Методичка по **C++** (v1.3)

R314 & TL

3 декабря 2017 г.

Оглавление

Глава 1

Ввод, вывод и переменные

1.1 Простейшая программа

1.1.1 Компиляторы

Существует множество компиляторов для языка программирования C++. Компилятор — это программа, выполняющая трансляцию программы на языке высокого уровня в аналогичную программу на низкоуровневом языке, близком к машинному коду. Получающийся в результате компиляции файл называют исполняемым файлом.
Для языка C++ существует множество различных компиляторов, каждый из которых поддерживает или не поддерживает те или иные конструкции языка, и эти различия следует учитывать. Самыми популярными компиляторами C++ в олимпиадном программировании являются G++ (от GNU) и Visual C++ (от Microsoft). Зачастую может случаться так, что код, корректный для одного компилятора, некорректен для другого. В методичке ниже будут указываться различия в версиях языка для этих компиляторов.

1.1.2 Пустой код и компиляция

Программа на C++ вначале должна компилироваться в исполняемый файл, который впоследствии должен запускаться.
Первое, что нужно сделать — это подключить необходимые *библиотеки*, в которых описаны те или иные функции. Библиотеки — уже готовый код "из коробки" с реализованными функциями, структурами данных, методами и т.п. В GNU G++11 5.1.0 есть библиотека "`bits/stdc++.h`", включающая в себя почти всё, что нужно в олимпиадном программировании. В Visual C++ такой библиотеки нет, поэтому придётся подключать нужные библиотеки отдельно и по мере необходимости. Библиотеки подключаются первой строкой в программе.
Изначально всё помещено в так называемую глобальную область. Тут можно объявлять переменные (в том числе и массивы), создавать свои типы данных и объявлять свои функции. Основные действия можно выполнять только в функциях. Выполнять инструкции программа начнёт с главной функции `main`, которая **всегда** должна быть объявлена в программе.

Рассмотрим, как выглядят функции в C++. У них есть 4 важных атрибута:

- название;
- тип возвращаемого значения (если это процедура, то в качестве типа выступает ‘void’);
- аргументы (могут отсутствовать);
- тело функции.

Рассмотрим, как описать функцию `main`, в которой нет инструкций.

```
#include<bits/stdc++.h>

int main()
{
    return 0;
}
```

В приведённом коде объявлена функция `main`, которая возвращает `int`, то есть знаковое 32-битное число (см. ниже Типы данных), параметры могут отсутствовать. При успешном завершении она должна всегда возвращать ноль. Неуспешное завершение происходит, когда программа обратилась не к своей памяти, сделала что то неопределённое (поделила на ноль) и т.п. Таким образом, `main` возвращает так называемый код ошибки. Он должен быть равен нулю, если всё хорошо.
Обратите внимание на следующие важные детали: тело функции пишется не только с отступом (форматирование кода), но и заключается в фигурные скобки. Табуляция формально не требуется, но без неё код, как правило, бывает неразборчив для человека. В конце каждой команды ставится точка с запятой, кроме строчек с условиями, циклами и объявлением функций.
Давайте выполним данный код, то есть скомпилируем и запустим исполняемый файл. Если мы будем использовать компилятор GCC, это будет выглядеть примерно так:

Консоль

```
> g++ -O2 -Wall -std=gnu++11 program.cpp -o program.exe
> program.exe
```

Первая строка компилирует код из файла program.cpp с помощью оптимизаций кода (ключ `-O2`), с выводом помимо ошибок ещё и предупреждений о возможном непредвиденном поведении программы (ключ `-Wall`), в исполняемый файл program.exe. Ключ `-std=gnu++11` говорит компилятору использовать стандарт языка C++11. Впоследствии мы не будем писать строку компиляции в примерах, подразумевая, что она выполнена.

1.1.3 Привет, мир!

Рассмотрим простую программу. Помимо подключения библиотек в коде также написано, какое пространство имён использует программа. Что именно без этого не будет

работать, и зачем это нужно, мы разберём позже. Следующие три версии кода делают одно и тоже, а именно выводят на экран строку “Hello, World!”.

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    cout << "Hello, World!\n";
    return 0;
}
```

```
#include<bits/stdc++.h>
using namespace std;
int main()
{
    puts("Hello, World!");
    return 0;
}
```

Функция `printf()` выводит на экран свой аргумент (то, что дано в скобках). У этой функции есть ряд особенностей, о которых мы поговорим позже. Это быстрая функция вывода. Обратите внимание, что строки в C++ заключаются строго в двойные кавычки, а не в одинарные. В одинарные кавычки в C++ заключаются только отдельные символы.

Функция `cout` выводит на экран то, что стоит после двух уголков влево. Эта функция работает медленнее, чем `printf()`, но её можно ускорить.

Функция `puts()` выводит на экран свой аргумент (то, что дано в скобках). Эта функция работает только со *строками-как-в-C*, то есть массивами символов со специальным завершающим символом, но это очень быстрая функция.

Существует также функция `putchar()`, которая выполняет то же, что и `puts()`, но с отдельными символами.

Функции `printf()`, `puts()` и `putchar()` называют функциями стандартного ввода, они унаследованы языком C++ от языка C. Функция `cout` — функция потокового ввода.

```
> program.exe
Hello, World!
```

1.2 Типы данных

Информация в памяти компьютера хранится в виде последовательности битов — простейших единиц памяти, которые могут хранить только два значения: 0 или 1. Любая информация кодируется в виде битового кода, который интрепретируется компьютером, как числа в двоичной системе счисления. По сути, любая информация, записанная в памяти компьютера — это число.
Различия между типами данных — это различия между способами записи и чтения различных значений в памяти.

1.2.1 Числа

Типы данных для хранения чисел можно поделить на целочисленные и "с плавающей точкой".

Целые числа

Самый простой пример хранения информации в памяти — хранение *беззнаковых* (неотрицательных) целых чисел. В памяти отводится определённое количество битов, которое используется для записи целого числа в двоичной системе счисления. Например, если бы мы использовали тип данных, занимающий 1 байт (8 бит) памяти, то число 152 внутри памяти можно было бы представить как последовательность битов `10011000`.
Диапазон значений чисел, которые можно хранить в памяти, напрямую зависит от отводимого для хранения объёма. Если для записи отводится n битов, то в них можно хранить числа от 0 до $2^n - 1$. Если попробовать хранить в этих типах данных числа вне диапазона, компьютер обрежет данные. Если к восьмибитному числу `11111111` прибавить двойку, выйдет восьмибитное число `00000001`, так как при сложении двух значений произошло *переполнение типа данных*: результатом сложения стало девятибитное число `100000001`.
Переполнение восьмибитного типа данных стало причиной одного из самых известных багов в истории компьютерных игр, когда в игре Цивилизация лидер-пацифист Индии Махатма Ганди с уровнем враждебности 1 в процессе игры терял ещё два пункта враждебности и становился самым агрессивным правителем с уровнем враждебности 255 ($2^8 - 1$).
Если необходимо хранить *знаковые числа*, то есть числа, которые могут быть как положительными, так и отрицательными, из n битов, отводимых в памяти под запись числа, один бит отводится под хранение знака числа, оставляя для хранения значения оставшиеся $n - 1$ бит. В итоге, диапазон значений для знаковых чисел уменьшается *по модулю* и становится диапазоном от -2^{n-2} до $2^{n-2} - 1$ (при этом *количество* различных хранимых значений сохраняется!).

Числа с плавающей точкой

(Также известные, как числа с плавающей запятой)
Эти типы данных созданы для хранения вещественных (действительных) чисел. Название "с плавающей точкой" описывает принцип хранения вещественных чисел в памяти,

который можно проиллюстрировать следующим образом. Например, мы хотим записать на бумаге десятичный вариант дроби $\frac{5}{4} = 1.25$. Сначала мы запишем цифры `125`, а затем поставим точку в нужном месте — после первой *значащей цифры*. Таким же образом хранятся вещественные числа в памяти компьютера: отдельно хранятся значащие цифры числа (мантисса), отдельно — позиция точки в этом числе (показатель степени). Именно из-за отдельного кодирования позиции точки их называют числами с плавающей точкой.
Ограничения мантиссы и показателя степени, которые в отдельности являются целыми числами, точно так же, как в целочисленных типах данных, напрямую зависят от отводимых под их хранение объёмов памяти.
Стоит помнить, что в знаковых типах данных дополнительный бит отводится под хранение знака числа!

1.2.2 Символы

Многим знакомо слово "кодировка". Его смысл — как раз формат хранения данных о символах в компьютерной памяти. В общем случае, каждый символ в кодировке сопоставляется определённому числу (коду).
ASCII — классический восьмибитный стандарт кодирования. Впоследствии был создан Юникод, плюсом которого является то, что он содержит все возможные символы для языков мира и не только, а минусом — больший объём требуемой для хранения памяти.

1.2.3 Логические значения

Логические (булевские) значения — это значения, описывающие истинность или ложность какого-либо утверждения. То, что $5 < 9$ — это *истина*, а $5 > 9$ — это *ложь*. Логические типы данных должны хранить всего два различных значения, как один бит: 1 для истины, 0 для лжи.
На деле это не совсем так. Как правило, логический тип данных действительно хранит 0 для лжи, но для истины он может хранить любое ненулевое значение, и в итоге тип данных занимает больше, чем один бит. На самом деле, как правило, объём памяти логической переменной определяется размером минимальной адресуемой ячейки в памяти — это байт, то есть 8 бит. Таким образом с этим типом данных удобнее работать процессору и оперативной памяти.

1.2.4 Типы данных в C++

Модификатор `unsigned`, который пишется перед типом данных через пробел, означает, что это — беззнаковое значение (по умолчанию они знаковые). Знаковым и беззнаковыми в C++ могут быть только целые числа.

Тип данных	Память
Целочисленные	
`__int8`, `char`	1 байт
`__int32`, `int`	4 байта *[1]*
`__int64`, `long long`	8 байт
`__int128`	16 байт *[2]*
С плавающей точкой	
`float`	4 байта
`double`	8 байт
`long double` в GNU C++	80 бит
Символьный	
`char`	1 байт
Логический	
`bool`	1 байт

1

— в старых стандартах 32-битные целые числа — это ***`long int`****, а* ***`int`*** *— 16-битное число.*

2

— далеко не все операции реализованы для ***`__int128`*** *из-за того, что не существует пока 128-битных процессоров.*

1.3 Условные операторы и циклы

1.3.1 Условный оператор

Алгоритмическая конструкция "если *условие*, то *делай действия*, иначе *делай другие действия*" описывается в C++ с помощью `if-else`. Условие в условном операторе пишется в скобках, и в случае, если условие — истина (или не ноль), выполняется действие, указанное сразу после оператора. Опционально можно после действия для `if` добавить команду `else`, в таком случае если условие в условном операторе — ложь (или ноль), выполнится действие, указанное сразу после `else`.
В качестве условия можно использовать несколько выражений, объединённых логическими операторами. Оператор `&&` (логическое "И") даёт истину только тогда, когда применяется к двум выражениям, также дающим истину. Оператор `||` (логическое "ИЛИ") даёт ложь только тогда, когда применяется к двум выражениям, также дающим ложь. Оператор `!` (логическое "НЕ") меняет истину на ложь, и наоборот.

Пример: проверить, что число a входит в диапазон значений [5, 10]:

```
if(a >= 5 && a <= 10)
    printf("YES");
else
    printf("NO");
```

```
if(a < 5 || a > 10)
    printf("NO");
else
    printf("YES");
```

1.3.2 Цикл с предусловием

Если команда или команды в условном операторе должны выполняться не один раз, а много раз, пока условие возвращает истину, примеряют оператор `while`. Проверив условие на истинность, оператор выполнит команду после `while`, а затем снова проверит условие, и снова выполнит команду, и так далее до тех пор, пока условие не станет ложным.

Пусть надо вывести числа то 1 до 10 (см. следующую секцию Форматированный вывод):

```
int i = 1;
while(i <= 10) {
    printf("%d ", i);
    i++;
}
```

1.3.3 Цикл со счётчиком

Основанный на идее цикла с предусловием, оператор `for` в C++ помимо условия выполнения цикла имеет также пространство, где указываются операции, которые должны выполниться прямо **перед началом** выполнения цикла, а также операции, которые выполняются **сразу после каждой итерации, но перед проверкой условия**. Синтаксис оператора такой: `for (`*начальные значения* `;` *условие выполнения* `;` *операции после итераций*`)`.

Тот же самый код из предыдущегоп примера, где выводились числа от 1 до 10, можно записать с помощью цикла `for` в две строки:

```
for(i = 1; i <= 10; i++) {
    printf("%d ", i);
}
```

1.3.4 Перебор элементов множества

Новый стандарт языка C++11 добавил синтаксис для цикла `for`, позволяющий кратко записать перебор элементов структур данных из стандартной библиотеки шаблонов C++, таких как вектор, сет, мультисет, мап, дек (см. Главу 3). В этом варианте синтаксис цикла выглядит так: `for (auto` *новая переменная* `:` *множество*`)`.
Таким образом, вывод элементов сета может выглядеть не так:

```
for(auto it = s.begin(); it != s.end(); it++) {
    a = *it;
    printf("%d ", a);
}
```

А так:

```
for(auto a : s) {
    printf("%d ", a);
}
```

1.3.5 Цикл с постусловием

Если требуется сначала провести некоторые действия, а только потом проверять условие, чтобы их повторять, можно использовать оператор `do while`. Кроме момента с указанием условия выполнения в конце, принцип работы этого оператора не отличается от цикла с предусловием (в отличие от языка Pascal, где цикл с постусловием использует условие выхода их цикла вместо условия продолжения).
Данный код также выводит числа от 1 до 10:

```
int i = 1;
do {
    printf("%d ", i);
    i++;
} while(i <= 10);
```

1.3.6 Прерывание цикла

Особые команды `break` и `continue` позволяют прервать ход выполнения цикла изнутри цикла. Команда `break` моментально завершает выполняющийся цикл (только один!), а команда `continue` прерывает текущую итерацию и переходит к следующей (но в цикле `for` сначала выполняются `операции после итераций`).
Эти операции полезны, когда условием выполнения цикла нельзя полностью описать всю логику алгоритма. Они применяются, например, когда неизвестно, в какой момент внутри цикла выполнится нужное условие.

Пусть надо возводить в квадрат вводимые числа до тех пор, пока на ввод не будет дан ноль:

```
while(true) {
    cin >> a;
    if(a == 0)
        break;
    cout << a*a << ' ';
}
```

1.4 Форматированный вывод

При выводе строки функцией `printf()` можно подставлять в неё значения. В том месте, где в выводимую строку нужно вставить нужное значение, пишется оператор `%` и так называемый спецификатор (для целых чисел это `d`). После самой строки ставится запятая, а потом через запятую перечисляются все подставляемые значения **в том же порядке, в котором они должны быть подставлены в строку**. В отличие от Python, эти значения не заключаются в скобки.

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    printf ("%d + %d = %d\n",  3, 5, 3+5);
    printf ("And %d * %d = %d\n", 3, 5, 3*5);
    return 0;
}
```

Выполнение программы

```
> program.exe
3 + 5 = 8
And 3 * 5 = 15
```

Список некоторых спецификаторов для различных типов данных:

`d`	Знаковое целое десятичное число
`I64d`	Знаковое целое десятичное длинное (64-битное) число в Windows
`lld`	Знаковое целое десятичное длинное (64-битное) число в Unix
`o`	Знаковое целое восьмеричное число
`x, X`	Знаковое шестнадцатеричное число
`f`	Вещественное число типа данных float
`lf`	Вещественное число типа данных double
`c`	Символьная переменная
`s`	Строка

Пример использования различных спецификаторов:

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    printf ("My name is %s.\nI am %i and love the number
                        %12.4f\n", "Ruslan", 22, 3.1415926);
    printf ("My name is %s.\nI am %i and love the number
                        %2.10f\n", "Ruslan", 22, 3.1415926);
    return 0;
}
```

(Здесь и далее перенос строки в коде — лишь для того, чтобы код влезал в страницу. На самом деле нельзя так разрывать код, за исключением некоторых случаев.)

Выполнение программы:

```
> program.exe
My name is Ruslan.
I am 22 and love the number       3.1416
My name is Ruslan.
I am 22 and love the number 3.1415926000
```

Отдельное внимание стоит уделить выводу вещественных чисел. В этом примере между `%` и спецификатором `f` стоят дополнительные параметры. Число до точки — это минимальное количество символов, которое должно занимать выводимое число целиком. “Лишние” символы забиваются пробелами. Число после точки — это количество знаков после запятой, которое должно быть у выводимого числа. Этот параметр не просто

обрезает число, а округляет его по всем правилам (поэтому в выводе число `3.1416`, а не `3.1415`).

Эти параметры необязательны для использования, в таком случае спецификатор будет выглядеть просто как `%f`. Если же используется только один из параметров, нужно обязательно ставить точку: `%.2f`, `%5.f`.

Всё сказанное выше применимо и к спецификатору `lf`.

Вместо известных значений в качестве параметров в функцию `print()` можно подставлять переменные:

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    int a = 3, b = 5;
    printf ("%d + %d = %d\n", a, b, a+b);
    printf ("And %d * %d = %d\n", a, b, a*b);
    return 0;
}
```

Выполнение программы:

```
> program.exe
3 + 5 = 8
And 3 * 5 = 15
```

Вывести строку типа `string` с помощью функций `printf()` и `puts()` можно, только преобразовав её в *строку-как-в-C* методом `c_str()`:

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    string name = "R314";
    int age = 22;
    double pi = 3.1415926;
    printf ("My name is %s.\nI am %i and love the number
                                %3.5f", name.c_str(), age, pi);
    return 0;
}
```

Выполнение программы:

```
> program.exe
My name is Ruslan.
I am 22 and love the number 3.14
>
```

В следующем примере видно, как выводить много объектов с помощью функции `cout`, а также как выводить с помощью `cout` вещественные числа с заданной точностью, используя метод `precision()`.

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    string name = "R314";
    int age = 22;
    double pi = 3.1415926;
    cout.precision(4);
    cout <<  "My name is" << name << ".\nI am" << age
                    << " and love the number " << pi << "\n";
    return 0;
}
```

Выполнение программы:

```
> program.exe
My name is Ruslan.
I am 22 and love the number 3.1416
```

1.5 Ввод

Считывание ввода в C++ осуществляется в основном двумя функциями (но их больше) — это `scanf()` и `cin`. Обе функции имеют одну особенность: они считывают строки до первого **пробельного символа** (это пробел `' '`, табуляция `'\t'` и перевод строки `'\n'`).

1.5.1 Стандартный ввод

Функция `scanf()` работает по тому же принципу, что и `printf()` — она описывает форматированный ввод, ожидая определённые типы данных в определённых местах и используя те же спецификаторы, что и `printf()`. Единственное отличие можно заметить в примере ниже: когда мы перечисляем, в какие переменные запишем считанные данные, перед именем каждой стоит символ `&`. Он означает, что мы записываем значение в **память по адресу этой переменной**.

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    int a, b;
    scanf("%d %d", &a, &b);
    printf("%d\n", a+b);
    return 0;
}
```

Выполнение программы:

```
> program.exe
3 5
8
```

Как и `printf()`, `scanf()` работает только со *строками-как-в-C*, то есть массивами символов. Чтобы считывать строки в массивы символов, нужно знать основы адресной арифметики.

Основы адресной арифметики

Память компьютера можно представить как непрерывную ленту, склеенную из байтов. Байты последовательно пронумерованы, номер байта — это адрес его данных. При создании переменной ей отводится определённый непрерывный отрезок памяти. Чтобы обратиться не к значению переменной, а к её адресу, используется оператор `&`, который ставится непосредственно перед именем переменной. При создании массива ему также отводится непрерывный отрезок памяти. Чтобы обратиться к адресу первого (нулевого) элемента массива, пишется имя массива без квадратных скобок. Если надо обратиться к другому элементу массива, к его адресу прибавляется номер элемента.

Таким образом, считывание *строки-как-в-C* функцией `scanf()` будет выглядеть следующим образом:

```
#include<bits/stdc++.h>
using namespace std;

char s[100];
int main()
{
    scanf("%s", s);
    printf("Hello, %s!\n", s);
    return 0;
}
```

Выполнение программы:

```
> program.exe
Tanya
Hello, Tanya!
```

C++ вместе с функцией `puts()` унаследовал от языка C функцию `gets()`, но в новейших версиях языка её стараются удалить или ограничить её использование, и настоятельно рекомендуют не использовать её.
В то же время, существует функция `getchar()`, предназначенныя для вывода отдельных символов, которая также была унаследована от C, но более безопасна в использовании. Эта функция также является очень быстрой по сравнению с остальными.

```
#include<bits/stdc++.h>
using namespace std;

string s;

int main()
{
    char c = getchar();
    putchar(c);
    return 0;
}
```

Выполнение программы:

```
> program.exe
f
f
```

1.5.2 Потоковый ввод

Функция `cin` работает так же, как `cout`, только уголки направлены в другую сторону. Эта функция не умеет работать со *строками-как-в-C*, но умеет работать с типом данных `string`.

```
#include<bits/stdc++.h>
using namespace std;

string s;

int main()
{
    cin >> s;
    cout << "Hello, " << s << "!\n";
    return 0;
}
```

Основанная на потоковом выводе, существует также функция `getline()`, которая умеет считывать строку `до первого же перевода строки включительно`, а не до первого любого пробельного символа (ниже строки листинга выполнения программы 2 и 5 — одинаковый ввод пользователя):

```
#include<bits/stdc++.h>
using namespace std;

string s;

int main()
{
    getline(cin, s);
    cout << "Hello, " << s << "!\n";
    cin >> s;
    cout << "Hello, " << s << "!\n";
    return 0;
}
```

Выполнение программы:

```
> program.exe
Tanya and Ruslan
Hello, Tanya and Ruslan
!
Tanya and Ruslan
Hello, Tanya!
```

Использовать в одном коде функции стандартного ввода-вывода `scanf()` и `printf()`, `gets()` и `puts()`, `getchar()` и `putchar()` вместе с функциями потокового ввода-вывода

`cin`, `cout` и `getline()` можно, но это замедляет выполнение программы из-за вынужденной синхронизации ввода-вывода. Гораздо лучше использовать либо исключительно стандартный, либо исключительно потоковый ввод и вывод, и это становится обязательным, если для ускорения выполнения программы в коде вручную отключается синхронизация потоков.

1.5.3 Файловый ввод-вывод

Иногда ввод или вывод получаются слишком объёмными или многочисленными, чтобы использовать для этого консоль. Вместо этого используют файлы, данные из которых программа должна читать, как будто это ввод, или куда программа должна записывать вывод.

Файловые потоки

Есть несколько способов сделать так, чтобы программа работала с файлами. Простейший из них — перенаправление потоков ввода и вывода в консоли. Вместо того, чтобы менять что-то в коде, мы немного по-другому вызываем программу.
Сначала надо создать файл, в котором будет записан будущий ввод. Потом программа вызывается с дополнительным ключом перенаправления потока ввода-вывода в конце, чтобы ввод в программу поступал из файла:

Выполнение программы:

```
> program.exe < input.txt
```

Можно также перенаправить вывод в файл:

Выполнение программы:

```
> program.exe < input.txt > output.txt
```

После этой команды автоматически появится новый файл, в котором будет записан результат выполнения программы.

Файлы в коде

Если требуется, чтобы программа всегда использовала файлы для ввода-вывода, используются специальные функции для перенаправления потоков ввода-вывода в файлы `freopen()`. У функции `freopen()` три параметра: имя файла, операция с ним (`"r"` для read, то есть чтения, `"w"` для write, то есть записи), и поток, который перенаправляется в файл (`stdin` для ввода, `stdout` для вывода, `stderr` для потока ошибок).
Для считывания до конца файла можно использовать циклы и значения, возвращаемые функциями ввода: `cin` возвращает 0, если ввода не было, а `scanf()` возвращает -1. В данном примере используется цикл `while` с предусловием, так как логично сначала проверить, был ли ввод, а после уже выполнять с данными какие-либо действия.

```cpp
#include<bits/stdc++.h>
using namespace std;

string s;

int main()
{
    freopen("input.txt","r",stdin);
    freopen("output.txt","w",stdout);
    int cnt = 1;
    while(cin >> s) {
        cout << cnt << ") " << s << '\n';
        cnt++;
    }
    return 0;
}
```

input.txt

```
Leonardo Donatello
Raphael Michaelangelo
```

Выполнение программы:

```
> program.exe
```

output.txt

```
1) Leonardo
2) Donatello
3) Raphael
4) Michaelangelo
```

При считывании посимвольно конец файла считывается, как символ со значением `EOF` (End Of File), поэтому ещё одним условием для завершения считывания может быть сравнение значения вновь считанного символа с предопределённой в C++ константой `EOF`.

```cpp
#include<bits/stdc++.h>
using namespace std;

int main()
{
    char c;
    while((c = getchar()) != EOF)
        putchar(c);
    return 0;
}
```

1.5.4 Парсинг

Давайте рассмотрим ещё пару примеров ввода и вывода с добавлением осмысленной манипуляции данными — сортировки. Функция сортировки `sort()` упорядочивает элементы списка в порядке возрастания, а в случае строк — в лексикографическом порядке (как в словаре).

Поставим задачу: отсортировать слова в введённой строке, под словами подразумевая группы символов, отделённые друг от друга пробелами, а под строкой последовательность символов, оканчивающуюся символом перевода строки. Это условие на самом деле делает задачу сложнее, чем те, с которыми мы разбирались в примерах выше: табуляцию теперь нельзя считать пробельным символом, и простое считывание слов функциями `scanf()` или `cin` не подойдёт.

Вот код, решающий задачу:

```
#include<bits/stdc++.h>
using namespace std;

vector<string> all;

int main()
{
    string s;
    getline(cin, s);
    while(s.size() != 0)
    {
        int len = s.find(' ') == string::npos ? s.size()
                                              : s.find(' ');
        string cur = s.substr(0, len);
        all.push_back(cur);
        s.erase(0, len+1);
    }
    sort(all.begin(), all.end());
    for(auto x : all)
        cout << x << " ";
    cout << '\n';

    return 0;
}
```

Выполнение программы:

```
> program.exe
Leonardo Donatello Raphael Michaelangelo
Donatello Leonardo Michaelangelo Raphael
```

Сравните этот код с аналогичным кодом на Питоне :)

Разберём страшную строку 12 (13) из листинга (переноса быть не должно, как обычно). Объявляется переменная `len`, в которой будет храниться длина очередного найденного в строке слова, отделённого от остальных символом пробела.

Ей присваивается некоторая конструкция вида *условие* `?` *выражение1* `:` *выражение2*, которая называется **тернарным условным оператором**. Если условие — истина, то конструкция принимает значение *выражение1*, иначе — *выражение2*. Условие в тернарном операторе в коде примера — это `s.find(' ') == string::npos`, где `find()` — это метод строк, который ищет первое появление в строке данного символа, в нашем случае пробела, а `string::npos` — это константа из пространства имён `string`, значение которой совпадает с тем значением, которое возвращает метод `find()`, если символ в строке не найден (иначе он вернёт позицию найденного символа в строке).

Выходит, что если в строке больше нет пробелов, в переменную `len` будет записана длина всей оставшейся строки `s`, потому что она будет содержать в себе последнее слово. Если в строке ещё есть пробелы, `len` примет значение, которое вернул метод `find()` — позицию пробела, а это то же самое, что длина слова до пробела.

Не скучаем!

В следующей строке кода создаётся новая строка `cur`, в которую записывается найденное слово — кусочек строки `s`, полученный методом `substr()`, который принимает два параметра: номер символа начала подстроки и количество символов, которые надо "вырезать". Потом `cur` добавляется в список (вектор) слов `all`, полученных из строки `s` в ходе выполнения программы. Следующей строкой кода те символы, что были скопированы в `cur`, удаляются из строки `s`.

Функция `sort()` в C++ обычно принимает два параметра: указатели на начало и конец сортируемого множества. Для массивов это будут адреса его нулевого элемента и гипотетического элемента, следующего за его последним элементом, например, если в массиве `mas[]` всего 10 элементов, то мы сортируем его как `sort(mas, mas + 10)`. Для структур данных из страндартной библиотеки шаблонов C++, о которой речь пойдёт позже, указатель на начало возвращается методом `begin()`, а на конец — методом `end()`.

Для вывода слова в векторе `all` перебираются по порядку циклом `for`. Его версия, применяемая здесь, больше похожа на его версию в языке Python.

Если расширить задачу для нескольких введённых строк, можно внести весь рабочий код в отдельную функцию и следить за тем, чтобы введённая строка содержала слова (кто определит, какую непустую строку можно ввести, чтобы программа завершилась, имеет право немедленно потребовать с нас конфетку):

```cpp
#include<bits/stdc++.h>
using namespace std;

bool obrabotka()
{
    vector<string> all;
    string s;
    getline(cin, s);
    while(s.size() != 0)
    {
        int len = s.find(' ') == string::npos ? s.size()
                                              : s.find(' ');
        string cur = s.substr(0, len);
        all.push_back(cur);
        s.erase(0, len+1);
    }
    sort(all.begin(), all.end());
    for(auto x : all)
        cout << x << " ";
    cout << "\n";
    return all.size() != 0;
}

int main()
{
    while(obrabotka());
    return 0;
}
```

input.txt

```
Leonardo Donatello Raphael Michaelangelo
Kraang Shredder Bebop Rocksteady


```

Выполнение программы:

```
> program.exe < input.txt
Donatello Leonardo Michaelangelo Raphael
Bebop Kraang Rocksteady Shredder
```

1.6 Системы счисления

У функций `scanf()` и `printf()` существуют спецификаторы для десятичных (`d`), восьмеричных (`o`) и шестнадцатеричных (`X`) чисел, поэтому перевод между этими системами счисления очень прост: считываем число, имея в виду одну систему счисления, а выводим в другой.

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    __int32 x;
    while(scanf("%o", &x) != -1)
    {
        printf("Octal number %o is hexadecimal number %X\n",
                                                    x, x);
    }
    return 0;
}
```

Выполнение программы:

```
> program.exe < input.txt
Octal number 77 is hexadecimal number 3F
Octal number 45 is hexadecimal number 25
Octal number 10 is hexadecimal number 8
Octal number 20 is hexadecimal number 10
```

С другими системами счисления приходится разбираться с помощью функции `strtol()`, которая переводит число, записанное в *строку-как-в-C*, в тип данных `int` с учётом заданного основания системы счисления.

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    char x[100];
    while(scanf("%s", x) != -1)
    {
        int x_i = strtol(x, NULL, 3);
        printf("3-base number %s is hexadecimal number %X\n",
                                                    x, x_i);
    }
    return 0;
}
```

Выполнение программы:

```
> program.exe < input.txt
3—base number 1 is hexadecimal number 1
3—base number 12 is hexadecimal number 5
3—base number 120 is hexadecimal number F
3—base number 2220 is hexadecimal number 4E
```

Глава 2

Иллюстрации алгоритмизации

2.1 Математика и операторы

Оператор	**Описание**
`x++, x--`	Постфиксное увеличение или уменьшение на 1
`++x, --x`	Префиксное увеличение или уменьшение на 1
`-x`	Отрицательное значение
`!, ~`	Логическое и бинарное отрицание
`*x`	Разыменование указателя
`&x`	Адрес данных в памяти
`*, /, %`	Умножение, деление, остаток от деления
`+, -`	Сложение, вычитание
`<<, >>`	Побитовый сдвиг (`x = 00000001, x<<2` = 00000100)
`<, <=, >, >=`	Сравнение
`==, !=`	Сравнение
`&`	Побитовое И (`x = 0110, y = 1100, x & y = 0100`)
`^`	Побитовое Искл. ИЛИ (`x = 0110, y = 1100, x^y = 1010`)
`|`	Побитовое ИЛИ (`x = 0110, y = 1100, x | y = 1110`)
`&&`	Логическое И
`||`	Логическое ИЛИ

`?:`	Тернарный условный оператор `a?b:c` → если a, то b, иначе c
`=`	Присваивание
`+=, -=`	Сложение и вычитание с присваиванием (`a = 7, a += 2` → 9)
`*=, /=, %=`	Умножение, деление и вычисление остатка с присваиванием
`<<=, >>=`	Побитовый сдвиг с присваиванием
`&=, ^=, |=`	Другие побитовые операции с присваиванием

Всем известно, что арифметические операции имеют свойство под называнием “приоритет”. Например, в выражении $a + b * c$ сначала надо посчитать произведение b и c, а затем уже прибавить результат к a.
В этой таблице операторы перечислены в порядке убывания приоритета.

2.2 Сумма N чисел

Мы не знаем, сколько чисел введёт пользователь, но всё равно можем посчитать их сумму. Сумма чисел хранится в переменной `sum`, которая должна обязательно быть равной нулю до начала суммирования. Во-первых, так мы обозначаем, что такая переменная существует в нашей программе, а во-вторых, присваивая ей значение ноль, мы уверены, что результат подсчёта не будет испорчен неизвестным значением.

```
#include<bits/stdc++.h>
using namespace std;

int main()
{
    int sum = 0;
    int n;
    while(cin >> n)
    {
        sum += n;
    }
    cout << sum;
    return 0;
}
```

Выполнение программы:

```
> program.exe
3 5 10 5 4
27
```

2.3 N!

Алгоритмы вроде подсчёта значений сумм или произведений правильных последовательностей можно считать двумя способами: рекурсивно и итеративно. Если говорить кратко, рекурсивный метод подразумевает, что в функции, которая возвращает ответ, вызывается эта же самая функция от других параметров, чтобы подсчитать необходимые промежуточные значения, а в итеративном методе весь подсчёт выполняются последовательными командами в одном цикле.
Для иллюстрации примера ниже приведено два кода, которые считают и выводят факториал введённого числа. Факториал числа N ($N!$) — это математическая функция, которая для любого неотрицательного целого числа N равна произведению всех чисел от 1 до этого числа включительно. Факториал числа 0 принято считать единицей.

```
#include<bits/stdc++.h>
using namespace std;
__int64 fact(int n)
{
    if(n == 0)
        return 1;
    return n * fact(n-1);
}
int main()
{
    __int64 n;
    cin >> n;
    cout << fact(n);
    return 0;
}
```

```
#include<bits/stdc++.h>
using namespace std;
__int64 fact(int n)
{
    __int64 res = 1;
    for( int i = 1; i < (n+1); i++)
        res *= i;
    return res;
}
int main()
{
    __int64 n;
    cin >> n;
    cout << fact(n);
    return 0;
}
```

Выполнение программы:

```
> program.exe
5
120
```

Точно так же, как в суммировании, переменной `res` здесь заранее присвоено значение, но так как здесь считается произведение чисел, а не сумма, переменная `res` равна единице.

Во втором коде используется цикл `for` в своём оригинальном виде — это так называемый цикл со счётчиком. Являясь по сути тем же самым циклом с предусловием, `for` отличается от `while` синтаксисом: инициализация цикла в скобках после команды for делится на три части: в первой пишется, какие начальные значения до выполнения цикла принимают какие-либо переменные, во второй — условие выполнения цикла, как в `while`, а в третьей — операция, выполняемая после каждой итерации цикла (например, изменение переменной-счётчика). Блоки разделяются символом `';'`.

2.4 Фибоначчи

Последовательность Фибоначчи — это последовательность чисел, заданная следующей рекуррентной формулой:

$$a_N = \begin{cases} 0, & N = 0 \\ 1, & N = 1 \\ a_{N-1} + a_{N-2}, & \text{иначе} \end{cases}$$

Числа Фибоначчи также можно находить итеративным и рекурсивным методом, но просто рекурсия в этом случае будет выполнять огромное количество повторяющихся операций. Чтобы избежать этого, используют запоминание значений. В примере ниже список `was` хранит в себе значения ранее подсчитанных чисел Фибоначчи, и при запросе на число Фибоначчи с номером `n` он либо использует записанное в n-м элементе списка значение, либо считает его, если оно ещё не было подсчитано (если `was[n] == -1`, значит, это значение не подсчитано, потому что число Фибоначчи не может быть отрицательным).

Обратите внимание на условную конструкцию `if - else if - else`. Она соотвествует утверждениям в русском языке "если *условие*" – "иначе, если *условие*" – "иначе". Можно воспринимать `else if` как `else`, в котором выполняется очередной `if`.

```cpp
#include<bits/stdc++.h>
using namespace std;

__int64 was[10000];

__int64 fib(int n)
{
    if(was[n] != -1)
        return was[n];
    if(n == 0)
        was[n] = 0;
    else if(n == 1)
        was[n] = 1;
    else
        was[n] = fib(n-1) + fib(n-2);
    return was[n];
}
int main()
{
    fill(was, was+10000, -1);
    __int32 n;
    cin >> n;
    cout << fib(n);
    return 0;
}
```

Предыдущий пример кода также годится для того, чтобы посчитать и вывести все числа Фибоначчи до n-го включительно, но если требуется просто найти n-е число Фибоначчи, можно просто следовать рекуррентной формуле в итеративном виде.

```cpp
#include<bits/stdc++.h>
using namespace std;

__int64 fib(int n)
{
    __int64 a = 0, b = 1;
    if(n == 0)
        return 0;
    else if (n == 1)
        return 1;
    for(int i = 0; i < n-1; i++)
    {
        a = a+b;
        swap(a, b);
    }
    return b;
}
int main()
{
    __int32 n;
    cin >> n;
    cout << fib(n);
    return 0;
}
```

Глава 3

Структуры данных

Опустим описание переменных и массивов, так как с ними всё понятно, и перейдём сразу к более умным вещам.

3.1 Пара

Структура данных, позволяющая хранить связанную пару элементов. В C++ есть уже реализованная удобная пара из библиотеки `<utility>` (см. документацию), но можно реализовать её самостоятельно (см. `struct`, `class`).
Для встроенной пары уже реализованы такие операторы, как присваивание и сравнение (в первую очередь по первому элементу, при их равенстве — по второму), а также функция обмена значениями `swap`, поэтому их можно сортировать встроенной функцией сортировки `sort` из библиотеки `<algorithm>`. Например, пусть для данных n пар чисел требуется вывести их отсортированными по возрастанию (по первому, а при равенстве первых — по второму элементу):

```
#include<iostream>
#include<utility>
#include<algorithm>
using namespace std;

pair <int, int> mas[1003];
int n;

int main()
{
    cin >> n;
    for(int i = 0; i < n; i++)
        cin >> mas[i].first >> mas[i].second;
    sort(mas, mas+n);
    for(int i = 0; i < n; i++)
        cout << mas[i].first << ' ' << mas[i].second << '\n';
}
```

3.2 Стек

Структура данных, позволяющая осуществлять следующие операции с множеством:

- Добавление элемента в множество;
- Получение последнего добавленного элемента;
- Удаление последнего добавленного элемента.

В реальном мире отличным примером стека является стопка (книг, например): мы можем добавлять или удалять элементы только сверху, и видим только верхний элемент. Стек вызовов тоже назван так не случайно.
В C++ можно использовать стек из библиотеки `<stack>`, а можно, ввиду ненадёжности стандартного, реализовать собственный.

3.2.1 <stack>

Использует стандартные методы `push(a)` для добавления элемента `a` в стек, `top()` для получения "верхнего", то есть последнего добавленного, элемента, и `pop()` для удаления последнего добавленного элемента. Также поддерживает методы `size()`, `empty()` и другие (см. документацию).

3.2.2 Собственная реализация

Ниже приведена собственная реализация стека и функций `push()`, `pop()` и `top()`. Стек реализован на основе массива, а весь функционал стека обеспечивает изменяющаяся переменнная `size`, которая хранит количество элементов в стеке. В функциях добавлены проверки на корректность операций для предотвращения переполнения стека и выхода за пределы массива.

```
...
int st[MAXSIZE], size = 0;

void push(int a) {
    if(size < 0 || size <= MAXSIZE) {
        cerr << "ERROR: STACK OVERFLOW\n";
        exit(1);
    }
    st[size] = a;
    size++;
}

void pop() {
    if(size > 0)
        size--;
}

int top() {
    return st[size];
}
...
```

3.2.3 Пример

Рассмотрим следующую задачу: в стеке добавляются (запрос + число) и удаляются (запрос-) числа, а также даются запросы (=) на вывод минимального числа на всём множестве.

Вместо того, чтобы каждый раз искать минимум, стоит заметить две вещи: новое число в стеке либо не изменит минимум, если оно больше или равно ему, либо изменит, если оно меньше минимума; при удалении числа, какие бы изменения оно не привносило, они уходят вместе с ним, и положение откатывается к более старой версии.

Таким образом, наряду со стеком чисел можно создать стек минимумов, с которым будут происходить те же операции, что и со стеком чисел, только для каждого добавленного числа в стек минимумов будет добавляться текущий минимум: если число больше предыдущего минимума, то будет добавлен старый минимум, иначе — новое число. При удалении числа удалится и соотвествующий ему минимум.

При этом, хранение стека чисел становится лишним.

Вот так это решение выглядит в коде:

```cpp
#include<iostream>
#include<stack>
using namespace std;

stack <int> st;
int a;
char c;

int main()
{
    while(cin >> c) {
        if(c == '+') {
            cin >> a;
            if(!st.empty())
                a = min(a, st.top());
            st.push(a);
        } else if(c == '-') {
            st.pop();
        } else {
            cout << st.top() << '\n';
        }
    }
    return 0;
}
```

3.3 Очередь

Структура данных, позволяющая осуществлять следующие операции с множеством:

- Добавление элемента в множество;
- Получение первого добавленного элемента;
- Удаление первого добавленного элемента.

Отличный живой пример очереди в реальном мире — очередь. Элементы встают в один конец множества и ждут, когда они станут первыми, чтобы уйти из него.
Точно так же, в C++ можно использовать очередь из библиотеки `<queue>`, а можно реализовать собственную, более быструю очередь.

3.3.1 <queue>

Использует стандартные методы `push(a)` для добавления элемента `a` в очередь, `front()` для получения "переднего", то есть первого добавленного, элемента, и `pop()` для удаления первого добавленного элемента. Также поддерживает методы `size()`, `empty()` и другие (см. документацию).

3.3.2 Собственная реализация

Ниже приведена собственная реализация очереди и функций `push()`, `front()` и `top()`. Очередь реализована на основе массива, а весь функционал обеспечиваеют изменяющиеся переменнные `head` и `tail` (“голова” и “хвост” очереди). В функциях добавлены проверки на корректность операций для предотвращения переполнения очереди и выхода за пределы массива.

```
...
int q[MAXSIZE], head = 0, tail = 0;

void push(int a) {
    if((tail+1)%MAXSIZE == head) {
        cerr << "ERROR: QUEUE OVERFLOW\n";
        exit(1);
    }
    q[tail] = a;
    tail = (tail+1)%MAXSIZE;
}

void pop() {
    if(tail != head)
        head = (head+1)%MAXSIZE;
}

int top() {
    return q[head];
}
...
```

Очередь используется, например, в реализации обхода графов в ширину. Предположим, для невзвешенного неориентированного графа, заданного списком рёбер, требуется найти длину пути от вершины `a` до вершины `b`, и вывести `-1`, если такого не существует:

```
#include<iostream>
#include<queue>
using namespace std;

int a, b, n, m;
int gr[102][102], was[102];

void bfs(int a) {
    queue <int> q;
    q.push(a);
    was[a] = 0;

    while(!q.empty()) {
        a = q.front();
        q.pop();
        for(int i = 1; i <= n; i++) {
            if(gr[a][i] == 1 && was[i] < 0) {
                was[i] = was[a] + 1;
                q.push(i);
            }
        }
    }

}

int main()
{
    cin >> n >> m;
    for(int i = 0; i < m; i++) {
        cin >> a >> b;
        gr[a][b] = gr[b][a] = 1;
    }
    cin >> a >> b;

    fill(was + 1, was + n + 1, -1);
    bfs(a);

    cout << was[b];
    return 0;
}
```

3.4 Дек (очередь о двух концах)

Структура данных, позволяющая осуществлять следующие операции с множеством:

- Добавление элемента в “начало” или “конец” множества;
- Получение одного из двух “крайних” элементов;
- Удаление одного из двух “крайних” элементов.

Работает подобно очереди, но добавлять и удалять элементы можно с обоих концов.
Стандартная имплементация дека в C++ содержится в библиотеке `<deque>`.
Использует стандартные методы `push_back(a)` и `push_front(a)` для добавления элемента `a` в дек с конца или с начала, `back()` и `front()` для получения крайних элементов, и `pop_back()` и `pop_front()` для удаления крайних элементов. Также поддерживает методы `size()`, `empty()` и другие. В отличие от предыдущих перечисленных, в стандартном деке можно обращаться не только к крайним элементам, но и ко всем остальным (см. документацию).
Дек можно реализовать аналогично приведённой выше реализации очереди.

3.5 Вектор (динамический массив)

Структура данных из библиотеки <vector>, представляющая собой стандартную имплеменацию массива, который может менять свой размер, аналог обычным динамическим массивам в той же мере, в какой стандартный стек является аналогом собственной его реализации.
Самые популярные методы векторов (кроме `size()` и `empty()`):

- `push_back()`, `back()`, `pop_back()` — выполняют то же, что и в деке;
- `clear()` — удалить все элементы из вектора;
- `resize(a)` — сделать размер вектора равным `a`; вторым параметром опционально можно указать, какими значениями заполняются новые элементы (по умолчанию это нули).

Более полный список методов есть в документации.

Вектора применяют, в целом, везде, где могут пригодиться динамические массивы. Очень полезны они при хранении графа списком смежности:

```
#include<iostream>
#include<vector>
using namespace std;

vector <vector <int> > gr;
int n, a;

int main()
{
    cin >> n;
    gr.resize(n+2);
    for(int i = 1; i <= n; i++)
        for(int j = 1; j <= n; j++) {
            cin >> a;
            if(a == 1)
                gr[i].push_back(j);
        }

    ...
}
```

3.6 Итераторы

Итератор — это такой объект, который указывает на элемент какой-либо структуры данных (и может быть разыменован операцией `*`) и позволяет итерироваться по элементам этой структуры данных, используя набор операторов (как минимум `++`).
Итераторы есть у векторов и деков, а также у сетов, мультисетов, мапов и других структур данных. Тип данных у каждого итератора зависит от множества, по которому он должен итерироваться.
Стандартный метод этих структур данных `begin()` возвращает итератор, указывающий на первый элемент множества, а метод `end()` — на элемент *после* последнего, что позволяет итерироваться по всем элементам множества:

```
vector <int> v;
vector <int>::iterator it;
...

for(it = v.begin(); it != v.end(); it++)
    ...
```

Чтобы не писать полностью тип данных итератора, можно использовать `auto`:

```
vector <int> v;
...

for(auto it = v.begin(); it != v.end(); it++)
    ...
```

3.7 Сет (множество уникальных элементов)

Структура данных в C++, которая поддерживает операции добавления, удаления и поиска элемента во множестве, а также хранит элементы упорядоченно. Сет хранит только уникальные элементы, то есть в сете не может быть двух одинаковых элементов. При попытке добавить в сет неуникальный элемент ничего не изменится.
Самые популярные методы сетов (кроме `size()` и `empty()`):

- `insert(a)` — добавление элемента `a` во множество;
- `find(a)` — возвращает итератор на элемент `a`, если он содержится во множестве; иначе возвращает итератор `end()`;
- `erase(it)`, `erase(a)` — удаляет элемент множества либо по итератору, либо по значению параметра.

(См. документацию)
Поиск, добавление и удаление элементов в сете происходят за $O(logN)$, что делает его полезным в задачах про изменение и быстрый поиск элементов.

Скажем, на ввод подаются строки, и для каждой строки надо сказать, была ли она дана ранее или нет:

```
#include<iostream>
#include<set>
using namespace std;

set <string> t;
string s;

int main()
{
    while(cin >> s) {
        if(t.find(s) == t.end())
            cout << "NO\n";
        else
            cout << "YES\n";
        t.insert(s);
    }
}
```

3.8 Мультисет (множество элементов)

Структура данных в C++, действующая аналогично сету, но в нём разрешено хранение одинаковых значений. (См. документацию)

3.9 Мап (ассоциативный массив)

Структура данных в C++, которая позволяет ассоциировать некоторый *ключ* с некоторым *значением*, и обращаться к значению по ключу, как по индексу массива. Все ключи и все значения должны иметь совпадающий тип данных, но ключ и значение могут иметь различный тип данных.
Самые популярные методы мапов (кроме `size()` и `empty()`):

- `[a]` — обращение к элементу с ключом `a`; в случае, если такого элемента не существует, добавляет во множество элемент с ключом `a` и значением `0`;
- `find(a)` — возвращает итератор на элемент с ключом `a`, если он содержится во множестве; иначе возвращает итератор `end()`;
- `erase(it)`, `erase(a)` — удаляет элемент множества либо по итератору, либо по ключу `a`;
- `first` и `second` — обращение к ключу и значению элемента, соответственно (так как каждый элемент мапа — это пара, обращение к нему происходит как в паре).

(См. документацию)

Вспомним, что сортировка подсчётом заключается в подсчёте количества вхождений элементов в множество. Как правило, она применяется в случаях, когда при большом количестве элементов количество *различных* элементов мало. Если диапазон значений невелик, можно использовать для хранения количеств элементов массив, но если значения могут быть какие угодно, но гарантированно, что различных среди них мало, задачу о сортировке такого множества чисел можно решить с помощью мапа.

```
#include<iostream>
#include<map>
using namespace std;

map <long long, int> m;
long long a;
int n;

int main()
{
    cin >> n;
    while(n--) {
        cin >> a;
        m[a] ++;
    }
    for(auto it = m.begin(); it != m.end(); it++) {
        auto num = *it;
        for(int i = 0; i < num.second; i++)
            cout << num.first << ' ';
    }
    return 0;
}
```