Введение в параллельные вычисления

д.т.н. Мокрова Наталия Владиславовна

асп. Морунов Егор, Сырко Денис

пятница	ауд. 119	
10:45 – 12:20	Лекция	
	1 неделя	2 неделя
12:50 - 16:10	Лаб – КС44	Лаб - КС40

Выбор модели программирования

Системы с общей памятью:
- C++11 threads (pthreads);
- OpenMP.

Системы с распределенной памятью:
- MPI = message passing interface.

Гетерогенные системы (графические процессоры):
- CUDA.

Лекция 2. Библиотека C++11 thread

Лекция основана на переводе примеров:

- https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial/ ;
- сами примеры https://github.com/sol-prog/threads (команда git clone).

Пример потоков

- ..\Пример потока с матрицей_linux.doc.

- http://en.cppreference.com/w/cpp/thread – библиотека потоков.
- http://coliru.stacked-crooked.com/ – компиляция и выполнение онлайн.

Linux и потоки

Linux поддерживает многозадачность и многопоточность, т.е. в системе одновременно может работать несколько задач (процессов), и каждая из задач может выполнятся в несколько потоков.

Поток выполнения – элемент кода программы, выполняемый последовательно.

Большинство приложений – однопоточные программы.

Многопоточная программа в один момент времени может выполняться в нескольких отдельных потоках. В случае, если задача выполняется на многопроцессорной машине, то все ее потоки могут выполняться одновременно, повышая таким образом производительность выполнения задачи.

Производительность многопоточного приложения можно улучшить даже на однопроцессорной системе. Например, если один из потоков приложения блокируется каким-то системным вызовом или ожидает поступления данных, в это время выполняется другой поток.

Стандарт С+11

Одно из самых больших изменений в языке – поддержка многопоточности.

До появления стандарта С+11 многопоточные программы можно было создавать при помощи средств ОС (pthreads на Unix-системах) или при помощи библиотек OpenMP, MPI.

Процесс и поток

Между процессами и потоками существуют различия.

Под процессами понимается программа находящаяся в стадии выполнения. Например, shell – это процесс который создается при входе пользователя в систему.

Каждая команда создает новый процесс. Согласно терминологии UNIX – это порожденный процесс, который выполняет команду от лица пользователя.

Процессы в Unix надежно изолированы друг от друга. Нарушения целостности данных одного процесса (например, в результате переполнения буфера или ошибок при работе с указателями) приводят к аварийному завершению этого процесса, но не затрагивает непосредственно другие процессы.

В современных Unix-системах было введено понятие нитей (*thread*), которые соответствуют единицам планирования в рамках одного процесса.

Потоки – часть процесса, и они используют сегменты данных и кода совместно.

Для многопоточного программирования существует два основных стандарта: многопоточные API Solaris (Sun Microsystems) и API POSIX.1c.

В Linux используется API POSIX.1c., присутствует системный вызов clone(), на основе которого и построено API для работы с потоками, соответствующие стандарту POSIX.1c с незначительными исключениями.

Зачем нужны многопоточные программы

1. Улучшение времени реакции интерактивных программ.
   - Фоновое скачивание страницы в браузере.
   - Фоновый ввод-вывод (считывать файл по мере его просмотра).
   - Фоновая проверка орфографии.
   - Фоновое переразбиение текста на страницы в WYSIWYG текстовых процессорах (В OpenOffice переразбиение текста на страницы осуществляется в фоновом режиме при помощи отдельной нити).
2. Улучшение времени реакции серверных приложений. Возможность обрабатывать несколько запросов одновременно.
3. Использование дополнительных ресурсов на многопроцессорных и гипертрединговых компьютерах.
4. Задачи реального времени.

Проблемы многопоточности

Попытки организации параллельных вычислений – 60-е годы XX столетия.

Массовое распространение многопоточного *программирования* – 90-е годы.

Технические ограничения многопоточного программирования:

- несовместимость со однопоточными компиляторами;
- несовместимость со старыми библиотеками;
- несовместимость или ограниченная поддержка инструментальными средствами, например, отладчиками.

Ограничение – несовместимость многих принятых практик программирования с многопоточностью.

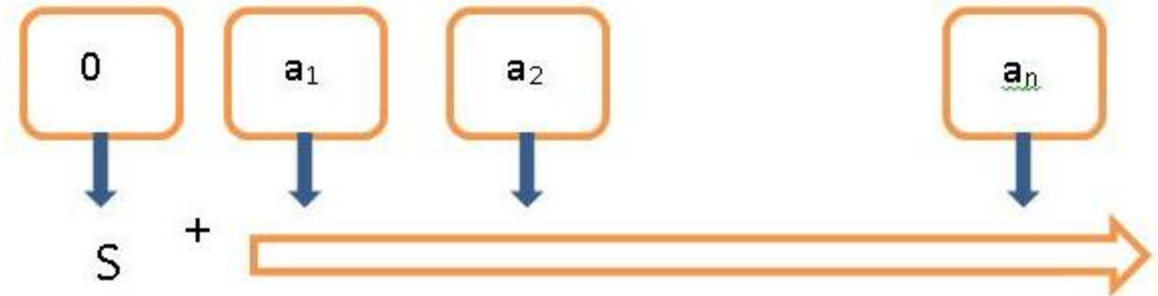
Использование специальных конструкций языка – Parallel Fortran получил широкое практическое применение.

При программировании на C / C++ с использованием *POSIX Thread Library* специальные компиляторы не требуются.

Все, необходимое для поддержки многопоточности, реализовано на уровне библиотек.

Свойства параллельных вычислений

Схема последовательного суммирования



Вычисление суммы можно распараллелить, построив пирамидальный алгоритм, но для него на первом шаге потребуется n/2 процессоров, которые будут вычислять суммы пар элементов массива. На следующем шаге, число процессоров сократится вдвое, когда вычисляются суммы четверок. Для получения окончательного результата на последнем шаге потребуется всего один процессор.

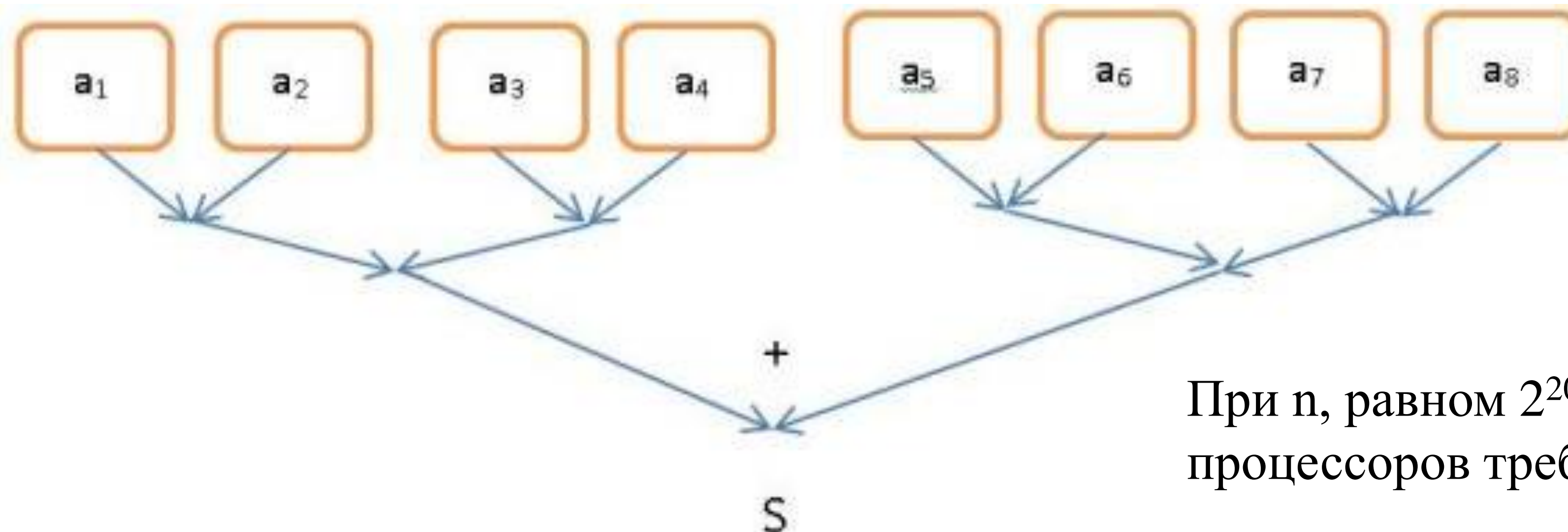


Схема параллельного вычисления суммы

При n, равном 2^{20}, ускорение более 2^{15}, но и процессоров требуется столько же.

Пример 1 – Создание потока

В примерах сайта solarianprogrammer.com также описываются некоторые из новейших особенностей C++11: регулярные выражения, “сырые” строки, и лямбда-функции.

Рассмотрим программу HelloWorld с потоками.

Компиляция при помощи g++

g++ -std=c++11 -pthread file_name.cpp

В настоящих приложениях функция “call_from_thread” будет выполнять какую-либо работу независимо от функции main.

В примере функция main создает поток, и ждет, пока он завершится в точке t1.join().

Если не подождать дочерний поток, возможна ситуация, когда функция main завершится первой, и вся программа закончит свою работу, удалив запущенный поток вне зависимости от того завершилась ли функция “call_from_thread”, или нет.

Пример 1

```
#include <iostream>
#include <thread>

//This function will be called from a thread

void call_from_thread() {
    std::cout << "Hello, World" << std::endl;
}

int main() {
    //Launch a thread
    std::thread t1(call_from_thread);

    //Join the thread with the main thread
    t1.join();

    return 0;
}
```

В новом стандарте C++11 многопоточность осуществлена в классе thread, который определен в файле thread.h.

Для того чтобы создать новый поток нужно создать объект класса thread и инициализировать, передав в конструктор имя функции, которая должна выполнятся в потоке.

Метод join синхронизирует потоки и возвращает выполнение программе, когда поток заканчивается, после чего объект класса thread можно безопасно уничтожить.

Пример 2 – несколько потоков

Обычно необходимо запустить несколько потоков, чтобы сделать какую-то работу параллельно. Для этого нужно создать массив потоков.

В примере 2 в функции main создается массив из 10 потоков и функция ожидает завершения их работы.

Помните, что функция main – тоже поток, который обычно называют главным потоком, так что этот код запускает не 10, а 11 потоков. Это позволяет проделать некоторые вычисления в функции main после запуска дочерних потоков и той точки, где она будет ожидать их завершения.

Пример 2

```
1    ...
2    static const int num_threads = 10;
3    ...
4    int main() {
5        std::thread t[num_threads];
6
7        //Launch a group of threads
8        for (int i = 0; i < num_threads; ++i) {
9            t[i] = std::thread(call_from_thread);
10       }
11
12       std::cout << "Launched from the main\n";
13
14       //Join the threads with the main thread
15       for (int i = 0; i < num_threads; ++i) {
16           t[i].join();
17       }
18
19       return 0;
20   }
```

main создает группу из 10 потоков.

Всего потоков – 11.

О выводе в параллельном коде

<< – конкурент в “гонке потоков”.

```
3   using namespace std;
4   static const int num_threads = 10;
5   void call_from_thread() {
6       cout << "Hello, World"<< endl;
7   }
8   int main() {
9       thread t[num_threads];
10      for (int i = 0; i < num_threads; ++i) {
11          t[i] = std::thread(call_from_thread);
12      }
13      cout << "Launched from the main\n";
14      for (int i = 0; i < num_threads; ++i) {
15          t[i].join();
16      }
17      return 0;
18  }
```

Работа потоков

```
Hello, World
Hello, World
Hello, World
Launched from the main
Hello, World
Hello, World
Hello, World
Hello, World
Hello, WorldHello, World

Hello, World
```

Пример 2

```
Hello, World
Hello, World
Hello, World
Launched from the main
Hello, World
Hello, World
Hello, World
Hello, World
Hello, World
Hello, WorldHello, World
```

Пример 3

```
Hello, WorldHello, WorldHello, WorldLaunched from the main
Hello, World4Hello, World
Hello, World6
Hello, World7
1
0
Hello, World9
8
2
Hello, World5
Hello, World3
```

Сравним программы

```
3   using namespace std;
4   static const int num_threads = 10;
5   void call_from_thread(int tt) {
6       for (int i = 0; i < num_threads; ++i)
7       cout << tt ;
8       cout <<endl;
9   }
10  int main() {
11      cout << " Main thread " << endl;
12      thread t[num_threads];
13      for (int i = 0; i < num_threads; ++i) {
14          t[i] = std::thread(call_from_thread, i);
15          t[i].join();
16      }
17      return 0;
18  }
```

```
Main thread
0000000000
2151151515151111
3333333333
42444444444
22222222
6666666666
55555
7777777777
8888888888
9999999999
```

```
using namespace std;
static const int num_threads = 10;
void call_from_thread(int tt) {
    for (int i = 0; i < num_threads; ++i)
    cout << tt ; cout <<endl;
}
int main() {
    cout << " Main thread " << endl;
    thread t[num_threads];
    for (int i = 0; i < num_threads; ++i) {
        t[i] = std::thread(call_from_thread, i);
    }
    for (int i = 0; i < num_threads; ++i)
        t[i].join();
    return 0;
}
```

```
Main thread
0000000000
1111111111
2222222222
3333333333
4444444444
5555555555
6666666666
7777777777
8888888888
9999999999
```

?

Где реально многопоточность?

Пример 3 – потоковая функция с параметрами

C++ позволяет создать потоковую функцию с любым количеством параметров. Добавим к примеру 2 целочисленный параметр (int).

При выполнении программы видно, что запущенные потоки выполняются в случайном порядке (Работа потоков рис. 3).

Это работа программиста – проследить, чтобы потоки не блокировали друг друга, пытаясь получить доступ к одним и тем же данным.

Также видно, что вывод из разных потоков смешивается – потому что все 11 потоков пытаются получить доступ к одному ресурсу – stdout потоку вывода (см. Работа потоков).

Пример 3

```
1   #include <iostream>
2   #include <thread>
3
4   static const int num_threads = 10;
5
6   //This function will be called from a thread
7
8   void call_from_thread(int tid) {
9       std::cout << "Launched by thread " << tid << std::endl;
10  }
11
12  int main() {
13      std::thread t[num_threads];
14
15      //Launch a group of threads
16      for (int i = 0; i < num_threads; ++i) {
17          t[i] = std::thread(call_from_thread, i);
18      }
19
20      std::cout << "Launched from the main\n";
21
22      //Join the threads with the main thread
23      for (int i = 0; i < num_threads; ++i) {
24          t[i].join();
25      }
26
27      return 0;
28  }
```

Некоторые выводы

- программист должен гарантировать, что группа потоков не будет блокировать и пытаться изменить одни и те же данные;
- все 11 нитей данной программы конкурируют за один и тот же ресурс, которым является stdout;
- можно избежать некоторых из вышеперечисленных проблем с использованием барьеров в коде (std::mutex);
- можно писать более сложные параллельные коды, используя только вышеописанный синтаксис (см. пример 4).

Пример 4 – удаление шума из изображения

Удаление шума из изображения при помощи фильтра blur.

Идея – можно размыть шум, если взять среднее взвешенное значение пикселя и его соседей.

Для пространственной свертки в частотной области использовано преобразование Фурье.

Использованы файлы формата ppm – без сжатия, заголовочный файл класса, который позволяет хранить изображения (ppm.h, ppm.cpp). (см. https://solarianprogrammer.com/2011/12/16/cpp-11-thread-tutorial/)

Как организован код?

- загрузить изображение в память ();
- разделить изображение между потоками (будет 8 потоков);
- запустить 7 потоков (N – 1), каждому – свой кусок картинки;
- главному потоку оставить последний кусок;
- подождать, пока все потоки завершат свою работу;
- сохранить изображение.

Фрагмент программы

```
//Lauch parts-1 threads
for (int i = 0; i < parts - 1; ++i) {
    tt[i] = std::thread(tst, &image, &image2, bnd[i], bnd[i + 1]);
}

//Use the main thread to do part of the work !!!
for (int i = parts - 1; i < parts; ++i) {
    tst(&image, &image2, bnd[i], bnd[i + 1]);
}

//Join parts-1 threads
for (int i = 0; i < parts - 1; ++i)
    tt[i].join();
```

Пример 4

Компиляция:

g++ -std=c++11 -pthread -o image.bin *.cpp

(в папке файлы ppm.cpp и main.cpp)

Чтобы увидеть эффект от распараллеливания, нужно создать значительный объём работы, иначе накладные расходы на создание потоков обесценивают усилия на распараллеливание.

Входное изображение должно быть достаточно большим, чтобы увидеть ускорение за счёт распараллеливания.

Использовано изображение 16000x10626 пикселов, которое занимает около
512 MB в формате PPM.

Шум добавлен при помощи GIMP.

Замерить время (1, 2, 4, 8) потоков. Выбрать наилучшее. Замерить время с оптимизацией -O4 при компиляции, и без.

Результат работы программы

Результаты выполнения примера 4 на двухъядерном ноутбуке MacBook Pro от 2010 г.

Compiler	Optim	Threads	Time	Speed
clang++	none	1	40 s	
clang++	none	4	20 s	2x
clang++	-O4	1	12 s	
clang++	-O4	4	6 s	2x

На компьютере с двухъядерным процессором этот код имеет лучшую скорость при использовании параллельно работающих потоков (в 2 раза) по сравнению с исполнением кода в последовательном режиме (один поток).

Результат работы программы

Результаты выполнения на четырехъядерном процессоре Intel i7 с Linux.

Compiler	Optim	Threads	Time	Speed
g++	none	1	33 s	
g++	none	8	13 s	2.54x
g++	-O4	1	9 s	
g++	-O4	8	3 s	3x

В данном случае Apple clang++ лучше при масштабировании параллельной программы, это может объясняться сочетанием компилятора и характеристик компьютера, а также потому что в использованном для проведения тестов MacBook Pro 8 Гб оперативной памяти, а на Linux-машине – 6 Гб.

Hello mas

- Случайные числа

```
a[i][j] = rand() % 10 + 1;
```

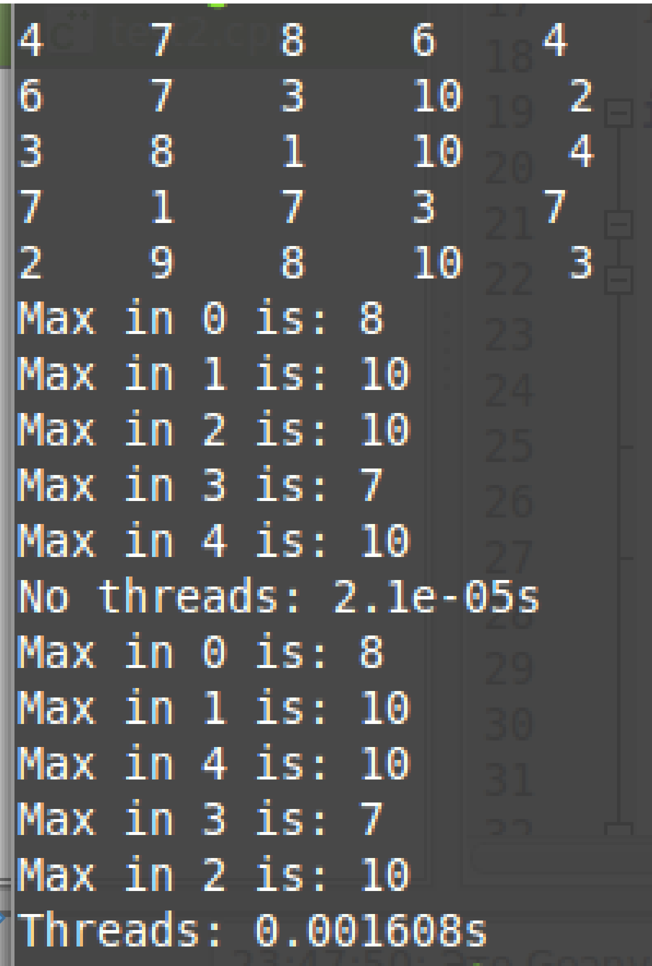
- Время работы

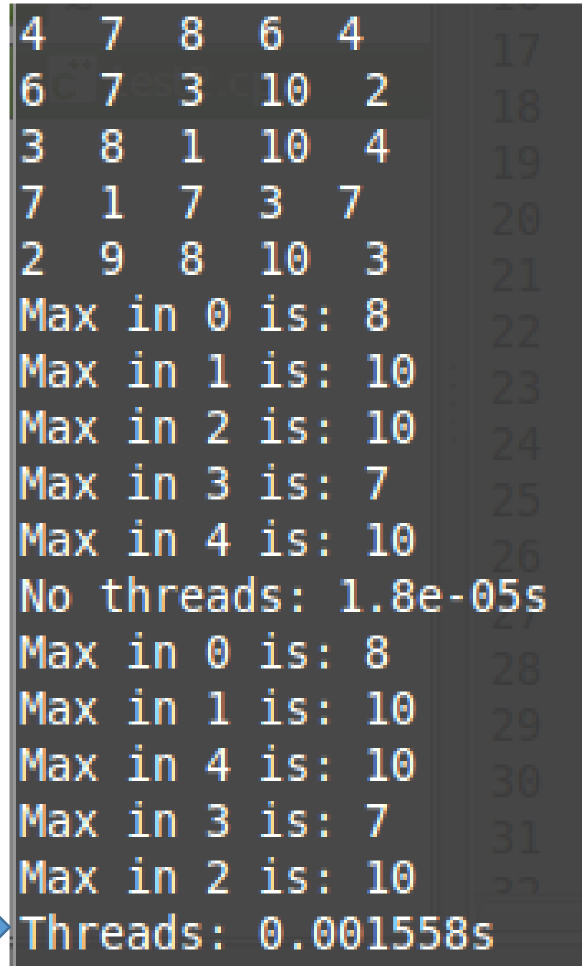
```
clock_t start, end;

start = clock();
```

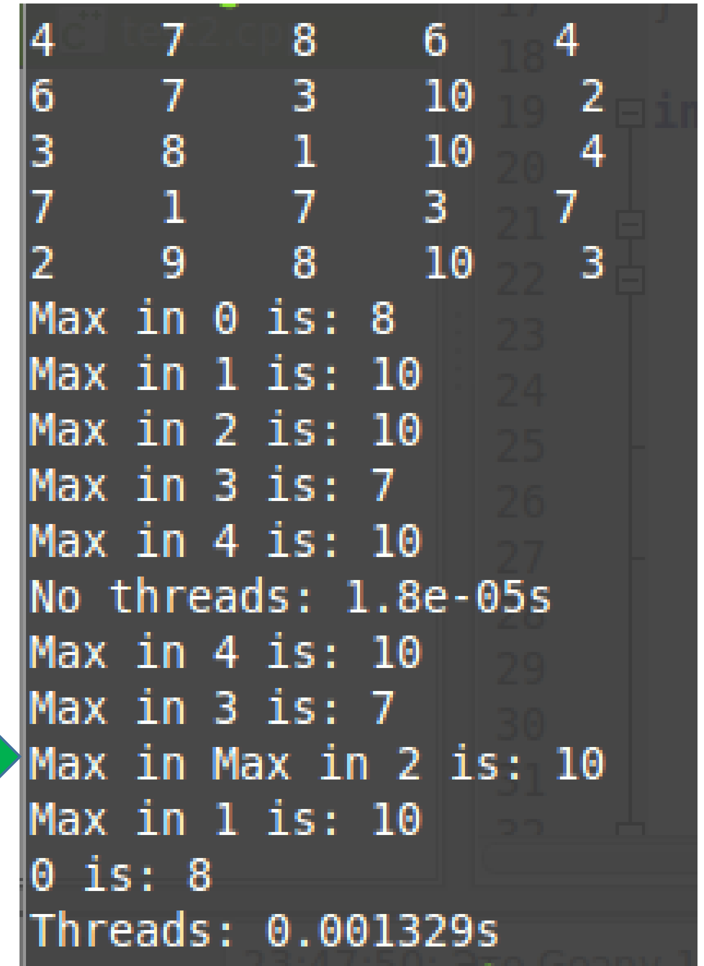
- Потоки

```
std::thread t[N];

start = clock();
for (int i=0; i<N; i++) {
    t[i] = std::thread(max_thread, i);
}
for (int i=0; i<N; i++) {
    t[i].join();
}
end = clock();
```

Результат работы потоков







Представление результатов

```
3 8 999 10 4
6 999 3 10 2
 2 9 9998  7 8 610  4999

7 1 7 999 7
```

Это массив?

Он выведен в потоке

```
999      7      8      6      4
  2      9      8     10    999
  7      1      7    999      7
  6    999      3     10      2
  3      8    999     10      4
```

```
#include <iomanip>
using namespace std;
```

```
<< setw(7)
```