C++

Генерация псевдослучайных чисел в C++

Что бы такого придумать…

Генерация псевдослучайных чисел в C++

В стандартной библиотеке C++ (в заголовочном файле **cstdlib**) существует функция rand(), которая при каждом вызове возвращает псевдослучайное целое число в диапазоне от 0 до константы RAND_MAX (обычно она равна 32767, но её значение зависит от настроек среды и, соответственно, может изменяться, самый простой способ избавиться от сомнений — вывести значение этой константы на экран).

Генерация псевдослучайных чисел в C++

Если создать и запустить программу, которая выводит два случайных числа на экран несколько раз, то можно увидеть, что от запуска к запуску числа повторяются, хотя до первого запуска, конечно, нельзя было предсказать эту пару значений.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
 cout << "Выводим 2 случайных значения от 0 до " << RAND_MAX << endl;
 cout << rand() << ' ' << rand();
 return 0;
}
```

Генерация псевдослучайных чисел в C++

Функция rand(), на самом деле, выбирает числа из последовательности значений, вычисленных по специальному алгоритму, где базой для построения этой последовательности является некоторый числовой аргумент (обычно называемый seed — «зерно» с англ.).

Значение этого аргумента неизменно между запусками программы, но изменится, если, например, перезагрузить компьютер.

Генерация псевдослучайных чисел в C++

Иметь одинаковые наборы при каждом запуске удобно, например, в процессе тестирования. Но для реальной программы нужно, чтобы последовательность псевдослучайных чисел менялась даже при двух последовательных запусках программы. Для её перемещивания существует функция **srand(seed)**, где *seed* — это некоторое значение типа unsigned int.

Соответственно, если в качестве аргумента в эту функцию передавать разные значения при каждом запуске программы, то и наборы получаемые с помощью rand() — будут разными.

Генерация псевдослучайных чисел в C++

Традиционно в качестве аргумента для функции srand() используют текущее значение времени в формате **UNIXTIME** (количество секунд, прошедших с 1 января 1970 года).

Его можно получить с помощью функции time (NULL), которая также является частью стандартной библиотеки и описана в заголовочном файле ctime.

Генерация псевдослучайных чисел в C++

Предыдущий пример можно легко модернизировать до такого состояния, чтобы числа не повторялись между запусками:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
 cout << "Выводим 2 случайных значения от 0 до " << RAND_MAX << endl;
 srand(time(NULL));
 cout << rand() << ' ' << rand();
 return 0;
}
```

Генерация псевдослучайных чисел в C++

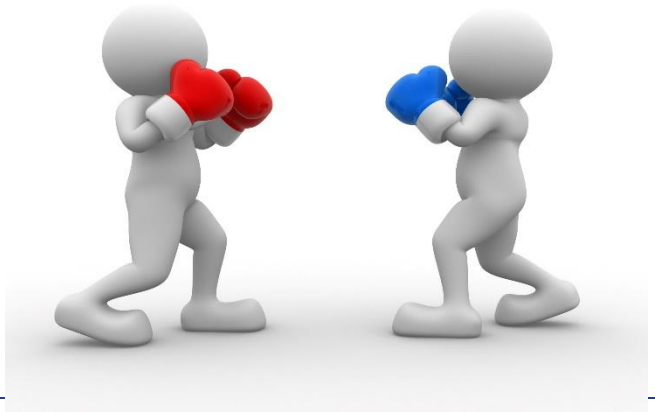
Отметим, что перемешивать последовательность псевдослучайных чисел нет смысла чаще одного раза в процессе исполнения программы.

В реальных задачах требуется получать числа из определенных промежутков (не от 0 до RAND_MAX).

Генерация псевдослучайных чисел в C++

Проблема решается с использованием пары арифметических операций. Для того, чтобы сузить промежуток до [0;n−1] достаточно применить к значению функции rand() операцию остатка от деления на n (оператор «%»).

То есть выражение rand()%n будет всегда возвращать псевдослучайное число из отрезка от 0 до n−1 (заметим, что целых чисел там ровно n штук), при это распределение получаемых значений по классам вычетов будет достаточно равномерным (а если RAND_MAX+1 кратно n, то полностью равномерным).



Генерация псевдослучайных чисел в C++

Когда требуется получить промежуток начинающийся не от 0, то результат функции просто сдвигают в положительном или отрицательном направлении, соответственно, прибавляя или вычитая нужное значение. То есть выражение rand()%n+a будет всегда возвращать псевдослучайное число из отрезка [a;a+n−1] (в нём тоже ровно n целых чисел).

Итак, чтобы получить псевдослучайное число из нужного отрезка, нужно сузить значение функции rand() до длины этого отрезка и сдвинуть на значение левого конца отрезка.

Генерация псевдослучайных чисел в C++

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;
int main()
{
    int a, b;
    cout << "Введите левый конец отрезка "; cin >> a;
    cout << "Введите правый конец отрезка "; cin >> b;
    srand(time(NULL));
    int rnd = rand()%(b-a+1)+a;
    cout << "Псевдослучайное число из отрезка [" << a <<
';' << b <<"]: " << rnd;
 }
```

Генерация псевдослучайных чисел в C++

Задачи

1. Вывести на экран 10 чисел.
2. Вывести на экран 5 натуральных чисел из диапазона от 50 до 75.
3. Вывести на экран 5 вещественных чисел из диапазона от 5 до 8.

Массивы в C++

На С я могу просто делать ошибки, на С++ я могу их наследовать!

Массивы в C++

Массив — это конечная последовательность элементов одного типа, доступ к каждому элементу в которой осуществляется по его индексу.

Размер или **длина массива** — это общее количество элементов в массиве. Размер массива задаётся при создании массива и не может быть изменён в дальнейшем, т. е. нельзя убрать элементы из массива или добавить их туда, но можно в существующие элементы присвоить новые значения.

Массивы в C++

- Индекс начального элемента — 0, следующего за ним — 1 и т. д. Индекс последнего элемента в массиве — на единицу меньше, чем размер массива. В памяти все элементы массива располагаются последовательно (т.е. между соседними элементами массива не может размещаться какая-то посторонняя переменная или, например, элементы другого массива).

Массивы в C++

Создается массив по следующей схеме:

тип имя[размер];

Где тип — это тип элементов массива, имя — это допустимый и уникальный в данной области видимости идентификатор, а размер — это положительный литерал, переменная или константа целого типа.

Массивы в C++

Можно не указывать размер массива (оставив пустые квадратные скобки после его имени), но тогда необходимо сразу перечислить все его элементы (инициализировать массив), в этом случае размер автоматически вычислит компилятор. Примеры корректного объявления массивов:

```
int mas1[4];
int mas2[] = {3,-7,9,1200,-713};
float mas3[400];
const int n = 173;
double mas4[n];
```

Массивы в C++

- Массив объявленный как char mas[128]; будет занимать в памяти — 128 байт (столько же заняли бы 128 разных переменных типа char, каждая из которых занимает по байту). При этом размер массива не может быть сколь угодно большим (например, если массив объявлен не как глобальный, то его максимально допустимый размер зависит от доступного стека).

Массивы в C++

Массив **mas3** займет в памяти 1600 байт.

Сколько места в памяти займут остальные из объявленных в примере массивов?

```
int mas1[4];
int mas2[] = {3,-7,9,1200,-713}
float mas3[400];
const int n = 173;
double mas4[n];
```

Массивы в C++

В неинициализированном массиве (по аналогии с неинициализированной переменной) будут храниться заранее неизвестные значения (какой-то «мусор», ранее записанный другими программами или даже вашей программой в выделяемую для объявленного массива память).

Массивы в C++

Чтобы обратиться к какому-то из элементов массива для того, чтобы прочитать или изменить его значение, нужно указать имя массива и за ним индекс элемента в квадратных скобках.

Элемент массива с конкретным индексом ведёт себя также, как переменная.

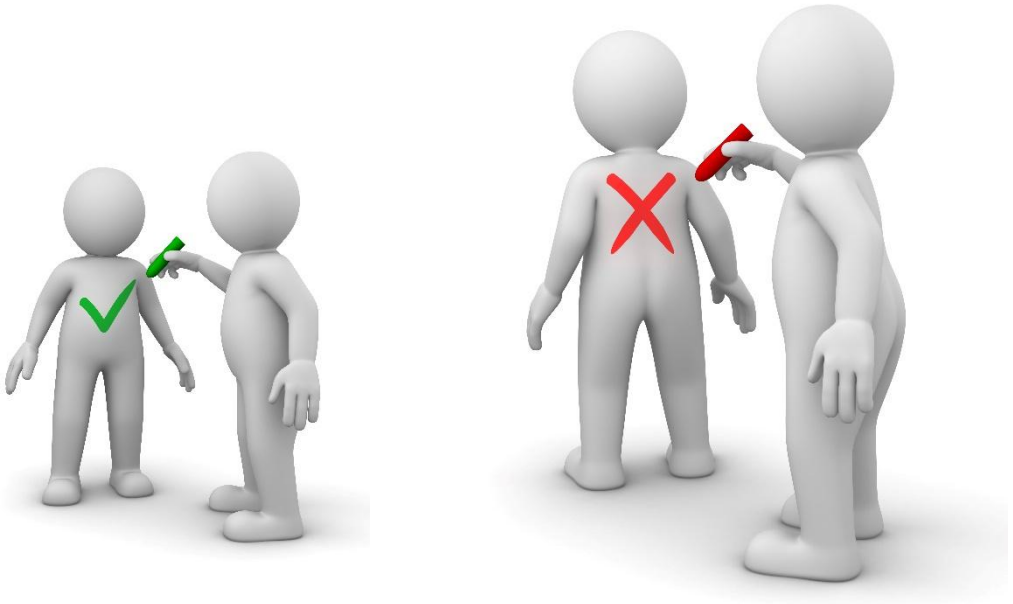
Например, чтобы вывести значения первого и последнего элементов массива **mas1** надо написать в программе:

```
cout << mas1[0];
cout << mas1[3];
```

Массивы в C++

А чтобы присвоить новые значения (10, 20, 30, 40) всем элементам массива, потребуется написать в программе:

- mas1[0] = 10;
- mas1[1] = 20;
- mas1[2] = 30;
- mas1[3] = 40;



Массивы в C++

Уже из последнего примера видно, что для того, чтоб обратиться ко всем элементам массива, приходится повторять однотипные действия.

Для многократного повторения подобных операций используются циклы. Соответственно, мы могли бы заполнить массив нужными элементами с помощью цикла:

```
for(int i=0; i<4; i++)
   {
   mas1[i] = (i+1) * 10;
   }
```

Массивы в C++

А после этого несложно вывести все элементы массива на экран:

```
for(int i=0; i<4; i++)
   {
    cout << mas1[i] << ' ';
   }
```

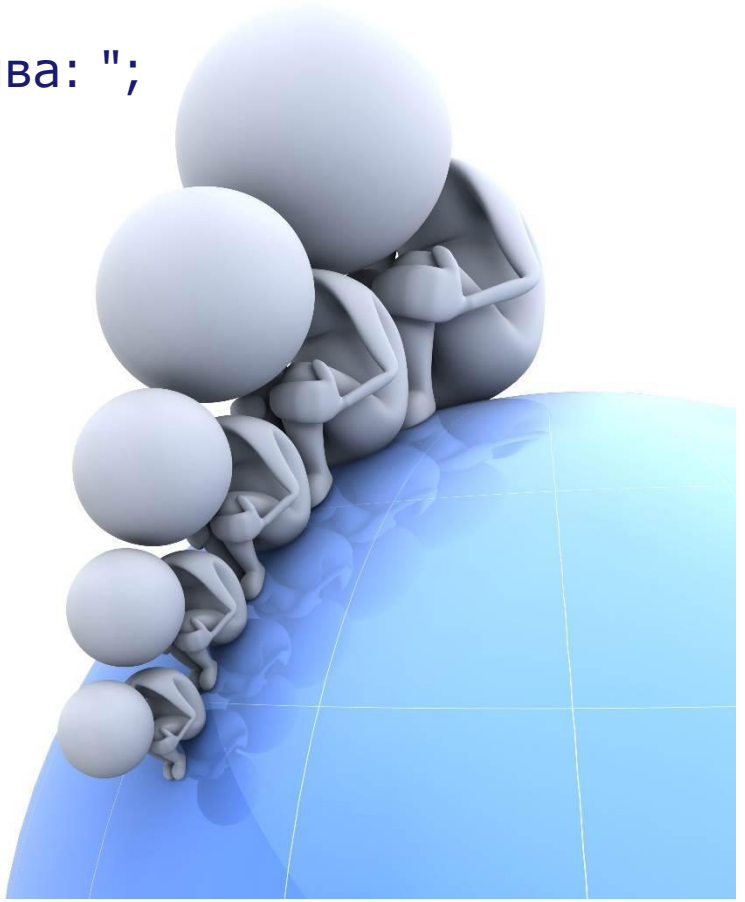
Массивы в C++

Размер каждого создаваемого массива правильнее хранить в отдельной константе, ведь если придётся изменить программу, то размер достаточно будет

отредактировать в одном месте (там, где инициализируется константа), не придётся править параметр каждого цикла, обрабатывающего массив.

Пример программы

```
#include <iostream>
using namespace std;
int main() {
    cout << "Укажите размер массива: ";
    int n;
    cin >> n;
    const int dim = n;
    int arr[dim];
    for(int i=0; i<dim; i++)
       {
          arr[i] = i+1;
       }
    for(int i=dim-1; i>=0; i--)
      {
          cout << arr[i] << ' ';
      }
 }
```

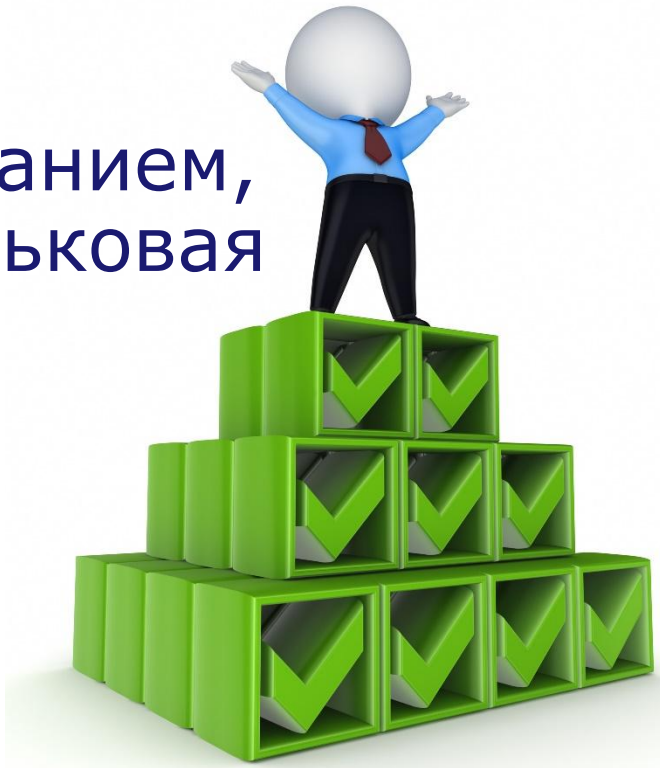


Сортировка массивов в C++

Я не перфекционист, я просто люблю порядок

Алгоритмы сортировки

1. Сортировка выбором
2. Сортировка пузырьком
3. Шейкерная сортировка (Сортировка перемешиванием, Двунаправленная пузырьковая сортировка)
4. Гномья сортировка
5. Сортировка вставками
6. Сортировка слиянием
7. Сортировка Шелла
8. Быстрая сортировка



Алгоритм «Сортировка выбором»

Является одним из самых простых алгоритмов сортировки массива. Смысл в том, чтобы идти по массиву и каждый раз искать минимальный элемент массива, обменивая его с начальным элементом неотсортированной части массива. На небольших массивах может оказаться даже эффективнее, чем более сложные алгоритмы сортировки, но в любом случае проигрывает на больших массивах. Число обменов элементов по сравнению с "пузырьковым" алгоритмом N/2, где N - число элементов массива.

Алгоритм «Сортировка выбором»

Алгоритм:

1. Находим минимальный элемент в массиве.
2. Меняем местами минимальный и первый элемент местами.
3. Опять ищем минимальный элемент в неотсортированной части массива.
4. Меняем местами уже второй элемент массива и минимальный найденный, потому как первый элемент массива является отсортированной частью.
5. Ищем минимальные значения и меняем местами элементы, пока массив не будет отсортирован до конца.

Алгоритм «Сортировка пузырьком»

Пожалуй самый известный алгоритм, применяемый в учебных целях, для практического же применения является слишком медленным.

Алгоритм лежит в основе более сложных алгоритмов: "Шейкерная сортировка", "Пирамидальная сортировка", "Быстрая сортировка". Примечательно то, что один из самых быстрых алгоритмов "Быстрый алгоритм" был разработан путем модернизации одного из самых худших алгоритмов "Сортировки пузырьком".

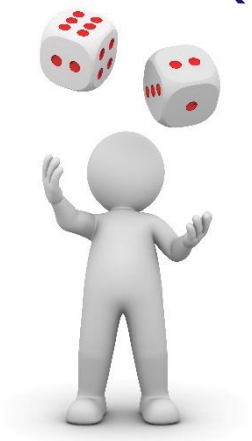
Смысл алгоритма заключается в том, что самые "легкие" элементы массива как бы "всплывают" , а самые "тяжелые" "тонут".

Отсюда и название "Сортировка пузырьком"

Алгоритм «Сортировка пузырьком»

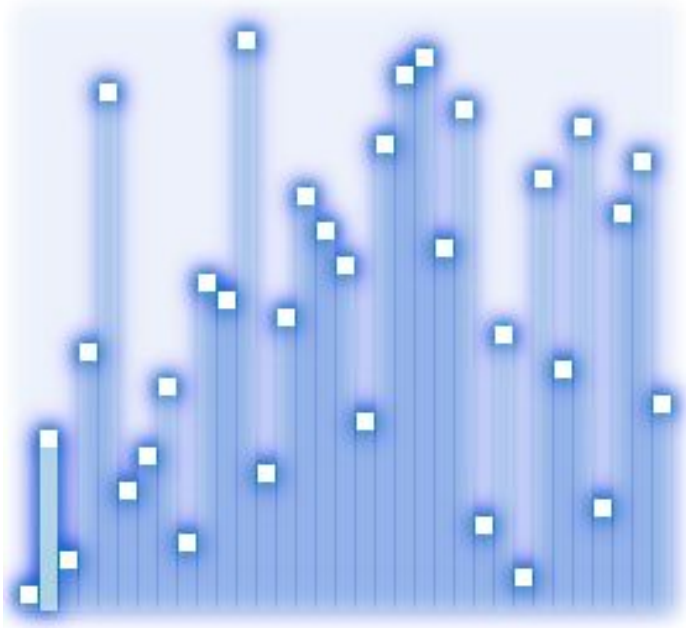
Алгоритм:

1. Каждый элемент массива сравнивается с последующим и если элемент[i] > элемент[i+1] происходит замена. Таким образом самые "легкие" элементы "всплывают" - перемещаются к началу списка,а самые тяжелые "тонут" - перемещаются к концу.
2. Повторяем Шаг 1 n-1 раз, где n - Массив. Количество ().



Алгоритм «Шейкерная сортировка»

- Алгоритм представляет собой одну из версий предыдущей сортировки - "сортировки пузырьком". Главное отличие в том, что в классической сортировке пузырьком происходит однонаправленное движение элементов снизу - вверх, то в шейкерной сортировке сначала происходит движение снизу-вверху, а затем сверху-вниз.
- Алгоритм такой же, что и у пузырьковой сортировки + добавляется цикл пробега сверху-вниз.
- В отличие от классической, используется в 2 раза меньше итераций.



Gnome Sort

Гномья сортировка

Алгоритм "Гномья сортировка"

“Глупая сортировка, простейший алгоритм сортировки всего с одним циклом…”

Хамид Сарбази-Азад

Алгоритм "Гномья сортировка"

Алгоритм так странно назван благодаря голландскому ученому Дику Груну.

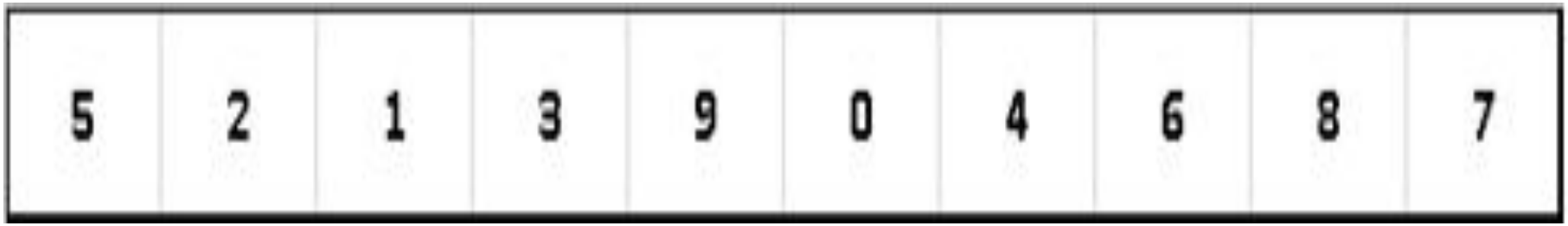
Гномья сортировка основана на технике, используемой обычным голландским садовым гномом (нидерл. tuinkabouter). Это метод, которым садовый гном сортирует линию цветочных горшков.

По существу он смотрит на следующий и предыдущий садовые горшки: если они в правильном порядке, он шагает на один горшок вперёд, иначе он меняет их местами и шагает на один горшок назад. Граничные условия: если нет предыдущего горшка, он шагает вперёд; если нет следующего горшка, он закончил.

Алгоритм не содержит вложенных циклов, а сортирует ве

Алгоритм "Гномья сортировка"

Техника сортировки



Алгоритм "Гномья сортировка"

	Гномья сортировка	Сортировка вставками
Худший случай	$O(n^2)$	$O(n^2)$
Лучший случай	$O(n)$	$O(n)$
Средний случай	$O(n^2)$	$O(n^2)$

Алгоритм "Гномья сортировка"

Это оптимизированная версия с использованием переменной j, чтобы разрешить прыжок вперёд туда, где он остановился до движения влево, избегая лишних итераций и сравнений:

```
gnomeSort(a[0..size - 1])
    i = 1;
    j = 2;
    while i < size
        if a[i - 1] <= a[i] i = j;
            j = j + 1;
        else
            swap (a[i - 1], a[i])
            i = i - 1;
            if i == 0
                i = j;
                j = j + 1;
```

Алгоритм «Сортировка вставками»

Представляет собой простой алгоритм сортировки. Смысл заключается в том, что на каждом шаге мы берем элемент, ищем для него позицию и вставляем в нужное место.

Элементарный пример: При игре в дурака, вы тянете из колоды карту и вставляете ее в соответствующее место по возрастанию в имеющихся у вас картах. Или в магазине вам дали сдачу 550 рублей- одна купюра 500, другая 50. Заглядываете в кошелек, а там купюры достоинством 10,100,1000. Вы вставляете купюру достоинством 50р. между купюрами достоинством 10р и 100р, а купюру в 500 рублей между купюрами 100р и 1000р. Получается 10,50,100,500,1000 - Вот вам алгоритм "Сортировка вставками". Таким образом с каждым шагом алгоритма, вам необходимо отсортировать подмассив данных и вставить значение в нужное место.

Алгоритм «Сортировка вставками»

Принцип действия

8 1 7 3 4

Алгоритм «Сортировка вставками»

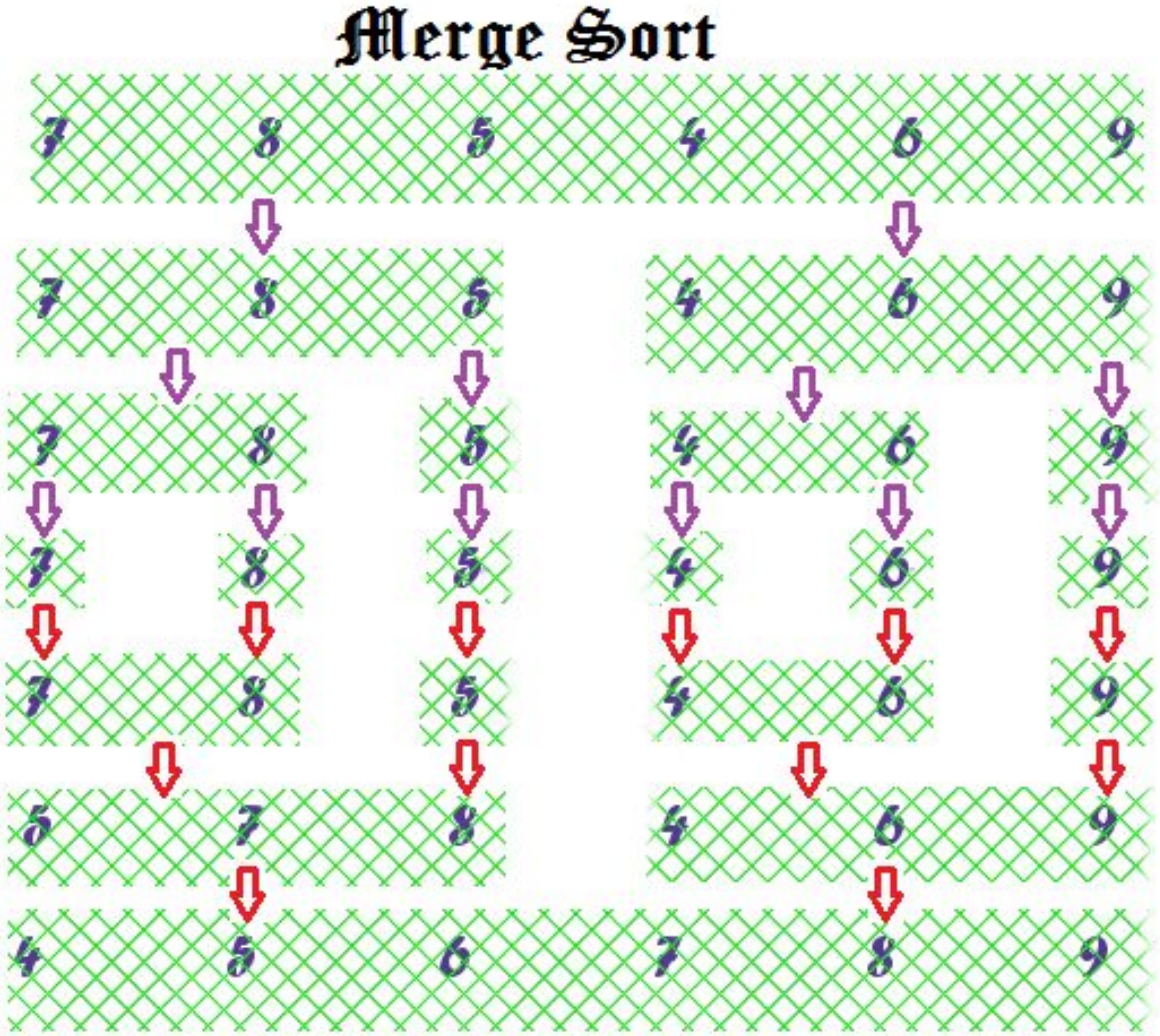
Применение способа

Этот алгоритм может оказаться эффективным на небольших и частично отсортированных списках, но скорость его работы примерно такая же, как у сортировки пузырьком и сортировки выбором, поэтому он не пригоден для серьезного практического применения.

Алгоритм Сортировка слиянием

- Массив разбивается пополам до тех пор, пока размер очередного подмассива не станет равным единице;
- Два единичных массива сливаются в общий массив, выбирается меньший элемент и записывается в свободную левую ячейку результирующего массива. После чего из двух результирующих массивов собирается третий общий отсортированный массив, и так далее.
- Элементы перезаписываются из результирующего массива в исходный.

Алгоритм Сортировка слиянием



Быстрая сортировка

- Выбирая алгоритм сортировки для практических целей, программист, вполне вероятно, остановиться на методе, называемом «Быстрая сортировка» (также «qsort» от англ. quick sort). Его разработал в 1960 году английский ученый Чарльз Хоар, занимавшийся тогда в МГУ машинным переводом. Алгоритм, по принципу функционирования, входит в класс обменных сортировок (сортировка перемешиванием, пузырьковая сортировка и др.), выделяясь при этом высокой скоростью работы.

Алгоритм быстрая сортировка

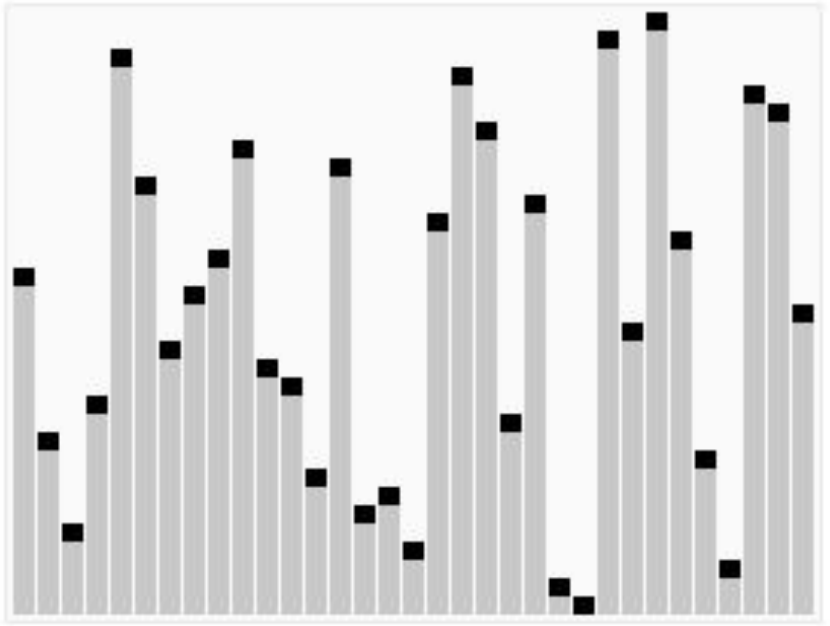
- Отличительной особенностью быстрой сортировки является операция разбиения массива на две части относительно опорного элемента. Например, если последовательность требуется упорядочить по возрастанию, то в левую часть будут помещены все элементы, значения которых меньше значения опорного элемента, а в правую элементы, чьи значения больше или равны опорному.

Алгоритм Сортировка методом Шелла

Алгоритм Сортировка методом Шелла

Таким образом, алгоритм быстрой сортировки включает в себя два основных этапа:

- разбиение массива относительно опорного элемента;
- рекурсивная сортировка каждой части массива.



Алгоритм Сортировка методом Шелла

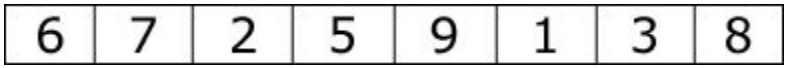
Процесс разбиения массива

- 1. вводятся переменные first и last для обозначения начального и конечного элементов последовательности, а также опорный элемент mid;
- 2. по формуле (first+last)/2 вычисляется значение среднего элемента, и заноситься в переменную mid;
- 3. значения элементов поочередно проверяются на их соответствие позиции, в которой они находятся, то есть элементы левой и правой частей сравниваются с опорным элементом, и если оказывается, что некоторые элементы Mas[i] (левее опорного) и Mas[j] (правее опорного) стоят не на своих позициях, то они меняются местами.
- 4. шаг 3 продолжает выполняться до тех пор, пока все элементы не встанут на свои места относительно опорного элемента (это ни в коем случае не означает, что по окончанию обменов массив будет отсортирован).

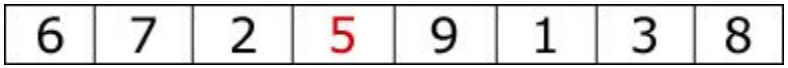
Алгоритм Сортировка методом Шелла

Процесс разбиения массива

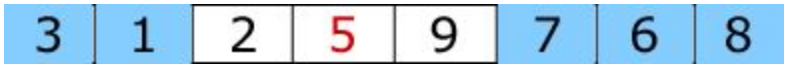
1. Исходный массив, (назовем его Mas) состоит из 8 элементов: Mas[1..8]. Начальным значением first будет 1, а last – 8.



2. В качестве опорного возьмем элемент со значением 5, и индексом 4. Его мы вычислили по формуле (first+last)/2, отбросив дробную Теперь mid=5.



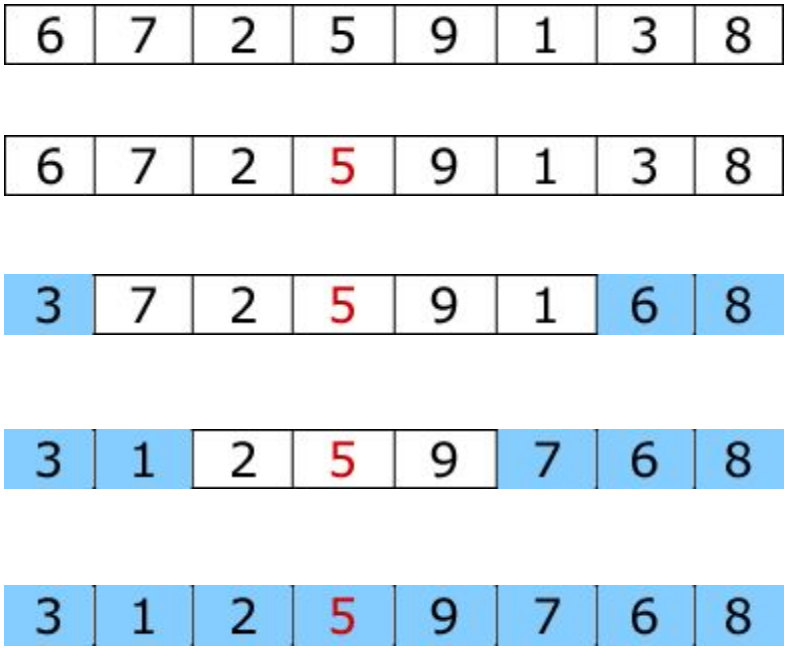
3. Далее, первый элемент левой части сравнивается с mid: 3 меньше 5, поэтому он остается на своем месте, а first увеличивается на 1. Правая часть проверяется с конца. Значения первых двух элементов (они имеют индексы 7 и 8) больше значения опорного элемента, но для третьего это не верно, т. к. 1<5. Элемент запоминается, проверка правой части на некоторое время приостанавливается. Сейчас last равен 6.



Алгоритм Сортировка методом Шелла

Процесс разбиения массива

- 4. На следующем шаге происходит перестановка. Как мы помним, first=2, а last=6, следовательно, элементами, требующими рокировки, являются Mas[2] и Mas[6].
- Тем же способом продолжим проверку оставшихся частей массива до тех пор, пока условие first<last истинно.
- 5. На этом этап разбиения закончен: массив разделен на две части относительно опорного элемента. Осталось произвести рекурсивное упорядочивание его частей.



Рекурсивное доупорядочивание

- Если в какой-то из получившихся в результате разбиения массива частей находиться больше одного элемента, то следует произвести рекурсивное упорядочивание этой части, то есть выполнить над ней операцию разбиения описанную выше. Для проверки условия «количество элементов > 1», нужно действовать примерно по следующей схеме:

Алгоритм Сортировка методом Шелла

Рекурсивное доупорядочивание

- Имеется массив Mas[L..R], где L и R – индексы крайних элементов этого массива. По окончанию разбиения, указатели first и last оказались примерно в середине последовательности, тем самым образую два отрезка: левый от L до last и правый от first до R. Проверить такие отрезки на заданное условие не составит труда.

Многие из вас знакомы с достоинствами программиста. Их всего три, и разумеется это:
лень, нетерпеливость и гордыня.

— *Larry Wall*

Работа с текстовыми файлами в Си++

Я пишу, пишу, пишу – все с клавиатуры…

Работа с текстовыми файлами в Си++

Файлы позволяют пользователю считывать большие объемы данных непосредственно с диска, не вводя их с клавиатуры. Существуют два основных типа файлов: **текстовые и двоичные**.

Работа с текстовыми файлами в Си++

Текстовыми называются файлы, состоящие из любых символов. Они организуются по строкам, каждая из которых заканчивается символом «*конца строки*». Конец самого файла обозначается символом «*конца файла*». При записи информации в текстовый файл, просмотреть который можно с помощью любого текстового редактора, все данные преобразуются к символьному типу и хранятся в символьном виде.

Работа с текстовыми файлами в Си++

В *двоичных* файлах информация считывается и записывается в виде блоков определенного размера, в которых могут храниться данные любого вида и структуры

Работа с текстовыми файлами в Си++

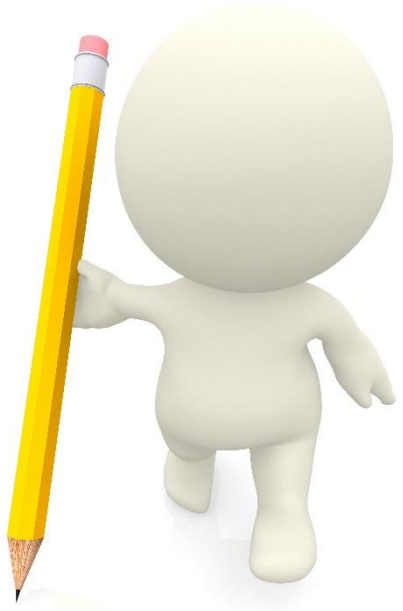
Для работы с файлами используются специальные типы данных, называемые *потоками.* Поток **ifstream** служит для работы с файлами в режиме чтения, а **ofstream** в режиме записи. Для работы с файлами в режиме как записи, так и чтения служит поток **fstream**.

В программах на С++ при работе с текстовыми файлами необходимо подключать библиотеки **iostream** и **fstream**.

Работа с текстовыми файлами в Си++

Для того чтобы записывать данные в текстовый файл, необходимо:

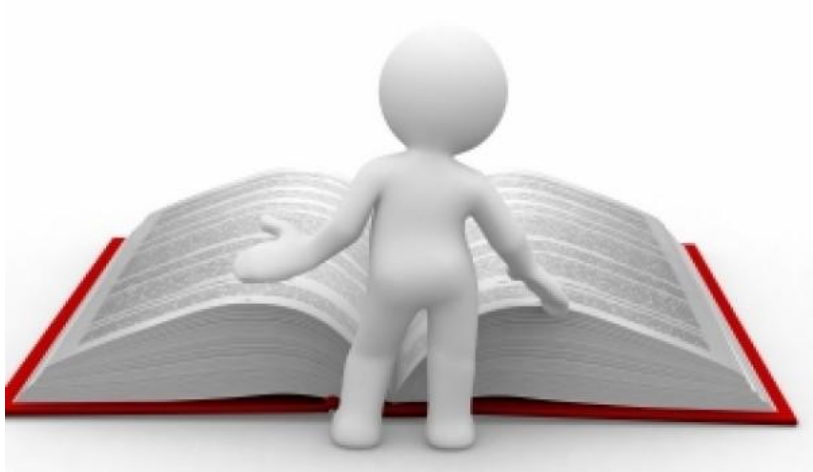
1. описать переменную типа **ofstream**.
2. открыть файл с помощью функции **open**.
3. вывести информацию в файл.
4. обязательно закрыть файл.



Работа с текстовыми файлами в Си++

Для считывания данных из текстового файла, необходимо:

1. описать переменную типа **ifstream**.
2. открыть файл с помощью функции **open**.
3. считать информацию из файла, при считывании каждой порции данных необходимо проверять, достигнут ли конец файла.
4. закрыть файл.



Запись информации в текстовый файл

Как было сказано ранее, для того чтобы начать работать с текстовым файлом, необходимо описать переменную типа **ofstream**. Например, так:

ofstream F;

Будет создана переменная **F** для записи информации в файл. На следующим этапе файл необходимо открыть для записи. В общем случае оператор открытия потока будет иметь вид:

F.open(«file», mode);

Здесь **F** — переменная, описанная как **ofstream**, **file** — полное имя файла на диске, **mode** — режим работы с открываемым файлом. Обратите внимание на то, что при указании полного имени файла нужно ставить двойной слеш. Для обращения, например к файлу **accounts.txt,** находящемуся в папке **sites** на диске **D**, в программе необходимо указать: **D:\\sites\\accounts.txt**.

Запись информации в текстовый файл

Файл может быть открыт в одном из следующих режимов:

- *ios::in* — открыть файл в режиме чтения данных; режим является режимом по умолчанию для потоков **ifstream**;
- *ios::out* — открыть файл в режиме записи данных (при этом информация о существующем файле уничтожается); режим является режимом по умолчанию для потоков **ofstream**;
- *ios::app* — открыть файл в режиме записи данных в конец файла;
- *ios::ate* — передвинуться в конец уже открытого файла;
- *ios::trunc* — очистить файл, это же происходит в режиме ios::out;
- *ios::nocreate* — не выполнять операцию открытия файла, если он не существует;
- *ios::noreplace* — не открывать существующий файл.

Запись информации в текстовый файл

- Параметр mode может отсутствовать, в этом случае файл открывается в режиме по умолчанию для данного потока.
- После удачного открытия файла (в любом режиме) в переменной **F** будет храниться **true**, в противном случае **false**. Это позволит проверить корректность операции открытия файла.

Запись информации в текстовый файл

Открыть файл в режиме записи можно одним из следующих способов:

I. ofstream F;
F.open("D:\\game\\noobs.txt");

II. *режим ios::out является режимом по умолчанию для потока ofstream*

ofstream F;
F.open("D:\\sites\\accounts.txt", ios::out);

II.

третий способ объединяет описание переменной и типа потока, открытие файла в одном операторе

ofstream F ("D:\\game\\noobs.txt", ios::out);

Запись информации в текстовый файл

После открытия файла в режиме записи будет создан пустой файл, в который можно будет записывать информацию.

Если необходимо открыть существующий файл в режиме дозаписи, то в качестве режима следует использовать значение **ios::app**.

После открытия файла в режиме записи, в него можно писать точно так же, как и на экран, только вместо стандартного устройства вывода **cout** необходимо указать имя открытого файла.

Например, для записи в поток **F** переменной **a**, оператор вывода будет иметь вид:

F<<a;

Для последовательного вывода в поток **G** переменных **b**, **c**, **d** оператор вывода станет таким:

G<<b<<c<<d;

Закрытие потока осуществляется с помощью оператора:

F.close();

Работа с текстовыми файлами в Си++

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    setlocale (LC_ALL, "RUS");
    int i, n;
    double a;
    ofstream f;
    f.open("D:\\sites\\accounts.txt", ios::out);
    cout<<"n="; cin>>n;
    for (i=0; i<n; i++)
      {
         cout<<"a=";
         cin>>a;
         f<<a<<"\t";
      }
    f.close();
}
```

открываем файл в режиме записи

описывает поток для записи данных в файл

режим ios::out устанавливается по умолчанию

закрытие потока

Чтение информации из текстового файла

Для того чтобы прочитать информацию из текстового файла, необходимо описать переменную типа **ifstream**. После этого нужно открыть файл для чтения с помощью оператора **open**. Если переменную назвать **F**, то первые два оператора будут такими:

```
ifstream F;
F.open("D:\\sites\\accounts.txt", ios::in);
```

После открытия файла в режиме чтения из него можно считывать информацию точно так же, как и с клавиатуры, только вместо **cin** нужно указать имя потока, из которого будет происходить чтение данных.

Чтение информации из текстового файла

Например, для чтения данных из потока **F** в переменную **a**, оператор ввода будет выглядеть так:

F>>a;

Чтение информации из текстового файла

Часто известен лишь тип значений, хранящихся в файле, при этом их количество может быть различным.

Чтобы решить данную проблему необходимо считывать значения из файла поочередно, а перед каждым считыванием проверять, достигнут ли конец файла.

Для этого используется функция **F.eof()**. Здесь **F** - имя потока функция возвращает логическое значение: **true** или **false**, в зависимости от того достигнут ли конец файла.

Чтение информации из текстового файла

Следовательно, цикл для чтения содержимого всего файла можно записать так:

Организуем для чтения значений из файла, выполнение цикла прервется, когда достигнем конец файла, в этом случае F.eof() вернет истину

```
while (!F.eof())
  {
    F>>a;
    …
  }
```

Чтение информации из текстового файла

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
 {
  setlocale (LC_ALL, "RUS");
  int n=0;
  float a;
  fstream F;  //открываем файл в режиме чтения
 F.open("D:\\sites\\accounts.txt");//если открытие файла прошло корректно, то
  if (F)
   {
     while (!F.eof())
       {
        F>>a;
        cout<<a<<"\t";
        n++;
       }
     F.close(); //закрытие потока
     cout<<"n="<<n<<endl;
  }
//если открытие файла прошло некорректно, то вывод сообщения об отсутствии такого файла
   else cout<<" Файл не существует"<<endl;
```