vector** из стандартной библиотеки. Там расширение массива происходит автоматически при добавлении нового элемента.

Теперь, когда все числа записаны в массив, можно отсортировать их любым известным методом, например, методом «пузырька» или с помощью «быстрой сортировки» (эти алгоритмы изучались в 10 классе). Закончить программу вы можете самостоятельно.

? Контрольные вопросы

1. Приведите примеры задач, в которых использование динамических массивов даёт преимущества (какие именно?).
2. Что такое динамические структуры данных? Где выделяется память под эти данные?
3. Как объявить в программе динамический массив и задать его размер?
4. Как расширить массив в ходе работы программы? Не потеряются ли при этом уже записанные в нём данные?
5. Как определить границы изменения индексов динамического массива? Нужно ли хранить его размер в отдельной переменной?
6. Как удалить массив из памяти?
7. Как разместить в памяти динамическую матрицу? Какие варианты есть в языках C и C++?
8. Как построить динамическую матрицу, в которой строки имеют разную длину?
9. Как передать динамический массив в подпрограмму?
10. Подумайте, как сохранять динамические массивы и матрицы в двоичных файлах?

Задачи

1. Напишите полные программы для решения задач, рассмотренных в тексте параграфа.
2. Введите с клавиатуры число **N** и вычислите все простые числа в диапазоне от 2 до **N**, используя решето Эратосфена.
3. Введите с клавиатуры число **N** и запишите в массив первые **N** простых чисел.
4. Введите с клавиатуры число **N** и запишите в массив первые **N** чисел Фибоначчи (напомним, что они задаются рекуррентной формулой $F_n = F_{n-1} + F_{n-2}$, $F_1 = F_2 = 1$).
5. Напишите функцию, которая находит максимальный элемент переданного ей динамического массива.
6. Напишите подпрограмму, которая находит максимальный и минимальный элементы переданного ей динамического массива (используйте изменяемые параметры).
7. Напишите рекурсивную функцию, которая считает сумму элементов переданного ей динамического массива.
8. Напишите функцию, которая сортирует значения переданного ей динамического массива, используя алгоритм «быстрой сортировки» (см. учебник 10 класса).

§ 41. Списки

Что такое список?

Задача 4. *В файле находится список слов, среди которых есть повторяющиеся. Каждое слово записано в отдельной строке. Построить алфавитно-частотный словарь: все различные слова должны быть записаны в другой файл в алфавитном порядке, справа от каждого слова указано, сколько раз оно встречается в исходном файле.*

Для решения задачи нам нужно составить *список*, в котором хранить пары «слово – количество». Список составляется по мере чтения файла, то есть это динамическая структура.

Список – это упорядоченный набор элементов одного типа, для которых введены операции вставки (включения) и удаления (исключения).

Обычно используют *линейные* списки, в которых для каждого элемента (кроме первого) можно указать предыдущий, а для каждого элемента, кроме последнего, – следующий.

Вернемся к нашей задаче построения алфавитно-частотного словаря. Алгоритм, записанный в виде псевдокода, может выглядеть так:

```
пока есть слова в файле
  {
  прочитать очередное слово
  если оно есть в списке то
```

```
    увеличить на 1 счётчик для этого слова
  иначе
    {
    добавить слово в список
    записать 1 в счетчик слова
    }
```

Теперь нужно записать все шаги этого алгоритма с помощью операторов языка программирования. Сначала мы рассмотрим решение этой задачи на языке С, а затем – на языке С++, где очень удобно применить стандартную библиотеку шаблонов (STL).

Использование динамического массива

В нашем случае каждый элемент списка должен содержать пару значений: слово (символьную строку) и счётчик этих слов (целое число). Поэтому элементы списка – это структуры, тип которых можно описать так:

```
typedef struct {
  char word[40];
  int count;
  } TPair;
```

Для организации списка будем использовать динамические массивы.

Здесь ситуация похожа на задачу 2: размер массива становится известен только в конце работы программы. Поэтому требуется динамический массив, состоящий из описанных выше структур. С ним нужно выполнять следующие операции:

- искать заданное слово в списке;
- увеличивать счётчик заданного слова на 1;
- вставлять слово в определённое место списка (так, чтобы сохранить алфавитный порядок).

Как и в предыдущем параграфе, будем расширять размер массива сразу на 10 элементов[4], чтобы не перераспределять память слишком часто.

Объявим структуру-список, новый тип данных **TWordList**:

```
typedef struct {
  TPair *data;
  int capacity;
  int size;
  } TWordList;
```

Здесь поле **data** – этот указатель на динамический массив, состоящий из элементов типа **TPair**; **capacity** – размер выделенной памяти, и **size** – фактическое число элементов в списке (может быть меньше, чем **capacity**).

Введём переменную **L** типа **TWordList**:

```
TWordList L;
```

В начале основной программы в списке нет ни одного элементов, но мы сразу выделим в памяти массив элементов типа **TPair** размером 10 элементов:

```
L.size = 0;
L.capacity = 10;
L.data = (TPair*) calloc ( L.capacity, sizeof(TPair) );
```

Объявим переменные, которые будут использоваться в программе:

```
int p;
char s[80];
FILE *F;
```

Здесь **p** и **s** – вспомогательные переменные, а **F** – указатель на файл.

Основной цикл (чтение данных из файла и построение списка) можно записать так (для последующего объяснения строки пронумерованы):

```
F = fopen ( "input.txt", "r" );
while ( 1 == fscanf(F, "%s", s) )    // 1
```

[4] Возможны и другие варианты, например, можно увеличивать размер массива в 2 раза.

```
  {
  p=Find ( L, s );                  // 2
  if ( p>=0 )
    L.data[p].count ++;             // 3
  else
    {
    p=FindPlace ( L, s );           // 4
    InsertWord ( &L, p, s );        // 5
    }
  }
fclose ( F );
```

В строке 1 очередное слово читается из файла в строку **s**. Если чтение заканчивается неудачно, (функция **scanf** возвращает 0 или –1), условие работы цикла нарушается и цикл завершается.

Если слово прочитали удачно, с помощью функции **Find** определяем, есть ли оно в списке (строка 2). Если есть (функция **Find** вернула существующий индекс), увеличиваем его счётчик на 1 (строка 3).

Если слова ещё нет в списке (функция **Find** вернула **-1**), нужно найти место, куда его вставить, так чтобы не нарушился алфавитный порядок. Это делает функция **FindPlace**, которая должна возвращать номер элемента массива, *перед* которым нужно вставить прочитанное слово. Вставку выполняет процедура **InsertWord**.

Когда список готов, остается вывести его в выходной файл:

```
F=fopen ( "output.txt", "w" );
for ( p=0; p<L.size; p++ )
  fprintf ( F, "%s: %d\n", L.data[p].word, L.data[p].count );
fclose ( F );
```

Для каждого элемента списка в файл выводится хранящееся в нём слово и через двоеточие – сколько раз оно встретилось в тексте.

Таким образом, нам остается написать функции **Find** и **FindPlace**, а также процедуру **InsertWord**.

Функция **Find** принимает список и слово, которое нужно искать. Из курса 10 класса вы знакомы с двумя алгоритмами поиска: линейным и двоичным. Здесь для простоты будем использовать линейный поиск. В цикле проходим все элементы, как только очередное слово списка совпало с образцом (функция **strcmp** вернёт 0), возвращаем в качестве результата функции номер этого элемента. Если просмотрены все элементы и совпадения не было, функция вернет **-1**.

```
int Find( TWordList L, char word[] )
{
  int i;
  for ( i=0; i<L.size; i++ )
    if ( 0==strcmp(L.data[i].word, word) )
      return i;
  return -1;
}
```

Здесь встретилось обозначение с двумя точками: **L.data[i].word**. Вспомним, что **L** – это структура-список, у него есть поле-массив **data**. В этом массиве идет обращение к элементу с номером **i**. Этот элемент – структура типа **TPair**, в составе которой есть поле **word**. Таким образом, **L.data[i].word** означает «поле **word** в составе **i**-ого элемента массива **data**, который входит в структуру **L**».

Функция **FindPlace** также принимает в параметрах список и слово. Она находит место вставки нового слова в список, при котором сохраняется алфавитный порядок расположения слов. Результат функции – номер слова, перед которым нужно вставить заданное. Для этого нужно найти в списке слово, которое «больше» заданного. Если такое слово не найдено, новое слово вставляется в конец списка:

```
int FindPlace ( TWordList L, char word[] )
```

```
{
  int i;
  for ( i=0; i<L.size; i++ )
    if ( strcmp(L.data[i].word, word)>0 )
      return i;
  return L.size;
}
```

Процедура **InsertWord** вставляет слово **word** в позицию **k** в список **L**:

```
void InsertWord ( TWordList *pL, int k, char word[] )
{
  int i;
  IncSize ( pL );                          // 1
  for ( i=pL->size-1; i>k; i-- )           // 2
    pL->data[i]=pL->data[i-1];             // 3
  strcpy ( pL->data[k].word, word );       // 4
  pL->data[k].count=1;                     // 5
}
```

Поскольку во время добавления нового элемента список расширяется, то есть изменяется значения полей **size** и (возможно) **capacity**, передавать список в процедуру нужно по адресу (указателю), который здесь назван **pL**.

Для увеличения списка при добавлении нового элемента введена процедура **IncSize**, которая вызывается в строке 1 (мы напишем её позже). Далее в цикле сдвигаем все последние элементы, включая элемент с номером **k**, на одну ячейку к концу массива (строки 2-3). Таким образом, элемент с номером **k** освобождается. В строке 4 в него записывается новое слово, а в строке 5 – счётчик этого слова устанавливается равным 1.

Процедура **IncSize** увеличивает размер списка на 1 элемент. Когда нужный размер становится больше, чем размер динамического массива, массив расширяется сразу на 10 элементов:

```
void IncSize ( TWordList *pL )
{
  pL->size++;
  if ( pL->size>pL->capacity ) {
    pL->capacity+=10;
    pL->data=
     (TPair*)realloc ( pL->data, sizeof(TPair)*pL->capacity );
    }
}
```

Процедура **IncSize** в программе должна располагаться выше вызывающей её процедуры **InsertWord**.

Приведем окончательную структуру программы:

```
// объявления типов TPair и TWordList
// процедуры и функции
main()
{
  TWordList L;
  int p;
  char s[80];
  FILE *F;
  L.size=0;
  L.capacity=10;
  L.data=(TPair*)calloc ( L.capacity, sizeof(TPair) );
    // основной цикл: чтение списка слов
    // вывод результата в файл
}
```

Блоки, обозначенные комментариями, уже были написаны ранее в этом параграфе.

Заметим, что если известно максимальное количество разных слов в файле (скажем, не более 1000), то же самое можно сделать и на основе обычного (статического) массива, в котором память выделена заранее на максимальное число элементов.

Модульность

При разработке больших программ нужно разделить работу между программистами так, чтобы каждый делал свой независимый блок (*модуль*). Все подпрограммы, входящие в модуль, должны быть связаны друг с другом, но слабо связаны с другими процедурами и функциями. В нашей программе в отдельный модуль можно вынести все операции со списком слов.

Модуль в языках C и C++ представляет собой отдельный файл, в котором нет основной программы **main**, а есть только функции и процедуры. Например, мы можем построить модуль **wordlist.c**, такого содержания

```
void IncSize ( TWordList *pL )
{
  ...
}
int Find ( TWordList L, char word[] )
{
  ...
}
int FindPlace ( TWordList L, char word[] )
{
  ...
}
void InsertWord ( TWordList *pL, int k, char word[] )
{
  ...
}
```

Здесь многоточие обозначает операторы, которые записаны в теле подпрограмм (см. выше).

Итак, мы фактически разбили программу на две части, поместив их в разные файлы. При этом возникают две проблемы:

1) как собрать эти две части в один исполняемый файл?
2) как сообщить основной программе, что функции **IncSize**, **InsertWord** и др. существуют и расположены в другом файле?

Чтобы объединить несколько исходных файлов в одну программу, используются *проекты*. В среде программирования, с которой вы работаете, нужно создать проект и включить в него файл с основной программой, а также файл **wordlist.c**.

Для решения второй проблемы в языке C используются так называемые *заголовочные* файл, которые по традиции имеют расширение **.h** (от англ. *header* –заголовок). В этом файле объявляются типы данных, функции и процедуры. В нашем случае заголовочный файл **wordlist.h** может выглядеть так:

```
typedef struct {
        char word[40];
        int count;
      } TPair;
typedef struct {
        TPair *data;
        int capacity;
        int size;
      } TWordList;
void IncSize ( TWordList *pL );
int Find ( TWordList L, char word[] );
int FindPlace ( TWordList L, char word[] );
void InsertWord ( TWordList *pL, int k, char word[] );
```

Обратите внимание, что здесь функции и процедуры только объявляются, от них остались только заголовки, которые заканчиваются точкой с запятой в конце. Это так называемый *интерфейс модуля*, то есть та информация, которую знают о нём другие модули. Действительно, основную программу интересуют только входные и выходные данные функций и процедур, но абсолютно не интересует, как именно они выполняют свои функции.

Заголовочный файл нужно подключить к основной программе и к модулю **wordlist.c** с помощью директивы **include**:

```
#include "wordlist.h"
```

Мы уже много раз использовали этот приём, подключая заголовочные файлы стандартных библиотек языков C и C++. Однако, на этот раз вы наверняка заметили, что имя файла заключено не в угловые скобки, как обычно, а в кавычки. Это означает, что транслятор будет искать файл не в своём стандартном каталоге с заголовочными файлами, а в текущем каталоге, где находятся все файлы проекта.

Структура модуля в чём-то подобна айсбергу: всем видна только надводная часть (интерфейс, содержимое заголовочного файла), а значительно более весомая подводная часть (*реализация*, код процедур и функций) скрыта. За счёт этого все, кто используют модуль, могут не думать о том, как именно он выполняет свою работу. Это один из приёмов, которые позволяют справляться со сложностью больших программ.

Разделение программы на модули облегчает понимание и совершенствование программы, потому что каждый модуль можно разрабатывать, изучать и оптимизировать независимо от других. Кроме того, такой подход ускоряет трансляцию больших программ, так как каждый модуль транслируется отдельно, причём только в том случае, если он был изменён.

Использование контейнера map

Теперь покажем, как решить ту же задачу с помощью языка C++. Стандартная библиотека шаблонов (STL) содержит тип данных **map** – так называемый *ассоциативный массив* или *словарь*, в котором индексом элемента может быть не только число, но и любой другой тип данных, например, символьная строка. В задаче построения алфавитно-частотного словаря элемент массива – целое число (тип **int**), а искать его нам нужно по символьной строке типа **string** (слову). В этом случае ассоциативный массив объявляется так:

```
map <string,int> L;
```

Для работы с ассоциативными массивами нужно подключить заголовочный файл **map**:

```
#include <map>
```

Что можно сделать с таким массивом? Во-первых, можно определить, сколько раз строка уже встречалась:

```
int p=L.count ( s );
```

Значение переменной **p** будет равно 0, если этого слова еще нет в словаре и 1, если оно уже было добавлено. Для того, чтобы увеличить счётчик для данного слова, можно обратиться к элементу по символьному индексу:

```
L[s] ++;
```

а для вставки элемента в массив используется метод **insert** (от англ. *insert* – вставка):

```
L.insert ( pair <string,int> (s, 1) );
```

В качестве аргумента этому методу передается пара «строка-целое» (**pair<string,int>**), составленная из слова **s** и числа 1 (первое вхождение этого слова, счётчик равен 1).

Таким образом, основной цикл получается очень простой:

```
while ( Fin >> s ) {
  int p;
  p = L.count ( s );
  if ( p==1 ) L[s] ++;
  else        L.insert ( pair <string,int> (s,1) );
  }
```

Здесь **Fin** – это входной файловый поток (см. главу 8 из учебника для 10 класса), а **s** – переменная типа **string**. Запись **while(Fin>>s)** обозначает «пока чтение очередного слова из файла завершается удачно».

Более того, это цикл можно ещё упростить так:

```
while ( Fin >> s ) L[s] ++;
```

Дело в том, что при обращении к неизвестному элементу в контейнере **map** он будет создан автоматически! Это значит, что если прочитанного слова еще не было в словаре, создается новый элемент, в него записывается это слово и счётчик устанавливается в нуль. Затем мы сразу увеличиваем счётчик до 1 с помощью оператора **++**.

Осталась одна нерешённая задача: вывести результаты в файл. Контейнер **map** позволят легко найти элемент по символьному индексу, но как перебрать *все* элементы, добавленные в контейнер?

Для этого служат *итераторы* (или курсоры) – специальные объекты, которые позволяют перебрать все элементы контейнера. Итератор указывает на текущий объект в ассоциативном массиве, с которым мы можем что-то сделать, например, вывести его данные в файл.

Итератор для нашего контейнера нужно объявить так:

```
map <string,int>::iterator it;
```

Этот итератор предназначен для контейнеров, состоящих из пар «строка-целое», к которым относится и наш контейнер **L**. Для того, чтобы установить итератор на первый по счёту элемент контейнера, напишем

```
it=L.begin();
```

Итератор – это специальный указатель на элемент. Поэтому обращаться к полям этого элемента нужно так же, как к полям структуры по указателю, с помощью оператора **->**. У элемента контейнера **map** два поля, первое (с именем **first**) – строка, второе (с именем **second**) – целое число. Поэтому вывести в выходной поток **Fout** данные по очередному элементу можно так:

```
Fout << it->first << ": " << it->second;
```

Оператор **++** сдвигает итератор к следующему элементу. Если значение итератора после очередного продвижения совпало с **L.end()**, значит все элементы коллекции уже просмотрены. Таким образом, цикл, который выводит данные по всем словам из словаря, выглядит так:

```
for ( it=L.begin(); it!=L.end(); it++ )
  Fout << it->first << ": " << it->second << endl;
```

Теперь можно записать всю программу целиком:

```
#include <iostream>
#include <fstream>
#include <map>
using namespace std;
main ()
{
  ifstream Fin;
  ofstream Fout;
  string s;
  map <string,int> L;

  Fin.open ( "input.txt" );
  while ( Fin >> s )
    L[s] ++;
  Fin.close();

  Fout.open ( "output.txt" );
  map <string,int>::iterator it;
  for ( it=L.begin(); it!=L.end(); it++ )
```

```
    Fout << it->first << ": " << it->second << endl;
  Fout.close();
}
```

Конечно, вы обратили внимание, что программа на C++ получилась намного короче, чем на языке C. Такой эффект вызван только тем, что мы использовали готовый контейнер **map** из STL, в котором уже запрограммированы все необходимые функции.

Связные списки

Линейный список иногда представляется в программе в виде *связного списка*, в котором каждый элемент может быть размещён в памяти в произвольном месте, но должен содержать ссылку (указатель) на следующий элемент. У последнего элемента эта ссылка нулевая (**NULL**), она показывает, что следующего элемента нет. Кроме того, нужно хранить где-то (в указателе **Head**) адрес первого элемента («головы») списка, иначе список будет недоступен:

Связный список часто рассматривают как рекурсивную структуру данных. Рекурсивное определение может быть дано так:

1) пустой список (состоящий из 0 элементов) – это список;
2) список – это узел и связанный с ним список.

Здесь вторая часть определяет рекурсию, а первая – условие окончания рекурсии.

Если замкнуть связный список в кольцо, так чтобы последний элемент содержал ссылку на первый, получается *циклический список*:

Поскольку элементы связного списка содержат ссылки только на следующий элемент, к предыдущему перейти нельзя. Поэтому перебор возможен только в одном направлении. Этот недостаток устранен в двусвязном списке, где каждый элемент хранит адрес как следующего, так и предыдущего элемента:

Для такого списка обычно хранится два адреса: «голова» списка (указатель **Head**) и его «хвост» (указатель **Tail**). Можно организовать и циклический двусвязный список. Использование двух указателей для каждого элемента приводит к дополнительному расходу памяти и усложнению всех операций со списком, потому что при добавлении и удалении элемента нужно правильно расставить оба указателя.

Преимущество связных списков состоит в том, что для них очень быстро выполняются операции вставки и удаления элементов, в то время как поиск нужного элемента требует перебора (прохода по списку).

Применение связных списков приводит к более сложным алгоритмам, чем работа с динамическими массивами, поэтому рассматривать соответствующие программы мы не будем. Заметим, что в библиотеке STL существует специальный тип данных **list** для работы со связными списками. Однако в большинстве случаев вместо него лучше использовать тип **vector**.

? *Контрольные вопросы*

1. Что такое список? Какие операции он допускает?

2. Верно ли, что элементы в списке упорядочены?
3. Какой метод поиска в списке можно использовать? Обсудите разные варианты.
4. Как добавить элемент в линейный список, сохранив заданный порядок сортировки?
5. Как можно представить список в программе? В каких случаях для этого можно использовать обычный массив?
6. Объясните запись **`L.data[i].word`**.
7. Что такое модуль? Зачем используют модули?
8. Как оформляется текст модуля? Как по нему отличить модуль от основной программы?
9. Что такое заголовочный файл? Как подключить заголовочный файл, находящийся в текущем каталоге?
10. Можно ли все переменные и подпрограммы поместить в заголовочный файл? Чем плохо такое решение?
11. Что такое связный список?
12. Дайте рекурсивное определение связного списка и объясните его.
13. Что такое циклический список? Попытайтесь придумать задачу, где после завершения просмотра списка нужно начать просмотр заново.
14. Сравните односвязный и двусвязный списки. В чем достоинства и недостатки одного и второго типов?

Задачи

1. Постройте программу, которая составляет алфавитно-частотный словарь для заданного файла со списком слов. Используйте модуль, содержащий все операции со списком.
2. *В предыдущей программе измените функцию **`Find`** так, чтобы в ней использовался двоичный поиск.
3. В предыдущей программе объедините функции **`Find`** и **`FindPlace`**, заменив их на одну функцию. Если слово найдено в списке, функция работает так же, как **`Find`**: возвращает номер слова в списке. Если слово не найдено, функция должна вернуть отрицательное число: номер элемента массива, перед которым нужно вставить слово, со знаком минус.
4. *В предыдущей задаче вывести все найденные слова в файл в порядке убывания частоты, то есть в начале списка должны стоять слова, которые встречаются в файле чаще всех.
5. Используя литературу и материалы Интернете, изучите работу контейнера **`list`** из библиотеки STL. Сравните контейнеры **`list`** и **`vector`**, укажите их достоинства и недостатки.

§ 42. Стек, очередь, дек

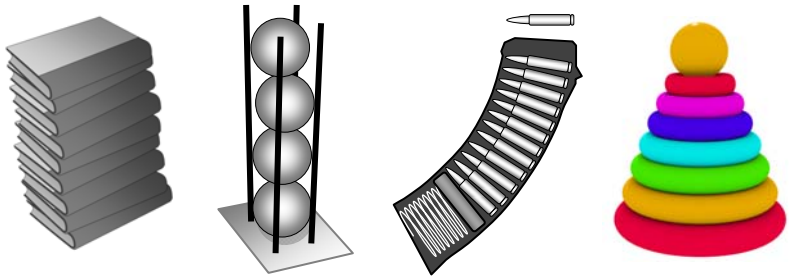
Что такое стек?

Представьте себе стопку книг (подносов, кирпичей и т.п.). С точки зрения информатики её можно воспринимать как список элементов, расположенных в определенном порядке. Этот список имеет одну особенность – удалять и добавлять элементы можно только с одной («верхней») стороны. Действительно, для того, чтобы вытащить какую-то книгу из стопки, нужно сначала снять все те книги, которые находятся на ней. Положить книгу сразу в середину тоже нельзя.

Стек (англ. *stack* – стопка) – это линейный список, в котором элементы добавляются и удаляются только с одного конца («последним пришел – первым ушел»[5]).

На рисунках показаны примеры стеков вокруг нас, в том числе автоматный магазин и детская пирамидка:

[5] Англ. LIFO = *Last In – First Out*.



Как вы знаете из главы 8 учебника для 10 класса, стек используется при выполнении программ: в этой области оперативной памяти хранятся адреса возврата из подпрограмм; параметры, передаваемые функциям и процедурам, а также локальные переменные.

Задача 5. *В файле записаны целые числа. Нужно вывести их в другой файл в обратном порядке.*

В этой задаче очень удобно использовать стек. Для стека определены две операции:

- добавить элемент на вершину стека (англ. *push* – втолкнуть);
- получить элемент с вершины стека и удалить его из стека (англ. *pop* – вытолкнуть).

Запишем алгоритм решения на псевдокоде. Сначала читаем данные и добавляем их в стек:

```
пока файл не пуст
  {
  прочитать x
  добавить x в стек
  }
```

Теперь верхний элемент стека – это последнее число, прочитанное из файла. Поэтому остается «вытолкнуть» все записанные в стек числа, они будут выходить в обратном порядке:

```
пока стек не пуст
  {
  вытолкнуть число из стека в x
  записать x в файл
  }
```

Как и в предыдущем параграфе, сначала мы разберём решение этой задачи на языке С («вручную»), а затем покажем, как можно сделать то же самое с помощью готового контейнера **stack** из библиотеки STL.

Использование динамического массива

Поскольку стек – это линейная структура данных с переменным количеством элементов, для создания стека в программе мы можем использовать динамический массив. Конечно, можно организовать стек из обычного (статического) массива, но его будет невозможно расширить сверх размера, выделенного при трансляции.

Для рассмотренной выше задачи 5 структура-стек содержит динамический целочисленный массив, размер этого массива и количество используемых в нем элементов:

```
typedef struct {
  int *data;
  int capacity;
  int size;
  } TStack;
```

Будем считать, что стек «растет» от начала к концу массива, то есть вершина стека – это последний элемент.

Для работы со стеком нужны две подпрограммы:

- процедура **Push**, которая добавляет новый элемент на вершину стека;
- функция **Pop**, которая возвращает верхний элемент стека и убирает его из стека.

Приведем эти подпрограммы:

```
void Push ( TStack *pS, int x )
{
  pS->size ++;
```

```
  if ( pS->size > pS->capacity ) {
    pS->capacity += 10;
    pS->data =
        (int*) realloc ( pS->data, sizeof(int)*pS->capacity  );
    }
  pS->data[pS->size-1] = x;
}
int Pop ( TStack *pS )
{
  pS->size --;
  return pS->data[pS->size];
}
```

Обратите внимание, что здесь структура типа **TStack** изменяется внутри подпрограмм, поэтому этот параметр передаётся по адресу (указателю).

Заметим, что если нам понадобится стек, которых хранит данные другого типа (например, символы, вещественные числа или структуры), в объявлении типа и в приведенных подпрограммах нужно просто заменить **int** на нужный тип данных.

Кроме того, введём процедуру **InitStack**, которая заполняет поля структуры начальными значениями (выполняет *инициализацию* стека):

```
void InitStack ( TStack *pS, int capacity )
{
  pS->data = (int*) calloc ( capacity, sizeof(int) );
  pS->capacity = capacity;
  pS->size = 0;
}
```

Теперь несложно написать цикл ввода данных в стек из файла:

```
InitStack ( &S, 10 );
while ( 1 == fscanf(Fin, "%d", &x) )
  Push ( &S, x );
```

Здесь **S** – переменная типа **TStack**; **Fin** – файловая переменная, связанная с файлом, открытым на чтение; **x** – целая переменная. Вывод результата в файл выполняется так:

```
while ( S.size > 0 )
  {
  x = Pop( &S );
  fprintf ( Fout, "%d\n", x );
  }
```

Здесь **i** – целая переменная, а **Fout** – файловая переменная, связанная с файлом, открытым на запись.

Использование контейнера stack

Библиотека STL содержит стандартный тип данных **stack**, который можно использовать для хранения элементов любого типа. Для использования стека необходимо подключить заголовочный файл **stack**. Стек для целых чисел объявляется так:

```
#include <stack>
...
stack <int> S;
```

У контейнера **stack** есть готовые методы, которые мы будем использовать:

- **push** – добавление элемента на вершину стека;
- **pop** – снятие элемента с вершины стека;
- **top** – функция, возвращающая верхний элемент стека, не удаляя его;
- **empty** – логическая функция, которая возвращает истинное значение, если стек пуст, и ложное в противном случае.

Полная программа решения задачи реверса массива выглядит так:

```
#include <iostream>
```

```
#include <fstream>
#include <stack>
using namespace std;
main ()
{
  ifstream Fin;
  ofstream Fout;
  stack <int> S;
  int x;
    // чтение данных из файла
  Fin.open ( "input.dat" );
  while ( Fin >> x )
    S.push ( x );
  Fin.close();

    // запись результата в файл
  Fout.open ( "output.dat" );
  while ( !S.empty() )
    {
    Fout << S.top() << endl;
    S.pop();
    }
  Fout.close();
}
```

Эта программа снова получилась такой короткой только потому, что все необходимые функции для работы со стеком уже реализованы в библиотеке STL.

Вычисление арифметических выражений

Вы не задумывались, как компьютер вычисляет арифметические выражения, записанные в такой форме: `(5+15)/(4+7-1)`? Такая запись называется *инфиксной* – в ней знак операции расположен *между* операндами (данными, участвующими в операции). Инфиксная форма неудобна для автоматических вычислений, из-за того, что выражение содержит скобки и его нельзя вычислить за один проход слева направо.

В 1920 году польский математик Ян Лукашевич предложил *префиксную* форму, которую стали называть польской нотацией. В ней знак операции расположен *перед* операндами. Например, выражение `(5+15)/(4+7-1)` может быть записано в виде `/ + 5 15 - + 4 7 1`. Скобок здесь не требуется, так как порядок операций строго определен: сначала выполняются два сложения (`+ 5 15` и `+ 4 7`), затем вычитание, и, наконец, деление. Первой стоит последняя операция. Таким образом, выражение в префиксной форме нужно вычислять с конца.

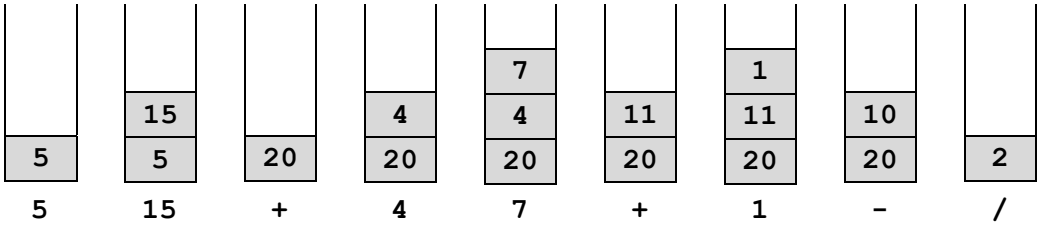
В середине 1950-х годов была предложена *обратная польская нотация* или *постфиксная* форма записи, в которой знак операции стоит *после* операндов:

```
5 15 + 4 7 + 1 - /
```

В этом случае также не нужны скобки, и выражение может быть вычислено за один просмотр с помощью стека следующим образом:

- если очередной элемент – число (или переменная), он записывается в стек;
- если очередной элемент – операция, то она выполняется с верхними элементами стека, и после этого в стек вталкивается результат выполнения этой операции.

Покажем, как работает этот алгоритм (стек «растёт» снизу вверх):



В результате в стеке остается значение заданного выражения.

Скобочные выражения

Задача 6. *Вводится символьная строка, в которой записано некоторое (арифметическое) выражение, использующее скобки трёх типов:* **()**, **[]** и **{}**. *Проверить, правильно ли расставлены скобки.*

Например, выражение **()[{()[]}]** – правильное, потому что каждой открывающей скобке соответствует закрывающая, и вложенность скобок не нарушается. Выражения

[() [[[() [()})(([)]

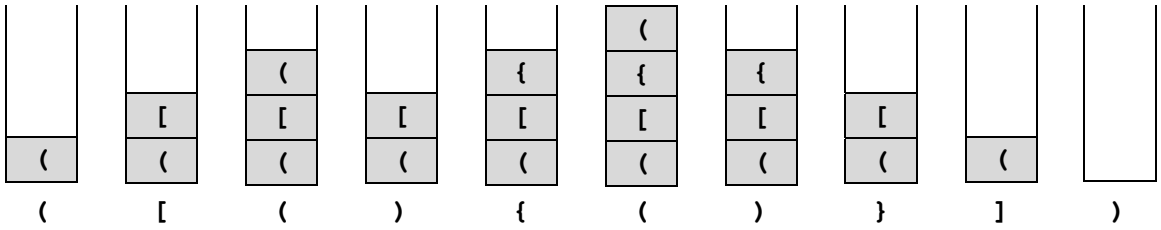
неправильные. В первых трёх есть непарные скобки, а в последних двух не соблюдается вложенность скобок.

Начнём с аналогичной задачи, в которой используется только один вид скобок. Её можно решить с помощью счётчика скобок. Сначала счётчик равен нулю. Строка просматривается слева направо, если очередной символ – открывающая скобка, то счётчик увеличивается на 1, если закрывающая – уменьшается на 1. В конце просмотра счётчик должен быть равен нулю (все скобки парные), кроме того, во время просмотра он не должен становиться отрицательным (должна соблюдаться вложенность скобок).

В исходной задаче (с тремя типами скобок) хочется завести три счётчика и работать с каждым отдельно. Однако, это решение неверное. Например, для выражения **({[)}]** условия «правильности» выполняются отдельно для каждого вида скобок, но не для выражения в целом.

Задачи, в которых важна вложенность объектов, удобно решать с помощью стека. Нас интересуют только открывающие и закрывающие скобки, на остальные символы можно не обращать внимания.

Строка просматривается слева направо. Если очередной символ – открывающая скобка, нужно втолкнуть её на вершину стека. Если это закрывающая скобка, то проверяем, что лежит на вершине стека: если там соответствующая открывающая скобка, то её нужно просто снять со стека. Если стек пуст или на вершине лежит открывающая скобка другого типа, выражение неверное и нужно закончить просмотр. В конце обработки правильной строки стек должен быть пуст. Кроме того, во время просмотра не должно быть ошибок. Работа такого алгоритма иллюстрируется на рисунке (для правильного выражения):



Введем структуру данных для описания стека:

```
typedef struct {
  char *data;
  int capacity;
  int size;
  } TStack;
```

Отличие этой структуры от стека из предыдущего примера только в том, что он содержит не целые числа, а символы (типа **char**). Поэтому приводить подпрограммы **Push** и **Pop** мы не будем, вы можете их переделать самостоятельно. Для удобства добавим только логическую функцию, которая возвращает значение **true** (истина), если стек пуст:

```
bool isEmpty ( TStack S )
{
  return S.size == 0;
}
```

Введём строковые константы **L** и **R**, которые содержат все виды открывающих и соответствующих закрывающих скобок.

```
const char L[] = "([{", // открывающие скобки
```

```
            R[] = ")]}"; // закрывающие скобки
```

Объявим переменные основной программы:

```
char str[80];     // рабочая строка
TStack S;         // стек
bool err;         // была ли ошибка?
int i;
char c, *p;
```

Переменная **str** – это исходная строка. Логическая переменная **err** будет сигнализировать об ошибке. Сначала ей присваивается значение **false** (ложь).

В основном цикле меняется целая переменная **i**, которая обозначает номер текущего символа, переменная-указатель **p** используется как вспомогательная:

```
for ( i=0; i<strlen(str); i++ )
  {
  p=strchr ( L, str[i] );          // 1
  if ( p!=NULL )
    Push ( &S, str[i] );           // 2
  p=strchr ( R, str[i] );          // 3
  if ( p!=NULL ) {                 // 4
    if ( isEmpty ( S ) )
      err=true;                    // 5
    else
      {
      c=Pop ( &S );                // 6
      if ( p-R!=strchr(L,c)-L )    // 7
        err=true;
      }
    if ( err ) break;              // 8
    }
  }
```

Сначала мы ищем символ **str[i]** в строке **L**, то есть среди открывающих скобок (строка 1). Если это действительно открывающая скобка, вталкиваем ее в стек (2).

Далее ищем символ среди закрывающих скобок (3). Если нашли (4), то в первую очередь проверяем, не пуст ли стек. Если стек пуст, выражение неверное и переменная **err** принимает истинное значение (5).

Если в стеке что-то есть, снимаем символ с вершины стека в символьную переменную **c** (6). В строке (7) сравнивается тип (номер) закрывающей скобки **p** и номер открывающей скобки, найденной на вершине стека. Если они не совпадают, выражение неправильное, и в переменную **err** записывается значение **true**.

Если при обработке текущего символа обнаружено, что выражение неверное (переменная **err** установлена в **true**), цикл завершается досрочно с помощью оператора **break** (8).

После окончания цикла нужно проверить содержимое стека: если он не пуст, в выражении есть незакрытые скобки, оно ошибочно:

```
if ( !isEmpty(S) ) err=true;
```

В конце программы остаётся вывести результат на экран:

```
if ( !err )
   printf ( "Скобки расставлены верно." );
else
   printf ( "Скобки расставлены неверно." );
```

Поскольку в этой задаче элементы стека – символы, решение можно значительно упростить, если использовать в качестве стека символьную строку. Попробуйте написать такой вариант программы самостоятельно.

Очереди, деки

Все мы знакомы с принципом очереди: первым пришёл – первым обслужен (англ. *FIFO = First In – First Out*). Соответствующая структура данных в информатике тоже называется очередью.

Очередь – это линейный список, для которого введены две операции:
- добавление нового элемента в конец очереди;
- удаление первого элемента из очереди.

Очередь – это не просто теоретическая модель. Операционные системы используют очереди для организации сообщения между программами: каждая программа имеет свою очередь сообщений. Контроллеры жестких дисков формируют очереди запросов ввода и вывода данных. В сетевых маршрутизаторах создается очередь из пакетов данных, ожидающих отправки.

Задача 7. *Рисунок задан в виде матрицы* **A**, *в которой элемент* **A[y][x]** *определяет цвет пикселя на пересечении строки* **y** *и столбца* **x**. *Перекрасить в цвет 2 одноцветную область, начиная с пикселя* (x_0, y_0). *На рисунке показан результат такой заливки для матрицы из 5 строк и 5 столбцов с начальной точкой* (1,0).

	0	1	2	3	4
0	0	1	0	1	1
1	1	1	1	2	2
2	0	1	0	2	2
3	3	3	1	2	2
4	0	1	1	0	0

(2,1) →

	0	1	2	3	4
0	0	2	0	1	1
1	2	2	2	2	2
2	0	2	0	2	2
3	3	3	1	2	2
4	0	1	1	0	0

Эта задача актуальна для графических программ. Один из возможных вариантов решения использует очередь, элементы которой – координаты пикселей (точек):

```
добавить в очередь точку (x0,y0)
запомнить цвет начальной точки
пока очередь не пуста
  {
  взять из очереди точку (x,y)
  если A[y][x] = цвету начальной точки то
   {
    A[y][x] = 2;
    добавить в очередь точку (x-1,y)
    добавить в очередь точку (x+1,y)
    добавить в очередь точку (x,y-1)
    добавить в очередь точку (x,y+1)
    }
  }
```

Конечно, в очередь добавляются только те точки, которые находятся в пределах рисунка (матрицы **A**). Заметим, что в этом алгоритме некоторые точки могут быть добавлены в очередь несколько раз (подумайте, когда это может случиться). Поэтому решение можно несколько улучшить, как-то помечая точки, уже добавленные в очередь, чтобы не добавлять их повторно (попробуйте сделать это самостоятельно).

Две координаты точки связаны между собой, поэтому в программе лучше объединить их в структуру **TPoint** (от англ. *point* – точка)

```
typedef struct {
  int x, y;
  } TPoint;
```

а очередь составить из таких структур:

```
typedef struct {
  TPoint *data;
  int capacity;
  int size;
  } TQueue;
```

Для удобства построим функцию **Point**, которая формирует структуру типа **TPoint** по заданным координатам:

```
TPoint Point ( int x, int y )
```

```
{
  TPoint P;
  P.x=x; P.y=y;
  return P;
}
```

Для работы с очередью, основанной на динамическом массиве, введем две подпрограммы:

- процедура **Put** добавляет новый элемент в конец очереди; если нужно, массив расширяется блоками по 10 элементов;
- функция **Get** возвращает первый элемент очереди и удаляет его из очереди (обработку ошибки «очередь пуста» вы можете сделать самостоятельно); все следующие элементы сдвигаются к началу массива.

```
void Put ( TQueue *pQ, TPoint P )
{
  pQ->size ++;
  if ( pQ->size>pQ->capacity ) {
    pQ->capacity+=10;
    pQ->data=
   (TPoint*) realloc ( pQ->data, sizeof(TPoint)*pQ->capacity  );
    }
  pQ->data[pQ->size-1]=P;
}
TPoint Get ( TQueue *pQ )
{
  TPoint P=pQ->data[0];
  int i;
  pQ->size--;
  for ( i=0; i<pQ->size; i++ )
    pQ->data[i]=pQ->data[i+1];
  return P;
}
```

Остается написать основную программу. Объявляем константы:

```
const int XMAX=5, YMAX=5,
          NEW_COLOR=2;
```

и переменные:

```
int A[YMAX][XMAX];
TQueue Q;
FILE *F;
int i, j, x0, y0, color;
TPoint pt;
```

Предположим, что матрица **A** заполнена. Задаём исходную точку, с которой начинается заливка, запоминаем её «старый» цвет и добавляем эту точку в очередь:

```
y0=0; x0=1;              // начать заливку отсюда
color=A[y0][x0];         // цвет начальной точки
Put ( &Q, Point(x0,y0) );
```

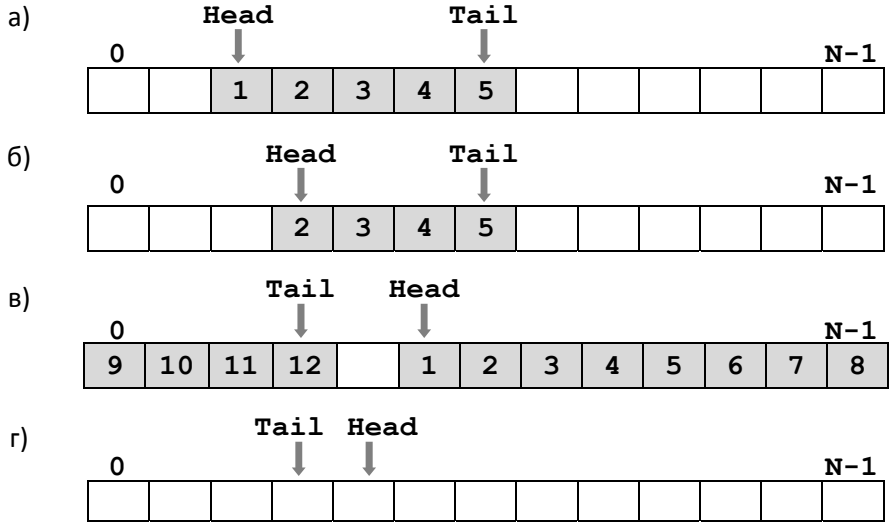
Основной цикл практически повторяет алгоритм на псевдокоде:

```
while ( !isEmpty(Q) )
  {
  pt=Get ( &Q );
  if ( A[pt.y][pt.x]==color )
    {
    A[pt.y][pt.x]=NEW_COLOR;
    if ( pt.x>0 ) Put ( &Q, Point(pt.x-1,pt.y) );
    if ( pt.y>0 ) Put ( &Q, Point(pt.x,pt.y-1) );
    if ( pt.x<XMAX-1 ) Put ( &Q, Point(pt.x+1,pt.y) );
    if ( pt.y<YMAX-1 ) Put ( &Q, Point(pt.x,pt.y+1) );
    }
```

```
}
```

Здесь функция **isEmpty** – такая же, как и для стека: возвращает **true**, если очередь пуста (поле **size** равно нулю).

В приведенном примере начало очереди всегда совпадает с первым элементом массива (имеющим индекс 0). При этом удаление элемента из очереди неэффективно, потому что требуется сдвинуть все оставшиеся элементы к началу массива. Существует и другой подход, при котором элементы очереди не передвигаются. Допустим, что мы знаем, что количество элементов в очереди всегда меньше **N**. Тогда можно выделить статический массив из **N** элементов и хранить в отдельных переменных номера первого элемента очереди («головы», англ. *head*) и последнего элемента («хвоста», англ. *tail*). На рисунке *а* показана очередь из 5 элементов. В этом случае удаление элемента из очереди сводится просто к увеличению переменной **Head** (рисунок *б*).



При добавление элемента в конец очереди переменная **Tail** увеличивается на 1. Если она перед этим указывала на последний элемент массива, то следующий элемент записывается в начало массива, а переменной **Tail** присваивается значение 0. Таким образом, массив оказывается замкнутым «в кольцо». На рисунке *в* показана полностью заполненная очередь, а на рисунке *г* – пустая очередь. Один элемент массива всегда остается незанятым[6], иначе невозможно будет различить состояния «очередь пуста» и «очередь заполнена».

Отметим, что приведенная здесь модель описывает работу кольцевого буфера клавиатуры, который может хранить до 15 двухбайтных слов.

В библиотеке STL есть класс **queue** (от англ. *queue* – очередь), который моделирует работу очереди. Как и для всех стандартных контейнеров, все нужные операции уже готовы:

- **push** – добавить элемент в конец очереди;
- **pop** – удалить первый элемент в очереди;
- **front** – получить первый элемент в очереди, не удаляя его;
- **empty** – логическая функция, которая возвращает истинное значение (**true**), если очередь пуста.

Очередь состоит из структур типа TPoint, поэтому объявить её нужно так:

```
#include <queue>
...
queue <TPoint> Q;
```

Приведём сразу основную часть программы, которая повторяет вариант, написанный ранее на языке С:

```
color = A[y0][x0];
Q.push ( Point(x0,y0) );
while ( !Q.empty() )
  {
```

[6] Возможен и другой вариант, когда длина очереди хранится в отдельной переменной.

```
  pt=Q.front(); Q.pop();
  if ( A[pt.y][pt.x]==color )
    {
    A[pt.y][pt.x]=NEW_COLOR;
    if ( pt.x>0 ) Q.push ( Point(pt.x-1,pt.y) );
    if ( pt.y>0 ) Q.push ( Point(pt.x,pt.y-1) );
    if ( pt.x<XMAX-1 ) Q.push ( Point(pt.x+1,pt.y) );
    if ( pt.y<YMAX-1 ) Q.push ( Point(pt.x,pt.y+1) );
    }
  }
```

Кроме того, очередь можно построить на основе связного списка (см. § 41). Детали такой реализации можно найти в литературе или в Интернете.

Существует еще одна линейная динамическая структура данных, которая называется *дек*.

Дек (от англ. *deque — double ended queue,* двухсторонняя очередь) – это линейный список, в котором можно добавлять и удалять элементы как с одного, так и с другого конца.

Из этого определения следует, что дек может работать и как стек, и как очередь. С помощью дека можно, например, моделировать колоду игральных карт.

В состав библиотеки STL входит тип **deque** для работы с деком, который строится на основе двусвязного списка. Это гарантирует быструю вставку и удаление элементов на концах дека.

? *Контрольные вопросы*

1. Что такое стек? Какие операции со стеком разрешены?
2. Как используется системный стек при выполнении программ?
3. Какие ошибки могут возникнуть при использовании стека?
4. В каких случаях можно использовать обычный массив для моделирования стека?
5. Как построить стек на основе динамического массива?
6. Почему при передаче объекта-стека в приведённые подпрограммы соответствующий параметр должен быть изменяемым?
7. Что такое очередь? Какие операции она допускает?
8. Приведите примеры задач, в которых можно использовать очередь.
9. Какие типы данных для работы со стеками, очередями и деками есть в библиотеке STL?

Задачи

1. Напишите программу, которая «переворачивает» массив, записанный в файл, с помощью стека. Размер массива неизвестен. Все операции со стеком вынесите в отдельный модуль.
2. Напишите программу, которая вычисляет значение арифметического выражения, записанного в постфиксной форме. Выражение вводится с клавиатуры в виде символьной строки.
3. Напишите программу, которая проверяет правильность скобочного выражения с четырьмя видами скобок: (), [], {} и <>. Все операции со стеком вынесите в отдельный модуль.
4. Напишите вариант предыдущей программы, в котором в качестве стека используется символьная строка.
5. Найдите в литературе или в Интернете алгоритм перевода арифметического выражения из инфиксной формы в постфиксную, и напишите программу, которая решает эту задачу.
6. Напишите программу, которая выполняет заливку одноцветной области заданным цветом. Матрица, содержащая цвета пикселей, вводится из файла. Затем с клавиатуры вводятся координаты точки заливки и цвет заливки. На экран нужно вывести матрицу, которая получилась после заливки. Все операции с очередью вынесите в отдельный модуль.

7. *Перепишите предыдущую программу – используйте статический массив для организации очереди. Считайте, что в очереди может быть не более 100 элементов. Предусмотрите обработку ошибки «очередь переполнена».
8. *Напишите решение задачи о заливке области, в котором точки, добавленные в очередь, как-то помечаются, чтобы не добавлять их повторно. В чём преимущества и недостатки такого алгоритма?

§ 43. Деревья

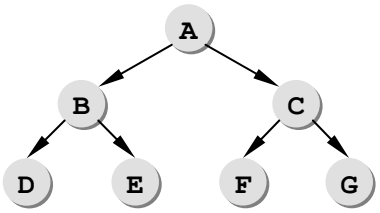
Что такое дерево?

Как вы знаете из учебника 10 класса, дерево – это структура данных, отражающая иерархию (отношения подчиненности, многоуровневые связи). Напомним некоторые основные понятия, связанные с деревьями.

Дерево состоит из узлов и связей между ними (они называются дугами). Самый первый узел, расположенный на верхнем уровне (в него не входит ни одна стрелка-дуга) – это *корень дерева.* Конечные узлы, из которых не выходит ни одна дуга, называются *листьями*. Все остальные узлы, кроме корня и листьев – это промежуточные узлы.

Из двух связанных узлов тот, который находится на более высоком уровне, называется «*родителем*», а другой – «*сыном*». Корень – это единственный узел, у которого нет «родителя»; у листьев нет «сыновей».

Используются также понятия «предок» и «потомок». «Потомок» какого-то узла – это узел, в который можно перейти по стрелкам от узла-предка. Соответственно, «предок» какого-то узла – это узел, из которого можно перейти по стрелкам в данный узел.



В дереве на рисунке справа родитель узла E – это узел B, а предки узла E – это узлы A и B, для которых узел E – потомок. Потомками узла A (корня дерева) являются все остальные узлы.

Высота дерева – это наибольшее расстояние (количество рёбер) от корня до листа. Высота дерева, приведённого на рисунке, равна 2.

Формально **дерево** можно определить следующим образом:
1) пустая структура – это дерево;
2) дерево – это корень и несколько связанных с ним отдельных (не связанных между собой) деревьев.

Здесь множество объектов (деревьев) определяется через само это множество на основе простого базового случая (пустого дерева). Такой приём называется *рекурсией* (см. главу 8 учебника 10 класса). Поэтому можно ожидать, что при работе с деревьями будут полезны рекурсивные алгоритмы.

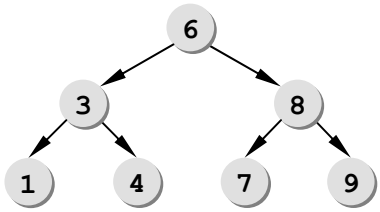
Чаще всего в информатике используются *двоичные* (или *бинарные*) деревья, то есть такие, в которых каждый узел имеет не более двух сыновей. Их также можно определить рекурсивно.

Двоичное дерево:
1) пустая структура – это двоичное дерево;
2) двоичное дерево – это корень и два связанных с ним отдельных двоичных дерева («левое» и «правое» поддеревья).

Деревья широко применяются в следующих задачах:
- поиск в большом массиве данных;
- сортировка данных;
- вычисление арифметических выражений;
- оптимальное кодирование данных (метод сжатия Хаффмана).

Деревья поиска

Известно, что для того, чтобы найти заданный элемент в неупорядоченном массиве из **N** элементов, может понадобиться **N** сравнений. Теперь предположим, что элементы массива организованы в виде специальным образом построенного дерева, например:



Значения, связанные с каждым из узлов дерева, по которым выполняется поиск, называются *ключами* этих узлов (кроме ключа узел может содержать множество других данных). Перечислим важные свойства показанного дерева:

- слева от каждого узла находятся узлы с меньшим ключом;
- справа от каждого узла находятся узлы, ключ которых больше или равен ключу данного узла.

Дерево, обладающее такими свойствами, называется *двоичным деревом поиска*.

Например, нужно найти узел, ключ которого равен 4. Начинаем поиск по дереву с корня. Ключ корня – 6 (больше заданного), поэтому дальше нужно искать только в левом поддереве, и т.д.

Скорость поиска наибольшая в том случае, если дерево *сбалансировано*, то есть для каждой его вершины высота левого и правого поддеревьев различается не более чем на единицу. Если при линейном поиске в массиве за одно сравнение отсекается 1 элемент, здесь – сразу примерно половина оставшихся (если дерево *сбалансировано*, то есть для каждой его вершины высота левого и правого поддеревьев различается не более чем на 1). Количество операций сравнения в этом случае пропорционально $\log_2 N$, то есть алгоритм имеет асимптотическую сложность $O(\log N)$. Конечно, нужно учитывать, что предварительно дерево должно быть построено. Поэтому такой алгоритм выгодно применять в тех случаях, когда данные меняются редко, а поиск выполняется часто (например, в базах данных).

Обход дерева

Обойти дерево – это значит «посетить» все узлы по одному разу. Если перечислить узлы в порядке их посещения, мы представим данные в виде списка.

Существуют несколько способов обхода двоичного дерева:

- КЛП = «корень – левый – правый» (обход в прямом порядке):

```
посетить корень
обойти левое поддерево
обойти правое поддерево
```

- ЛКП = «левый – корень – правый» (симметричный обход):

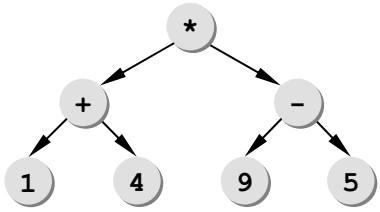
```
обойти левое поддерево
посетить корень
обойти правое поддерево
```

- ЛПК = «левый – правый – корень» (обход в обратном порядке):

```
обойти левое поддерево
обойти правое поддерево
посетить корень
```

Как видим, это рекурсивные алгоритмы. Они должны заканчиваться без повторного вызова, когда текущий корень – пустое дерево.

Рассмотрим дерево, которое может быть составлено для вычисления арифметического выражения `(1+4)*(9-5)`:



Выражение вычисляется по такому дереву снизу вверх, то есть корень дерева – это последняя выполняемая операция.

Различные типы обхода дают последовательность узлов:

КЛП: *** + 1 4 – 9 5**

ЛКП: **1 + 4 * 9 – 5**

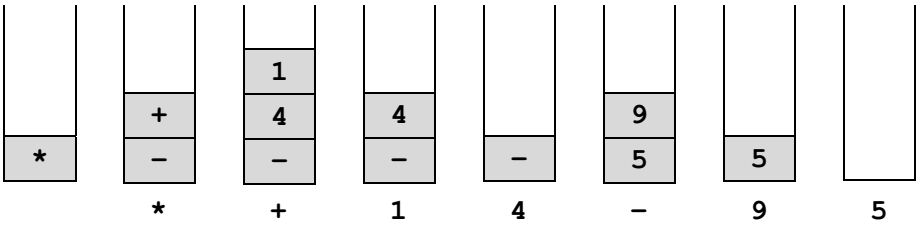
ЛПК: **1 4 + 9 5 – ***

В первом случае мы получили префиксную форму записи арифметического выражения, во втором – привычную нам инфиксную форму (только без скобок), а в третьем – постфиксную форму. Напомним, что в префиксной и в постфиксной формах скобки не нужны.

Обход КЛП называется «обходом в глубину», потому что сначала мы идём вглубь дерева по левым поддеревьям, пока не дойдём до листа. Такой обход можно выполнить с помощью стека следующим образом:

```
записать в стек корень дерева
пока стек не пуст
  {
  выбрать узел V с вершины стека
  посетить узел V
  если у узла V есть правый сын то
    добавить в стек правого сына V
  если у узла V есть левый сын то
    добавить в стек левого сына V
  }
```

На рисунке 6.13 показано изменение состояния стека при таком обходе дерева, изображенного на рис. 6.12. Под стеком записана метка узла, который посещается (например, данные из этого узла выводятся на экран).



Существует еще один способ обхода, который называют «*обходом в ширину*». Сначала посещают корень дерева, затем – всех его «сыновей», затем – «сыновей сыновей» («внуков») и т.д., постепенно спускаясь на один уровень вниз. Обход в ширину для приведённого выше дерева даст такую последовательность посещения узлов:

обход в ширину: *** + – 1 4 9 5**

Для того, чтобы выполнить такой обход, применяют очередь. В очередь записывают узлы, которые необходимо посетить. На псевдокоде обход в ширину можно записать так:

```
записать в очередь корень дерева
пока очередь не пуста
  {
  выбрать первый узел V из очереди
  посетить узел V
  если у узла V есть левый сын то
    добавить в очередь левого сына V
  если у узла V есть правый сын то
    добавить в очередь правого сына V
  }
```

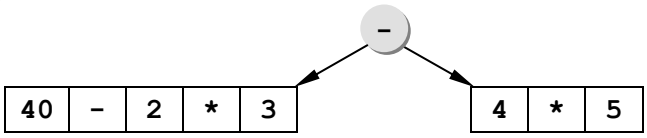
Вычисление арифметических выражений

Один из способов вычисления арифметических выражений основан на использовании дерева. Сначала выражение, записанное в линейном виде (в одну строку), нужно «разобрать» и построить соответствующее ему дерево. Затем в результате прохода по этому дереву от листьев к корню вычисляется результат.

Для простоты будем рассматривать только арифметические выражения, содержащие числа и знаки четырёх арифметических действий: **+-*/**. Построим дерево для выражения

40 – 2 * 3 – 4 * 5

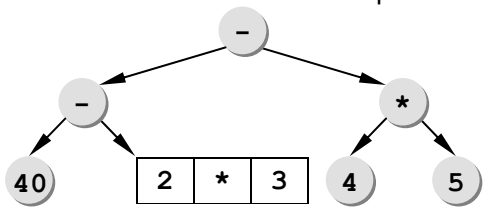
Так как корень дерева – это последняя операция, нужно сначала найти эту последнюю операцию, просматривая выражение слева направо. Здесь последнее действие – это второе вычитание, оно оказывается в корне дерева.



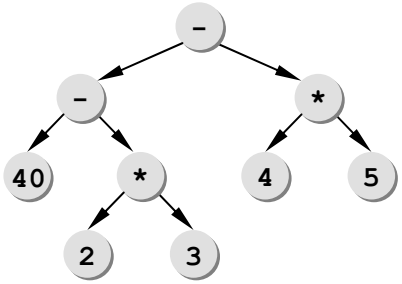
Как выполнить этот поиск в программе? Известно, что операции выполняются в порядке *приоритета* (старшинства): сначала операции с более высоким приоритетом (слева направо), потом – с более низким (также слева направо). Отсюда следует важный вывод.

> В корень дерева нужно поместить последнюю из операций с наименьшим приоритетом.

Теперь нужно построить таким же способом левое и правое поддеревья:



Левое поддерево требует еще одного шага:



Эта процедура рекурсивная, её можно записать в виде псевдокода:

```
найти последнюю выполняемую операцию
если операций нет то
  {
  создать узел-лист
  выход
  }
поместить найденную операцию в корень дерева
построить левое поддерево
построить правое поддерево
```

Рекурсия заканчивается, когда в оставшейся части строки нет ни одной операции, значит, там находится число (это лист дерева).

Теперь вычислим выражение по дереву. Если в корне находится знак операции, её нужно применить к результатам вычисления поддеревьев:

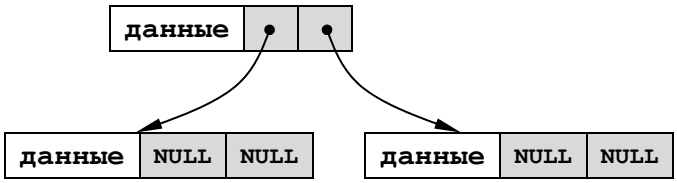
```
n1 = значение левого поддерева
n2 = значение правого поддерева
результат = операция(n1, n2)
```

Снова получился рекурсивный алгоритм.

Возможен особый случай (на нём заканчивается рекурсия), когда корень дерева содержит число (то есть это лист). Это число и будет результатом вычисления выражения.

Использование связанных структур

Поскольку двоичное дерево – это нелинейная структура данных, использовать динамический массив для размещения элементов не очень удобно (хотя возможно). Вместо этого будем использовать связанные узлы. Каждый такой узел – это структура, содержащая три области: область данных, ссылка на левое поддерево (указатель) и ссылка на правое поддерево (второй указатель). У листьев нет «сыновей», в этом случае в указатели будем записывать значение **NULL** (нулевой указатель). Дерево, состоящее из трёх таких узлов, показано на рисунке:



В данном случае область данных узла будет содержать одно поле – символьную строку, в которую записывается знак операции или число в символьном виде.

Введём два новых типа: **TNode** – узел дерева, и **PNode** – указатель (ссылку) на такой узел[7]:

```
typedef struct TNode *PNode;
typedef struct TNode {
    char data[20];
    PNode left;
    PNode right;
  } TNode;
```

Самый важный момент – выделение памяти под новую структуру. Предположим, что **p** – это переменная-указатель типа **PNode**. Для того, чтобы выделить память под новую структуру и записать адрес выделенного блока в **p**, в языке С используется уже знакомая нам функция **calloc**, а в С++ – оператор **new**:

```
p = (PNode) calloc ( 1, sizeof(TNode) );
```

```
p = new TNode;
```

Для освобождения памяти служит процедура **free** (англ. *free* – освободить) или, в С++, оператор **delete**:

```
free ( p );
```

```
delete p;
```

В основной программе объявим одну переменную типа **PNode** – это будет ссылка на корень дерева:

```
PNode T;
```

Вычисление выражения сводится к двум вызовам функций:

```
T = MakeTree ( s );
printf ( "Результат: %d", Calc(T) );
```

Здесь предполагается, что арифметическое выражение записано в символьной строке **s**, функция **MakeTree** строит в памяти дерево по этой строке, а функция **Calc** – вычисляет значение выражения по готовому дереву.

При построении дерева нужно выделить в памяти новый узел и искать последнюю выполняемую операцию – это будет делать функция **LastOp**. Она вернет 0, если ни одной операции не обнаружено, в этом случае создается лист – узел без потомков. Если операция найдена, её обозначение записывается в поле **data**, а в указатели – адреса поддеревьев, которые строятся рекурсивно для левой и правой частей выражения:

```
PNode MakeTree ( char s[] )
{
  int k;
  PNode Tree;
```

[7] Заметим, что при объявлении типа указателей **PNode** идет ссылка «вперёд», на тип **TNode**, который объявляется ниже.

```
  char sLeft[80] = "";
  Tree = (PNode) calloc ( 1, sizeof(TNode) );
  k = LastOp ( s );
  if ( k == -1 )
    {
    strcpy ( Tree->data, s );
    Tree->left = NULL;
    Tree->right = NULL;
    }
  else
    {
    Tree->data[0] = s[k];
    Tree->data[1] = '\0';
    strncpy ( sLeft, s, k );
    Tree->left = MakeTree ( sLeft );
    Tree->right = MakeTree ( &s[k+1] );
    }
  return Tree;
}
```

Функция **Calc** тоже будет рекурсивной:

```
int Calc ( PNode Tree )
{
  int n1, n2, res;
  if ( Tree->left == NULL )
    res = atoi ( Tree->data );
  else
    {
    n1 = Calc ( Tree->left );
    n2 = Calc ( Tree->right );
    switch ( Tree->data[0] )
      {
      case '+': res = n1 + n2; break;
      case '-': res = n1 - n2; break;
      case '*': res = n1 * n2; break;
      case '/': res = n1 / n2; break;
      default: res = 99999;
      }
    }
  return res;
}
```

Если ссылка, переданная функции, указывает на лист (нет левого поддерева), то значение выражения – это результат преобразования числа из символьной формы в числовую (с помощью функции **atoi**). В противном случае вычисляются значения для левого и правого поддеревьев, и к ним применяется операция, указанная в корне дерева. В случае ошибки (неизвестной операции) функция возвращает значение **99999** – большое целое число.

Осталось написать функцию **LastOp**. Нужно найти в символьной строке последнюю операцию с минимальным приоритетом. Для этого составим функцию, возвращающую приоритет операции (переданного ей символа):

```
int Priority ( char op )
{
  switch ( op )
    {
    case '+':
    case '-': return 1;
    case '*':
    case '/': return 2;
```

```
    }
  return 100;
}
```

Сложение и вычитание имеют приоритет 1, умножение и деление – приоритет 2, а все остальные символы (не операции) – приоритет 100 (условное значение).

Функция **LastOp** может выглядеть так:

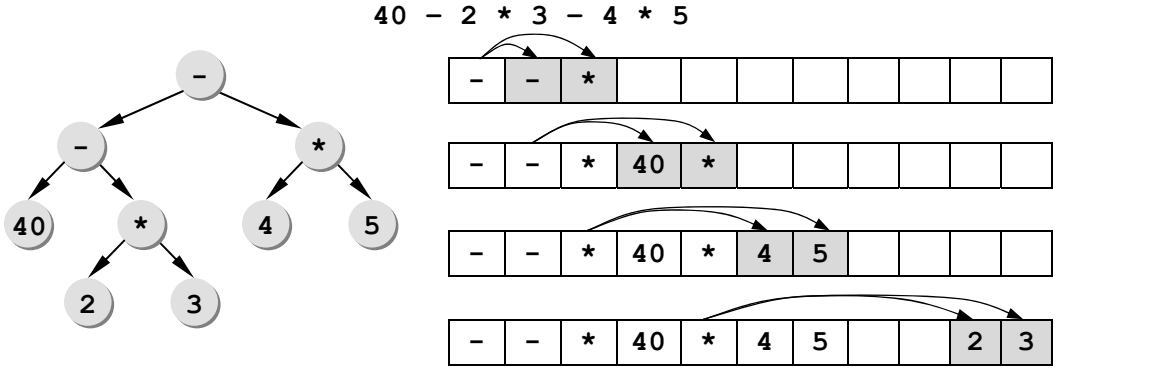
```
int LastOp ( char s[] )
{
  int i, minPrt, res;
  minPrt = 50;
  res = -1;
  for ( i = 0; i < strlen(s); i++ )
    if ( Priority(s[i]) <= minPrt )
      {
      minPrt = Priority(s[i]);
      res = i;
      }
  return res;
}
```

Обратите внимание, что в условном операторе указано нестрогое неравенство, чтобы найти именно *последнюю* операцию с наименьшим приоритетом. Начальное значение переменной **minPrt** можно выбрать любое между наибольшим приоритетом операций (2) и условным кодом не-операции (100). Тогда если найдена любая операция, условный оператор срабатывает, а если в строке нет операций, условие всегда ложно и в переменной **LastOp** остается начальное значение 0.

Аналогичную программу на языке C++, которая использует символьные строки типа **string** вместо символьных массивов, вы можете написать самостоятельно.

Хранение двоичного дерева в массиве

Двоичные деревья удобно хранить в (динамическом) массиве, почти так же, как и списки. Вопрос о том, как сохранить структуру (взаимосвязь узлов) решается достаточно просто. Если нумерация элементов массива **A** начинается с 0, то «сыновья» элемента **A[i]** – это **A[2*i+1]** и **A[2*i+2]**. На рисунке показан порядок расположения элементов в массиве для дерева, соответствующего выражению



Алгоритм вычисления выражения остается прежним, изменяется только метод хранения данных. Обратите внимание, что некоторые элементы остались пустые, это значит, что их «родитель» – лист дерева. Этот способ хорош для хранения сбалансированных (или почти сбалансированных) деревьев, иначе в массиве будет много пустых элементов, которые зря расходуют память.

? *Контрольные вопросы*

1. Дайте определение понятий «дерево», «корень», «лист», «родитель», «сын», потомок», «предок», «высота дерева».
2. Где используются структуры типа «дерево» в информатике и в других областях?

3. Объясните рекурсивное определение дерева.
4. Можно ли считать, что линейный список – это частный случай дерева?
5. Какими свойствами обладает дерево поиска?
6. Подумайте, как можно построить дерево поиска из массива данных?
7. Что такое сбалансированное дерево?
8. Какие преимущества имеет поиск с помощью дерева?
9. Что такое «обход» дерева?
10. Какие способы обхода дерева вы знаете? Придумайте другие способы обхода.
11. Как строится дерево для вычисления арифметического выражения?
12. Как представляется дерево с помощью связанных структур?
13. Как указать, что узел дерева не имеет левого (правого) «сына»?
14. Как выделяется память под новый узел?
15. Как вы думаете, почему рекурсивные алгоритмы работы с деревьями получаются проще, чем нерекурсивные?
16. Как хранить двоичное дерево в массиве? Можно ли использовать такой приём для хранения деревьев, в которых узлы могут иметь больше двух «сыновей»?

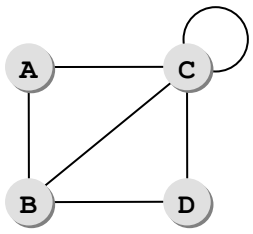
Задачи

1. Напишите программу, которая вводит и вычисляет арифметическое выражение без скобок. Все операции с деревом вынесите в отдельный модуль.
2. Добавьте в предыдущую программу процедуры обхода построенного дерева так, чтобы получить префиксную и постфиксную запись введенного выражения.
3. *Добавьте в предыдущую программу процедуру обхода дерева в ширину.
4. *Усовершенствуйте программу (см. задачу 1), чтобы она могла вычислять выражения со скобками.
5. *Включите в вашу программу обработку некоторых ошибок (например, два знака операций подряд). Поработайте в парах: обменяйтесь программами с соседом и попробуйте найти выражение, при котором его программа завершится аварийно (не выдаст сообщение об ошибке).
6. *Напишите программу для вычисления арифметического выражения на языке C++, используя тип данных **`string`** вместо массивов символов.
7. *Напишите программу вычисления арифметического выражения, которая хранит дерево в виде массива. Все операции с деревом вынесите в отдельный модуль.

§ 44. Графы

Что такое граф?

Как вы знаете из курса 10 класса, граф – это набор вершин (узлов) и связей между ними (рёбер). Информацию о вершинах и рёбрах графа обычно хранят в виде таблицы специального вида – *матрицы смежности*:

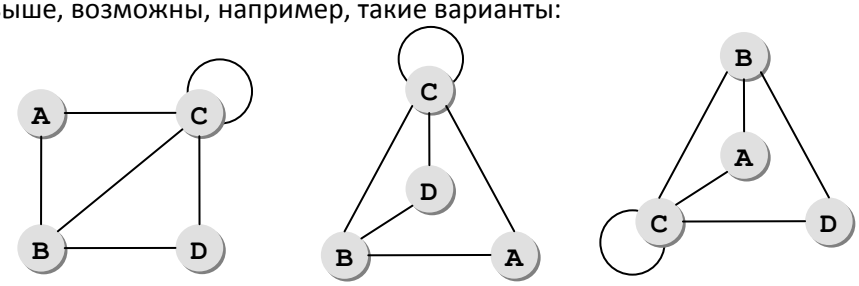


	A	B	C	D
A	0	1	1	0
B	1	0	1	1
C	1	1	1	1
D	0	1	1	0

Единица на пересечении строки A и столбца B означает, что между вершинами A и B есть связь. Ноль указывает на то, что связи нет. Матрица смежности симметрична относительно главной диагонали (серые клетки в таблице). Единица на главной диагонали обозначает *петлю* – ребро, которое начинается и заканчивается в одной и той же вершине (в данном случае – в вершине C).

Строго говоря, граф – это математический объект, а не рисунок. Конечно, его можно нарисовать на плоскости (например, как на рис. 1.16, б), но матрица смежности не даёт никакой инфор-

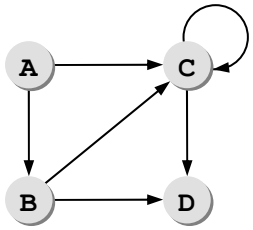
мации о том, как именно следует располагать вершины друг относительно друга. Для таблицы, приведенной выше, возможны, например, такие варианты:



В рассмотренном примере все узлы связаны, то есть, между любой парой вершин существует *путь* – последовательность рёбер, по которым можно перейти из одной вершины в другую. Такой граф называется *связным*.

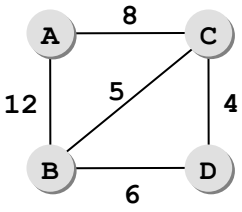
Вспоминая материал предыдущего пункта, можно сделать вывод, что дерево – это частный случай связного графа, в котором нет замкнутых путей – *циклов*.

Если для каждого ребра указано направление, граф называют *ориентированным* (или *орграфом*). Рёбра орграфа называют *дугами*. Его матрица смежности не всегда симметричная. Единица, стоящая на пересечении строки A и столбца B говорит о том, что существует дуга из вершины A в вершину B:



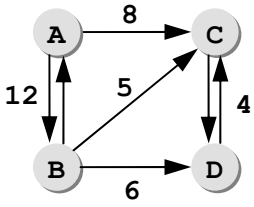
	A	B	C	D
A	0	1	1	0
B	0	0	1	1
C	0	0	1	1
D	0	0	0	0

Часто с каждым ребром связывают некоторое число – *вес ребра*. Это может быть, например, расстояние между городами или стоимость проезда. Такой граф называется *взвешенным*. Информация о взвешенном графе хранится в виде *весовой матрицы,* содержащей веса рёбер:



	A	B	C	D
A		12	8	
B	12		5	6
C	8	5		4
D		6	4	

У взвешенного орграфа весовая матрица может быть несимметрична относительно главной диагонали:

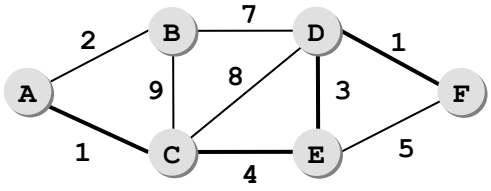


	A	B	C	D
A		12	8	
B	12		5	6
C				4
D			4	

Если связи между двумя вершинами нет, на бумаге можно оставить ячейку таблицы пустой, а при хранении в памяти компьютера записывать в нее условный код, например, 0, –1 или очень большое число (∞), в зависимости от задачи.

«Жадные» алгоритмы

Задача 8. *Известна схема дорог между несколькими городами. Числа на схеме (рис. 6.26) обозначают расстояния (дороги не прямые, поэтому неравенство треугольника может нарушаться):*

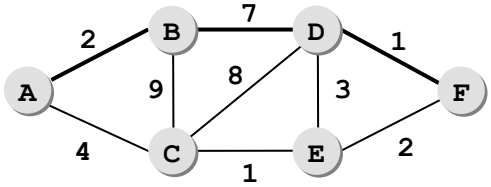


Нужно найти кратчайший маршрут из города **A** *в город* **F**.

Первая мысль, которая приходит в голову – на каждом шаге выбирать кратчайший маршрут до ближайшего города, в котором мы еще не были. Для заданной схемы на первом этапе едем в город **C** (длина 1), далее – в **E** (длина 4), затем в **D** (длина 3) и наконец в **F** (длина 1). Общая длина маршрута равна 9.

Алгоритм, который мы применили, называется «жадным». Он состоит в том, чтобы на каждом шаге многоходового процесса выбирать наилучший в данный момент вариант, не думая о том, что впоследствии этот выбор может привести к худшему решению.

Для данной схемы жадный алгоритм на самом деле дает оптимальное решение, но так будет далеко не всегда. Например, для той же задачи с другой схемой:



жадный алгоритм даст маршрут A-B-D-F длиной 10, хотя существует более короткий маршрут A-C-E-F длиной 7.

Жадный алгоритм не всегда позволяет получить оптимальное решение.

Однако есть задачи, в которых жадный алгоритм всегда приводит к правильному решению. Одна из таких задач (её называют *задачей Прима-Крускала* в честь Р. Прима и Д. Крускала, которые независимо предложили её в середине XX века) формулируется так:

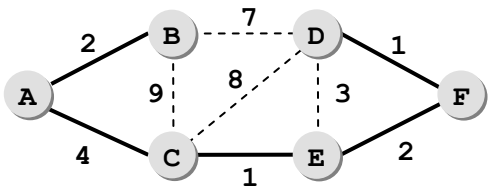
Задача 9. *В стране Лимонии есть* **N** *городов, которые нужно соединить линиями связи. Между какими городами нужно проложить линии связи, чтобы все города были связаны в одну систему и общая длина линий связи была наименьшей?*

В теории графов эта задача называется задачей построения *минимального остовного дерева* (то есть дерева, связывающего все вершины). Остовное дерево для связного графа с **N** вершинами имеем **N−1** ребро.

Рассмотрим жадный алгоритм решения этой задачи, предложенный Крускалом:

1) начальное дерево – пустое;
2) на каждом шаге к будущему дереву добавляется ребро минимального веса, которое ещё не было выбрано и не приводит к появлению цикла.

На рисунке показано минимальное остовное дерево для одного из рассмотренных выше графов (сплошные жирные линии):



Здесь возможна такая последовательность добавления рёбер: CE, DF, AB, EF, AC. Обратите внимание, что после добавления ребра EF следующее «свободное» ребро минимального веса – это DE (длина 3), но оно образует цикл с рёбрами DF и EF, и поэтому не было включено в дерево.

При программировании этого алгоритма сразу возникает вопрос: как определить, что ребро еще не включено в дерево и не образует цикла в нём? Существует очень красивое решение этой проблемы, основанное на раскраске вершин.

Сначала все вершины раскрашиваются в разные цвета, то есть все рёбра из графа удаляются. Таким образом, мы получаем множество элементарных деревьев (так называемый *лес*), каждое из которых состоит из одной вершины. Затем последовательно соединяем отдельные деревья, каждый раз выбирая ребро минимальной длины, соединяющее разные деревья (выкрашенные в разные цвета). Объединённое дерево перекрашивается в один цвет, совпадающий с цветом одного из вошедших в него поддеревьев. В конце концов все вершины оказываются выкрашены в один цвет, то есть все они принадлежат одному остовному дереву. Можно доказать, что это дерево будет иметь минимальный вес, если на каждом шаге выбирать подходящее ребро минимальной длины.

В программе сначала присвоим всем вершинам разные числовые коды («цвета»):

```
for ( i=0; i<N; i++ ) col[i]=i;
```

Здесь **N** – количество вершин, а **col** – целочисленный массив с индексами от 0 до **N-1**.

Затем в цикле **N-1** раз (именно столько рёбер нужно включить в дерево) выполняем следующие операции:

1) ищем ребро минимальной длины среди всех рёбер, *концы которых окрашены в разные цвета*;
2) найденное ребро **(iMin,jMin)** добавляется в список выбранных, и все вершины, имеющие цвет **col[jMin]**, перекрашиваются в цвет **col[iMin]**.

Приведем полностью основной цикл программы:

```
for ( k=0; k<N-1; k++ )
  {
    // поиск ребра с минимальным весом
  minDist=99999;                              // (*)
  for ( i=0; i<N; i++ )
    for ( j=0; j<N; j++ )
      if ( col[i]!=col[j] && W[i][j]<minDist )
        {
        iMin=i; jMin=j; minDist=W[i][j];
        }
    // добавление ребра в список выбранных
  ostov[k][0]=iMin;
  ostov[k][1]=jMin;
    // перекрашивание вершин
  c=col[jMin];
  for ( i=0; i<N; i++ )
    if ( col[i]==c )
      col[i]=col[iMin];
  }
```

Здесь **W** – целочисленная матрица размера **N** на **N** (индексы строк и столбцов начинаются с 0); **ostov** – целочисленный массив из **N**-1 строк и двух столбцов для хранения выбранных рёбер (для каждого ребра хранятся две вершины, которые оно соединяет). Если связи между вершинами **i** и **j** нет, в элементе **W[i][j]** матрицы будем хранить «бесконечность» – число, намного большее, чем длина любого ребра. При этом начальное значение переменной **minDist** в строке (*) должно быть ещё больше.

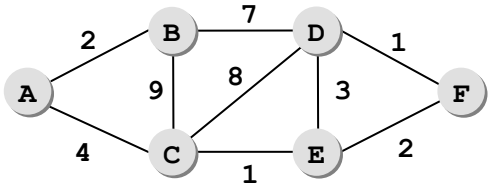
После окончания цикла остается вывести результат – рёбра из массива **ostov**:

```
for ( i=0; i<N-1; i++ )
  printf ( "(%d,%d)\n", ostov[i][0], ostov[i][1] );
```

Кратчайшие маршруты

На примере задачи выбора кратчайшего маршрута (см. задачу 8 выше) мы увидели, что в ней жадный алгоритм не всегда дает правильное решение. В 1960 году Э. Дейкстра предложил алгоритм, позволяющий найти все кратчайшие расстояния от одной вершины графа до всех остальных и соответствующие им маршруты. Предполагается, что длины всех рёбер (расстояния между вершинами) положительные.

Рассмотрим уже знакомую схему, в которой не сработал жадный алгоритм:



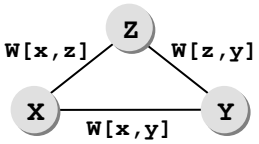
Алгоритм Дейкстры использует дополнительные массива: в одном (назовем его **R**) хранятся кратчайшие (на данный момент) расстояния от исходной вершины до каждой из вершин графа, а во втором (массив **P**) – вершина, из которой нужно приехать в данную вершину.

Сначала записываем в массив **R** длины рёбер от исходной вершины **A** до всех вершин, а в соответствующие элементы массива **P** – вершину **A**:

	A	B	C	D	E	F
R	0	2	4	∞	∞	∞
P	×	A	A			

Знак ∞, обозначает, что прямого пути нет из вершины **A** в данную вершину нет (в программе вместо ∞ можно использовать очень большое число). Таким образом, вершина **A** уже рассмотрена и выделена серым фоном. В первый элемент массива **P** записан символ **×**, обозначающий начальную точку маршрута (в программе можно использовать несуществующий номер вершины, например, **–1**).

Из оставшихся вершин находим вершину с минимальным значением в массиве **R**: это вершина **B**. Теперь проверяем пути, проходящие через эту вершину: не позволят ли они сократить маршрут к другим, которые мы ещё не посещали. Идея состоит в следующем: если сумма весов **W[x,z]+W[z,y]** меньше, чем вес **W[x,y]**, то из вершины **X** лучше ехать в вершину **Y** не напрямую, а через вершину **Z**:



Проверяем наш граф: ехать из A в C через B невыгодно (получается путь длиной 11 вместо 4), а вот в вершину D можно проехать (путь длиной 4), поэтому запоминаем это значение вместо ∞ в массиве **R**, и записываем вершину B на соответствующее место в массив **P** («в D приезжаем из B»):

	A	B	C	D	E	F
R	0	2	4	9	∞	∞
P	×	A	A	B		

Вершины **E** и **F** по-прежнему недоступны.

Следующей рассматриваем вершину **C** (для нее значение в массиве **R** минимально). Оказывается, что через неё можно добраться до **E** (длина пути 5):

	A	B	C	D	E	F
R	0	2	4	9	5	∞
P	×	A	A	B	C	

Затем посещаем вершину **E**, которая позволяет достигнуть вершины **F** и улучшить минимальную длину пути до вершины **D**:

	A	B	C	D	E	F
R	0	2	4	8	5	7
P	×	A	A	E	C	E

После рассмотрения вершин **F** и **D** таблица не меняется. Итак, мы получили, что кратчайший маршрут из **A** в **F** имеет длину 7, причем он приходит в вершину **F** из **E**. Как же получить весь мар-

шрут? Нужно просто посмотреть в массиве **P**, откуда лучше всего ехать в **E** – выясняется, что из вершины **C**, а в вершину **C** – напрямую из начальной точки **A**:

	A	B	C	D	E	F
R	0	2	4	8	5	7
P	×	A	A	E	C	E

Поэтому кратчайший маршрут **A-C-E-F**. Обратите внимание, что этот маршрут «раскручивается» в обратную сторону, от конечной вершины к начальной. Заметим, что полученная таблица содержит *все* кратчайшие маршруты из вершины **A** во все остальные, а не только из **A** в **F**.

Алгоритм Дейкстры можно рассматривать как своеобразный «жадный» алгоритм: действительно, на каждом шаге из всех невыбранных вершин выбирается вершина **X**, длина пути до которой от вершины **A** минимальна. Однако можно доказать, что это расстояние – действительно минимальная длина пути от **A** до **X**. Предположим, что для всех предыдущих выбранных вершин это свойство справедливо. При этом **X** – это ближайшая не выбранная вершина, которую можно достичь из начальной точки, проезжая только через выбранные вершины. Все остальные пути в **X**, проходящие через ещё не выбранные вершины, будут длиннее, поскольку все рёбра имеют положительную длину. Таким образом, найденная длина пути из **A** в **X** – минимальная. После завершения алгоритма, когда все вершины выбраны, в массиве **R** находятся длины кратчайших маршрутов.

В программе объявим константу

```
const int N = 6;
```

и переменные:

```
int W[N,N];
bool active[N];
int R[N], P[N];
int i, j, min, kMin;
```

Массив **W** – это весовая матрица, её удобно вводить из файла. Логический массив **active** хранит состояние вершин (просмотрена или не просмотрена): если значение **active[i]** истинно, то вершина активна (ещё не просматривалась).

В начале программы присваиваем начальные значения (объяснение см. выше), сразу помечаем, что вершина 0 просмотрена (не активна), с нее начинается маршрут.

```
for ( i = 0; i < N; i ++ )
  {
  active[i] = true;
  R[i] = W[0][i];
  P[i] = 0;
  }
active[0] = false;
P[0] = -1;
```

В основном цикле, который выполняется **N-1** раз (так, чтобы все вершины были просмотрены) среди активных вершин ищем вершину с минимальным соответствующим значением в массиве R и проверяем, не лучше ли ехать через неё:

```
for ( i = 0; i < N-1; i++ )
  {
    // поиск новой рабочей вершины R[j] -> min
  minDist = 99999;
  for ( j = 0; j < N; j ++ )
    if ( active[j] && R[j] < minDist)
      {
      minDist = R[j];
      kMin = j;
      }
  active[kMin] = false;
    // проверка маршрутов через вершину kMin
  for ( j = 0; j < N; j ++ )
```

```
      if ( R[kMin]+W[kMin][j] < R[j] )
        {
        R[j] = R[kMin] + W[kMin][j];
        P[j] = kMin;
        }
    }
```

В конце программы выводим оптимальный маршрут (здесь – до вершины с номером N) в обратном порядке следования вершин:

```
i = N-1;
while ( i != -1 )       // для начальной вершины P[i]=-1
  {
  printf ( "%d ", i );
  i = P[i];              // переход к следующей вершине
  }
```

Алгоритм Дейкстры, как мы видели, находит кратчайшие пути *из одной заданной вершины* во все остальные. Найти все кратчайшие пути (из любой вершины в любую другую) можно с помощью *алгоритма Флойда-Уоршелла*, основанного на той же самой идее сокращения маршрута (иногда бывает короче ехать через промежуточные вершины, чем напрямую):

```
for ( k = 0; k < N; k++ )
  for ( i = 0; i < N; i++ )
    for ( j = 0; j < N; j++ )
      if ( W[i][k]+W[k][j] < W[i][j] )
         W[i][j] = W[i][k] + W[k][j];
```

В результате исходная весовая матрица графа **W** размером **N** на **N** превращается в матрицу, хранящую длины оптимальных маршрутов. Для того, чтобы найти сами маршруты, нужно использовать еще одну дополнительную матрицу, которая выполняет ту же роль, что и массив **P** в алгоритме Дейкстры.

Некоторые задачи

С графами связаны некоторые классические задачи. Самая известная из них – задача коммивояжера (бродячего торговца).

Задача 3. *Бродячий торговец должен посетить N городов по одному разу и вернуться в город, откуда он начал путешествие. Известны расстояния между городами (или стоимость переезда из одного города в другой). В каком порядке нужно посещать города, чтобы суммарная длина пути (или стоимость) оказалась наименьшей?*

Эта задача оказалась одной из самых сложных задач оптимизации. По сей день известно только одно надёжное решение – полный перебор вариантов, число которых равно факториалу от **N-1**. Это число с увеличением **N** растет очень быстро, быстрее, чем любая степень **N**. Уже для **N = 20** такое решение требует огромного времени вычислений: компьютер, проверяющий 1 млн вариантов в секунду, будет решать задачу «в лоб» около четырёх тысяч лет. Поэтому математики прилагали большие усилия для того, чтобы сократить перебор – не рассматривать те варианты, которые заведомо не дают лучших результатов, чем уже полученные. В реальных ситуациях нередко оказываются полезны приближенные решения, которые не гарантируют точного оптимума, но позволяют получить приемлемый вариант.

Приведем формулировки еще некоторых задач, которые решаются с помощью теории графов. Алгоритмы их решения вы можете найти в литературе или в Интернете.

Задача 4 (о максимальном потоке). *Есть система труб, которые имеют соединения в* **N** *узлах. Один узел* **S** *является источником, еще один – стоком* **T**. *Известны пропускные способности каждой трубы. Надо найти наибольший поток (количество жидкости, перетекающее за единицу времени) от источника к стоку.*

Задача 5. *Имеется* **N** *населенных пунктов, в каждом из которых живет* $\mathbf{p_i}$ *школьников* **(i=1,...,N)**. *Надо разместить школу в одном из них так, чтобы общее расстояние, проходимое всеми учениками по дороге в школу, было минимальным. В каком пункте нужно разместить школу?*

Задача 6 (о наибольшем паросочетании). *Есть* **M** *мужчин и* **N** *женщин. Каждый мужчина указывает несколько (от* **0** *до* **N***) женщин, на которых он согласен жениться. Каждая женщина указывает несколько мужчин (от* **0** *до* **M***), за которых она согласна выйти замуж. Требуется заключить наибольшее количество моногамных браков.*

? Контрольные вопросы

1. Что такое граф?
2. Как обычно задаются связи узлов в графах?
3. Что такое матрица смежности?
4. Что такое петля? Как «увидеть» её в матрице смежности?
5. Что такое путь?
6. Какой граф называется связным?
7. Что такое орграф?
8. Как по матрице смежности отличить орграф от неориентированного графа?
9. Что такое взвешенный граф? Как может храниться в памяти информация о нём?
10. Что такое «жадный алгоритм»? Всегда ли он позволяет найти лучшее решение?
11. Подумайте, как можно было бы ускорить работу алгоритма Крускала с помощью предварительной сортировки рёбер.

Задачи

1. Напишите программу, которая вводит из файла весовую матрицу графа и строит для него минимальное остовное дерево.
2. Оцените асимптотическую сложность алгоритма Крускала.
3. Напишите программу, которая вводит из файла весовую матрицу графа, затем вводит с клавиатуры номера начальной и конечной вершин и определяет оптимальный маршрут.
4. Напишите программу, которая вводит из файла весовую матрицу графа и определяет длины всех оптимальных маршрутов с помощью алгоритма Флойда-Уоршелла.
5. Оцените асимптотическую сложность алгоритмов Дейкстры и Флойда-Уоршелла.
6. Напишите программу, которая решает задачу коммивояжера для 5 городов методом полного перебора. Можно ли использовать ее для 50 городов?
7. *Напишите программу, которая решает задачу размещении школы. Для определения кратчайших путей используйте алгоритм Флойда-Уоршелла. Весовую матрицу графа вводите из файла.

§ 45. Динамическое программирование

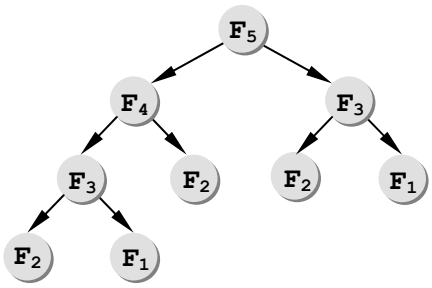
Что такое динамическое программирование?

Мы уже сталкивались с последовательностью чисел Фибоначчи (см. учебник 10 класса):

$$F_1 = F_2 = 1;\ F_n = F_{n-1} + F_{n-2} \text{ при } n > 2.$$

Для их вычисления можно использовать рекурсивную функцию:

```
int Fib ( int N )
{
  if ( N<3 )
       return 1;
  else return Fib(N-1) + Fib(N-2);
}
```



Каждое из этих чисел связано с предыдущими, вычисление F_5 приводит к рекурсивным вызовам, которые пока-

заны на рисунке справа. Таким образом, мы 2 раза вычислили F_3, три раза F_2 и два раза F_1. Рекурсивное решение очень простое, но оно неоптимально по быстродействию: компьютер выполняет лишнюю работу, повторно вычисляя уже найденные ранее значения.

Какой же выход? Напрашивается такое решение – хранить все предыдущие числа Фибоначчи в массиве. Пусть этот массив называется **F** (выделяем на 1 элемент больше, чтобы использовать элементы, начиная с номера 1):

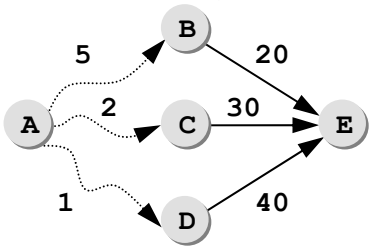
```
const int N=10;
int F[N+1];
```

Тогда для вычисления всех чисел Фибоначчи от F_1 до F_N можно использовать цикл:

```
F[1]=1; F[2]=1;
for ( i=3; i<=N; i++ )
  F[i]=F[i-1]+F[i-2];
```

Динамическое программирование – это способ решения сложных задач путем сведения их к более простым задачам того же типа

Такой подход впервые систематически применил американский математик Р. Беллман при решении сложных многошаговых задач оптимизации. Его идея состояла в том, что оптимальная последовательность шагов оптимальна на любом участке. Например, пусть нужно перейти из пункта A в пункт E через один из пунктов B, C или D (числами обозначена «стоимость» маршрута):



Пусть уже известны оптимальные маршруты из пунктов B, C и D в пункт E (они обозначены сплошными линиями) и их «стоимость». Тогда для нахождения оптимального маршрута из A в E нужно выбрать вариант, который даст минимальную стоимость по сумме двух шагов. В данном случае это маршрут A–B–E, стоимость которого равна 25. Как видим, такие задачи решаются «с конца», то есть решение начинается от конечного пункта.

В информатике динамическое программирование часто сводится к тому, что мы храним в памяти решения всех задач меньшей размерности. За счёт этого удается ускорить выполнение программы. Например, на одном и том же компьютере вычисление $\mathbf{F}_{45}$ с помощью рекурсивной функции требует около 8 секунд, а с использованием массива – менее 0,01 с.

Заметим, что в данной простейшей задаче можно обойтись вообще без массива:

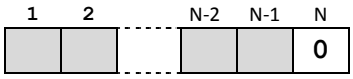
```
f2=1; f1=1; fN=1;
for ( i=3; i<=N; i++ )
  {
  fN=f1+f2;
  f2=f1;
  f1=fN;
  }
```

Задача 1. *Найти количество K_N цепочек, состоящих из* **N** *нулей и единиц, в которых нет двух стоящих подряд единиц.*

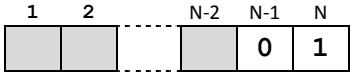
При больших **N** решение задачи методом перебора потребует огромного времени вычисления. Для того, чтобы использовать метод динамического программирования, нужно

1) выразить K_N через предыдущие значения последовательности $K_1, K_2, \ldots, K_{N-1}$;
2) выделить массив для хранения всех предыдущих значений $K_i (i = 1, \ldots N-1)$.

Самое главное – вывести рекуррентную формулу, выражающую K_N через решения аналогичных задач меньшей размерности. Рассмотрим цепочку из **N** бит, первый элемент которой – 0.



Поскольку дополнительный 0 не может привести к появлению двух соседних единиц, подходящих последовательностей длиной **N** с нулем в начале существует столько, сколько подходящих последовательностей длины **N-1**, то есть K_{N-1}. Если же первый символ – 1, то вторым обязательно должен быть 0, а остальная цепочка из **N-2** битов должна быть «правильной». Поэтому подходящих последовательностей длиной **N** с единицей в начале существует столько, сколько подходящих последовательностей длины **N-2**, то есть K_{N-2}.



В результате получаем $K_N = K_{N-1} + K_{N-2}$. Значит, для вычисления очередного числа нам нужно знать два предыдущих.

Теперь рассмотрим простые случаи. Очевидно, что есть две последовательности длиной 1 (0 и 1), то есть $K_1 = 2$. Далее, есть 3 подходящих последовательности длины 2 (00, 01 и 10), поэтому $K_2 = 3$. Легко понять, что решение нашей задачи – число Фибоначчи: $K_N = F_{N+2}$.

Поиск оптимального решения

Задача 2. *В цистерне N литров молока. Есть бидоны объемом 1, 5 и 6 литров. Нужно разлить молоко в бидоны так, чтобы все используемые бидоны были заполнены и их количество было минимальным*.

Человек, скорее всего, будет решать задачу перебором вариантов. Наша задача осложняется тем, что требуется написать программу, которая решает задачу для любого введенного числа **N**.

Самый простой подход – заполнять сначала бидоны самого большого размера (6 л), затем – меньшие и т.д. Это так называемый «жадный» алгоритм. Как вы знаете, он не всегда приводит к оптимальному решению. Например, для N = 10 «жадный» алгоритм даёт решение 6+1+1+1+1 – всего 5 бидонов, в то время как можно обойтись двумя (5+5).

Как и в любом решении, использующем динамическое программирование, главная проблема – составить рекуррентную формулу. Сначала определим оптимальное число бидонов K_N, а потом подумаем, как определить какие именно бидоны нужно использовать.

Представим себе, что мы выбираем бидоны постепенно. Тогда последний выбранный бидон может иметь, например, объем 1 л, в этом случае $K_N = 1 + K_{N-1}$. Если последний бидон имеет объём 5 л, то $K_N = 1 + K_{N-5}$, а если 6 л – $K_N = 1 + K_{N-6}$. Так как нам нужно выбрать минимальное значение, то

$$K_N = 1 + \min\left(K_{N-1}, K_{N-5}, K_{N-6}\right).$$

Вариант, выбранный при поиске минимума, определяет последний добавленный бидон, его нужно сохранить в отдельном массиве **P**. Этот массив будет использован для определения количества выбранных бидонов каждого типа. В качестве начальных значений берем $K_0 = 0$ и $P_0 = 0$.

Полученная формула применима при $N \ge 6$. Для меньших N используются только те данные, которые есть в таблице. Например,

$$K_3 = 1 + K_2 = 3, \quad K_5 = 1 + \min\left(K_4, K_0\right) = 1.$$

На рисунке показаны массивы для $N = 10$.

N	0	1	2	3	4	5	6	7	8	9	10
K	0	1	2	3	4	1	1	2	3	4	2
P	0	1	1	1	1	5	6	1	1	1	5

Как по массиву **P** определить оптимальный состав бидонов? Пусть, для примера $N = 10$. Из массива **P** находим, что последний добавленный бидон имеет объем 5 л. Остается 10 – 5= 5 л, в эле-

менте **P[5]** тоже записано значение 5, поэтому второй бидон тоже имеет объём 5 л. Остаток 0 л означает, что мы полностью определили набор бидонов.

Можно заметить, что такая процедура очень похожа на алгоритм Дейкстры, и это не случайно. В алгоритмах Дейкстры и Флойда-Уоршелла по сути используется метод динамического программирования.

Задача 3 (Задача о куче). *Из камней весом* $p_i (i = 1,\ldots,N)$ *набрать кучу весом ровно W или, если это невозможно, максимально близкую к W (но меньшую, чем W). Все веса камней и значение W – целые числа.*

Эта задача относится к трудным задачам целочисленной оптимизации, которые решаются только полным перебором вариантов. Каждый камень может входить в кучу (обозначим это состояние как 1) или не входить (0). Поэтому нужно выбрать цепочку, состоящую из N бит. При этом количество вариантов равно 2^N, и при больших N полный перебор практически невыполним.

Динамическое программирование позволяет найти решение задачи значительно быстрее. Идея состоит в том, чтобы сохранять в массиве решения всех более простых задач этого типа (при меньшем количестве камней и меньшем весе W).

Построим матрицу **T**, где элемент **T[i][w]** – это оптимальный вес, полученный при попытке собрать кучу весом **w** из **i** первых по счёту камней. Очевидно, что первый столбец заполнен нулями (при заданном нулевом весе никаких камней не берём).

Рассмотрим первую строку (есть только один камень). В начале этой строки будут стоять нули, а дальше, начиная со столбца $\mathbf{p_1}$ – значения $\mathbf{p_1}$ (взяли единственный камень). Это простые варианты задачи, решения для которых легко подсчитать вручную. Рассмотрим пример, когда требуется набрать вес 8 из камней весом 2, 4, 5 и 7 единиц:

	0	1	2	3	4	5	6	7	8
2	0	0	2	2	2	2	2	2	2
4	0								
5	0								
7	0								

Теперь предположим, что строки с 1-ой по **(i-1)**-ую уже заполнены. Перейдем к **i**-ой строке, то есть добавим в набор **i**-ый камень. Он может быть взят или не взят в кучу. Если мы не добавляем его в кучу, то **T[i][w] = T[i-1][w]**, то есть решение не меняется от добавления в набор нового камня. Если камень с весом p_i добавлен в кучу, то остается «добрать» остаток $w - p_i$ оптимальным образом (используя только предыдущие камни), то есть $\mathbf{T[i][w] = T[i-1][w-p_i]+p_i}$.

Как же выбрать, «брать или не брать»? Проверить, в каком случае полученный вес будет больше (ближе к **w**). Таким образом, получается рекуррентная формула для заполнения таблицы:

при $\mathbf{w < p_i}$: **T[i][w] = T[i-1][w]**

при $\mathbf{w \ge p_i}$: $\mathbf{T[i][w] = max\ (\ T[i-1][w]\ ,\ T[i-1][w-p_i] + p_I\ )}$

Используя эту формулу, заполняем таблицу по строкам, сверху вниз; в каждой строке – слева направо:

	0	1	2	3	4	5	6	7	8
2	0	0	2	2	2	2	2	2	2
4	0	0	2	2	4	4	6	6	6
5	0	0	2	2	4	5	6	7	7
7	0	0	2	2	4	5	6	7	7

Видим, что сумму 8 набрать невозможно, ближайшее значение – 7 (правый нижний угол таблицы).

Эта таблица содержит все необходимые данные для определения выбранной группы камней. Действительно, если камень с весом $\mathbf{p_i}$ не включен в набор, то **T[i][w] = T[i-1][w]**, то есть, число в таблице не меняется при переходе на строку вверх. Начинаем с левого нижнего угла таблицы, идем вверх, пока значения в столбце равны 7. Последнее такое значение – для камня с

весом 5, поэтому он и выбран. Вычитая его вес из суммы, получаем 7 – 5 = 2, переходим во второй столбец на одну строку вверх, и снова идем вверх по столбцу, пока значение не меняется (равно 2). Так как мы успешно дошли до самого верха таблицы, взят первый камень с весом 2.

Как мы уже отмечали, количество вариантов в задаче для N камней равно 2^N, то есть алгоритм полного перебора имеет асимптотическую сложность $O(2^N)$. В данном алгоритме количество операций равно числу элементов таблицы, то есть сложность нашего алгоритма – $O(N \cdot W)$. Однако нельзя сказать, что он имеет линейную сложность, так как есть еще сильная зависимость от заданного веса W. Такие алгоритмы называют *псевдополиномиальными*, то есть «как бы полиномиальными». В них ускорение вычислений достигается за счёт использования дополнительной памяти для хранения промежуточных результатов.

Количество решений

Задача 4. *У исполнителя Утроитель две команды, которым присвоены номера*:

```
1. прибавь 1
2. умножь на 3
```

Первая из них увеличивает число на экране на 1, вторая – утраивает его. Программа для Утроителя – это последовательность команд. Сколько есть программ, которые число 1 преобразуют в число 20?

Заметим, что при выполнении любой из команд число увеличивается (не может уменьшаться). Начнем с простых случаев, с которых будем начинать вычисления. Понятно, что для числа 1 существует только одна программа – пустая, не содержащая ни одной команды. Для числа 2 есть тоже только одна программа, состоящая из команды сложения. Если через K_N обозначить количество разных программ для получения числа N из 1, то $K_1 = K_2 = 1$.

Теперь рассмотрим общий случай, чтобы построить рекуррентную формулу, связывающую K_N с предыдущими элементами последовательности $K_1, K_2, \ldots, K_{N-1}$, то есть с решениями таких же задач для меньших N.

Если число N не делится на 3, то оно могло быть получено только последней операцией сложения, поэтому $K_N = K_{N-1}$. Если N делится на 3, то последней командой может быть как сложение, так и умножение. Поэтому нужно сложить K_{N-1} (количество программ с последней командой сложения) и $K_{N/3}$ (количество программ с последней командой умножения). В итоге получаем:

$$K_N = \begin{cases} K_{N-1}, & \text{если } N \text{ не делится на 3} \\ K_{N-1} + K_{N/3}, & \text{если } N \text{ не делится на 3} \end{cases}$$

Остается заполнить таблицу для всех значений от 1 до заданного $N = 20$. Для небольших значений N эту задачу легко решить вручную:

N	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
K_N	1	1	2	2	2	3	3	3	5	5	5	7	7	7	9	9	9	12	12	12

Заметим, что количество вариантов меняется только в тех столбцах, где N делится на 3, поэтому из всей таблицы можно оставить только эти столбцы (и первый):

N	1	3	6	9	12	15	18	21
K_N	1	2	3	5	7	9	12	15

Заданное число 20 попадает в последний интервал (от 18 до 20), поэтому ответ в данной задаче – 12.

При составлении программы с полной таблицей нужно выделить в памяти целочисленный массив **K**, индексы которого изменяются от 0 до N, и заполнить его (начиная с элемента **K[1]**) по приведённым выше формулам:

```
K[1] = 1;
for ( i = 2; i <= N; i ++ )
  {
```

```
K[i] = K[i-1];
if ( i % 3 == 0 )
  K[i] = K[i] + K[i/3];
}
```

Ответом будет значение **K[N]**.

Задача 5 (Размен монет). *Сколькими различными способами можно выдать сдачу размером W рублей, если есть монеты достоинством $p_i (i = 1,\ldots,N)$? Для того, чтобы сдачу всегда можно было выдать, будем предполагать, что в наборе есть монета достоинством 1 рубль ($p_1 = 1$).*

Это задача, так же, как и задача о куче, решается только полным перебором вариантов, число которых при больших N очень велико. Будем использовать динамическое программирование, сохраняя в массиве решения всех задач меньшей размерности (для меньших значений N и W).

В матрице **T** значение **T[i][w]** будет обозначать количество вариантов сдачи размером **w** рублей (**w** изменяется от 0 до W) при использовании первых **i** монет из набора. Очевидно, что при нулевой сдаче есть только один вариант (не дать ни одной монеты), так же и при наличии только одного типа монет (напомним, что $p_1 = 1$) есть тоже только один вариант. Поэтому нулевой столбец и первую строку таблицы можно заполнить сразу единицами. Для примера мы будем рассматривать задачу для $W = 10$ и набора монет достоинством 1, 2, 5 и 10 рублей:

	0	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1	1
2	1										
5	1										
10	1										

Таким образом, мы определили простые базовые случаи, от которых «отталкивается» рекуррентная формула.

Теперь рассмотрим общий случай. Заполнять таблицу будем по строкам, слева направо. Для вычисления **T[i][w]** предположим, что мы добавляем в набор монету достоинством $\mathbf{p_i}$. Если сумма **w** меньше, чем $\mathbf{p_i}$, то количество вариантов не увеличивается, и **T[i][w] = T[i-1][w]**. Если сумма больше $\mathbf{p_i}$, то к этому значению нужно добавить количество вариантов с «участием» новой монеты. Если монета достоинством $\mathbf{p_i}$ использована, то нужно учесть все варианты «разложения» остатка $\mathbf{w-p_i}$ на все доступные монеты, то есть

T[i][w] = T[i-1][w] + T[i][w-p_i].

В итоге получается рекуррентная формула

при $\mathbf{w < p_i}$: **T[i,w] = T[i-1][w]**

при $\mathbf{w \ge p_i}$: **T[i,w] = T[i-1][w] + T[i][w-p_i]**

которая используется для заполнения таблицы:

	0	1	2	3	4	5	6	7	8	9	10
1	1	1	1	1	1	1	1	1	1	1	1
2	1	1	2	2	3	3	4	4	5	5	6
5	1	1	2	2	3	4	5	6	7	8	10
10	1	1	2	2	3	4	5	6	7	8	(11)

Ответ к задаче находится в правом нижнем углу таблицы.

Вы могли заметить, что решение этой задачи очень похоже на решение задачи о куче камней. Это не случайно, две эти задачи относятся к классу сложных задач, для решения которых известны только переборные алгоритмы. Использование методов динамического программирования позволяет ускорить решение за счёт хранения промежуточных результатов, однако требует дополнительного расхода памяти.

? Контрольные вопросы

1. Что такое динамическое программирование?
2. Какой смысл имеет выражение «динамическое программирование» в теории многошаговой оптимизации?
3. Какие шаги нужно выполнить, чтобы применить динамическое программирование к решению какой-либо задачи?
4. За счет чего удается ускорить решение сложных задач методом динамического программирования?
5. Какие ограничения есть у метода динамического программирования?

Задачи

1. Напишите программу, которая определяет оптимальный набор бидонов в задаче с молоком. С клавиатуры или из файла вводится объём цистерны, количество типов бидонов и их размеры.
2. Напишите программу, которая решает задачу о куче камней заданного веса, рассмотренную в тексте параграфа.
3. * Задача о ранце. Есть N предметов, для каждого из которых известен вес $p_i (i = 1, \ldots, N)$ и стоимость $c_i (i = 1, \ldots, N)$. В ранец можно взять предметы общим весом не более W. Напишите программу, которая определяет самый дорогой набор предметов, который можно унести в ранце.
4. У исполнителя Калькулятор две команды, которым присвоены номера:
   ```
   1. прибавь 1
   3. умножь на 4
   ```
   Напишите программу, которая вычисляет, сколько существует различных программ, преобразующих число M в число N, оба эти числа вводятся с клавиатуры.
5. У исполнителя Калькулятор три команды, которым присвоены номера:
   ```
   1. прибавь 1
   2. умножь на 3
   3. умножь на 4
   ```
   Напишите программу, которая вычисляет, сколько существует различных программ, преобразующих число M в число N, оба эти числа вводятся с клавиатуры.
6. У исполнителя Калькулятор две команды, которым присвоены номера:
   ```
   1) прибавь 1
   2) увеличь каждый разряд числа на 1
   ```
   Сколько есть программ, которые число 24 преобразуют в число 46?
7. У исполнителя Калькулятор две команды, которым присвоены номера:
   ```
   1) прибавь 1
   2) увеличь каждый разряд числа на 1
   ```
   Сколько существует программ, которые число 26 преобразуют в число 49?
8. *Прямоугольный остров разделён на квадраты так, что его размеры — N × M квадратов. В каждом квадрате зарыто некоторое число золотых монет, эти данные хранятся в матрице (двумерном массиве) **Z**, где **Z[i][j]** — число монет в квадрате с координатами **(i, j)**. Пират хочет пройти из юго-западного угла острова в северо-восточный, причём он может двигаться только на север или на восток. Как пирату собрать наибольшее количество монет? Напишите программу, которая находит оптимальный путь пирата и число монет, которое ему удастся собрать.

Самое важное в главе 6:

- Структура – это сложный тип данных, который позволяет объединить данные разных типов. Элементы структуры называют полями. При обращении к полям структуры используется точечная нотация:
  <имя структуры>.<имя поля>.
- Указатель – это переменная, в которой можно хранить адрес другой переменной заданного типа. В указателе можно запомнить адрес массива, метод для которого выделяется в памяти во время работы программы.
- Динамические массивы – это массивы, память для которых выделяется во время работы программы. Динамический массив в программе на языках C и C++ – это указатель, в который записывается адрес выделенного блока памяти. При записи такой переменной в файл сохранится только значение указателя, а значения элементов массива будут потеряны.
- Список – это упорядоченный набор элементов одного типа, для которых введены операции вставки (включения) и удаления (исключения).
- Стек – это линейный список, в котором добавление и удаление элементов разрешается только с одного конца. Системный стек применяется для хранения адресов возврата из подпрограмм и размещения локальных переменных.
- Дерево – это структура данных, которая моделирует иерархию – многоуровневую структуру. Как правило, дерево определяется с помощью рекурсии, поэтому для его обработки удобно использовать рекурсивные алгоритмы. Деревья используются в задачах поиска, сортировки, вычисления арифметических выражений.
- Граф – это набор вершин и связывающих их рёбер. Информация о графе чаще всего хранится в виде матрицы смежности или весовой матрицы. Наиболее известные задачи, которые решаются с помощью теории графов – поиск оптимальных маршрутов.
- Динамическое программирование – это метод, позволяющий ускорить решение задачи за счет хранения решений более простых задач того же типа. Для его использования нужно вывести рекуррентную формулу, связывающее решение задачи с решением подобных задач меньшей размерности, и определить простые базовые случаи (условие окончания рекурсии).