C++ [2]

На прошлой лекции рассказывалось, как использовать переменные и условные операторы. Но на этом особо ничего не построить, только какие-то примитивные программы. Поэтому введём ещё несколько конструкций.

Циклы

Для начала научимся многократно выполнять похожие действия. В программировании для этой цели могут использоваться циклы. В C++ их 3 основных вида.

Понятие, которое, может быть пока не всем знакомо — итерация. Значит оно "повторение какого-либо действия".

Цикл for

Пожалуй, самым полезным и мощным из циклов является цикл for. Его синтаксис выглядит так:

```
for (/*объявление переменных*/; /*условие*/; /*действие после тела*/) {
    /* тело цикла */
}
```

Условие будет проверяться перед каждую итерацию перед заходом в тело цикла. Вообще, вы можете писать произвольные условия и произвольные действия при итерации.

Чаще всего встречается использование этого цикла со счётчиком:

```
for (int i = 0; i < 10; i++) {
    /* давайте напишем вывод переменной i */
    cout << i << endl;
}
```

continue и break

Как и во многих других языках, в C++ есть операторы для работы с любыми циклами. В C++ их два:

- *continue* — начинает следующую итерацию
- *break* — завершает цикл безусловно

В качестве примера использования *continue*, давайте напишем программу, которая выводит только чётные числа до 10.

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 1) {  // если число даёт остаток 1 при
    ↪  делении на 2
        continue;      // то перейдёт к следующему числу
    }
    cout << i << endl;
}
```

Перепишем цикл, используя *break*, чтобы он заканчивался, когда i становится больше или равно 5.

```
for (int i = 0; i < 10; i++) {
    if (i % 2 == 1) {
        continue;
    }
    if (i >= 5) {
        break; // цикл завершится на 6-й итерации
    }
    cout << i << endl;
}
```

Цикл while

while - цикл с предусловием. В круглых скобках вы пишете условие, при котором цикл будет выполняться.

Давайте перепишем предыдущий код, используя цикл *while*. Для этого нужно заранее завести переменную и присвоить ей значение 0.

```cpp
int i = 0;
while (i < 10) {
    if (i % 2 == 1) {
        i += 1; // если не написать здесь инкремент, то цикл будет работать вечно
        continue;
    }
    if (i >= 5) {
        break; // цикл завершится
    }
    cout << i << endl;
    i += 1;
}
```

Цикл do while

В C++ также есть цикл с постусловием. Если в *while* сначале смотрится условие, и если оно верно, затем выполняется тело цикла, то в *do while* всё наоборот. Сначала выполняется тело, а затем проверяется условие. Если оно ложно, то цикл завершает свою работу.

Цикл *do while* встречается реже предыдущих двух, но всё-таки бывает полезен.

Снова напишем вывод всех чисел от 0 до 10.

```cpp
int i = 0;
do {
    cout << i << endl;
    i += 1;
} while (i < 10);
```

Вектора

Что мы будем делать с введёнными циклами? Во-первых, давайте научимся считывать данные с их помощью.

Часто в задачах нам нужно хранить не пару-тройку чисел, а довольно большое количество. В C++ для того, чтобы хранить линейные последовательности значений существует специальный тип. Он называется *vector*. По сути, это реализация массива переменной длины.

Для того, чтобы воспользоваться вектором в программе, нужно подключить заголовочный файл с таким же именем "#include <vector>".

При объявлении вектора необходимо указывать, какой тип будет в нём храниться. Это так называемый шаблон класса вектор.

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

Что можно делать с вектором? Например, можно указать размер вектора при объявлении, так как по умолчанию он создаётся пустым. При указании размера, вектор заполняется стандартными значениями типов. Для чисел это 0.

```
int n;
cin >> n;
vector<int> v(n); // сейчас здесь n нулей
```

Также можно самим указать, какими значениями должен быть заполнен вектор. Для этого нужно указать желаемое значение вторым параметром.

```
int n;
cin >> n;
vector<int> v(n, -1); // сейчас здесь n нулей
```

Теперь давайте считаем в него n чисел. Для обращения к элементам вектора, нужно просто написать справа от названия номер элемена в квадратных скобках. Так как индексация элементов в C++ с нуля, то в цикле нужно будет считать с 0-го по $n - 1$ элемент вектора.

```
int n;
cin >> n;
vector<int> v(n);
for (int i = 0; i < n; i++) {
    cin >> v[i];
}
```

Также можно удалять или добавлять элементы в уже существующий вектор. Чаще всего на олимпиадах нужно добавлять что-то в конец вектора.

```
v.pop_back(); // удалить последний элемент
v.push_back(2); // добавить двойку в конец
```

Для вывода размера вектора используется метод $size()$:

```
cout << v.size(); // выводим размер вектора в консоль
```

Ещё есть метод, который говорит нам, является вектор пустым или нет: $empty()$. Он возвращает булевое значение $true$ или $false$.

```
cout << v.empty(); // выводит 0, если пустой, и 1, если
↪  непустой
```

Иногда нужно поменять размер нашего вектора. Для этого есть метод $resize()$, аргументом которого нужно указывать новый размер. Если вы уменьшаете размер вектора, то все элементы, которые были правее новой границы, исчезнут. Если вы его увеличиваете, то добавляются значения по умолчанию (для чисел — нули). Также, как и при объявлении, можно указать, какие значения должны дополниться.

```
v.resize(n / 2);
v.resize(n + 100, 100);
```

Бывает также, что нужно переобъявить вектор. Для этого можно использовать метод $assign()$, который принимает новый размер ветора и значения, которыми мы хотим заполнить весь этот вектор.

```
v.assign(100, -1); // вектор из ста -1
```

Для того, чтобы удалить все элементы из вектора, существует метод $clear()$.

```
v.clear(); // теперь в нём ничего нет
```

Выводить вектор можно таким же образом, каким считывали.

```
for (int i = 0; i < int(v.size()); i++) {
    cout << v[i] << ' ';
}
```

О выходах за границы и других ошибках

Пусть у нас есть такой код.

```
vector<int> v(n);
for (int i = 0; i < n; i++) {
    cin >> v[i];
}
v.pop_back();
for (int i = 0; i < n; i++) {
    cout << v[i] << endl;
}
```

Он нормально скомпилируется, но на этапе выполнения мы можем заметить пару ошибок.

Первая, и самая очевидная — при вводе $n = 0$, v.pop_back() не сможет удалить ни одного элемента и программа завершится на этом месте с ошибкой исполнения. В C++ за вас ничего проверяться не будет.

Вторая ошибка — вывод всех n элементов, тогда как в векторе остался только $n - 1$ элемент. По умолчанию g++ не выведет никаких предупреждений, но в действительности произойдёт выход за границы вектора и выведется то значение, которое лежало на $n-1$ позиции до удаления. Если же попытаться вывести значения n, $n+1$ и далее, то в выводе консоли мы увидим мусор, который лежал в тех местах когда-то до этого.

Чтобы отслеживать ошибки выхода за границы, при компиляции нужно указывать опцию -fsanitize=address. Тогда во время выполнения программы, при выходе за границу, нам выведется ошибка, а если добавить -g, то в выводе об ошибке будет указан также номер строки, в которой она возникла. Исправленный код выглядит так:

```
vector<int> v(n);
for (int i = 0; i < n; i++) {
    cin >> v[i];
}
if (!v.empty()){ // оператор отрицания ! (!true = false,
↪  !false = true)
    v.pop_back();
}
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << endl;
}
```

Как обычно даются входные данные

Есть 4 типовых способа задания входных данных.

n чиселок

Самый нормальный и удобный вариант входных данных. На вход даётся размер массива чисел, а затем сам массив.

Посмотрим на примере программы, суммирующей n чисел из введённого массива.

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

int main() {
    int n;
    cin >> n;
    int sum = 0;
    for (int i = 0; i < n; i++) {
        int x;
        cin >> x;
        sum += x;
    }
    cout << sum << endl;
    return 0;
}
```

Считать все числа

Вам сразу даётся массив чисел, без указания его длины.

Что нужно делать с таким вводом? Оказывается, *cin* возвращает своё состояние. Если можно считать очередное число, то это состояние эквивалентно *true*. Если же при считывании встретился символ конца файла или произошла какая-то ошибка, то вернётся значение, эквивалентное *false*.

При вводе с консоли в Linux символ конца файла получается при нажатии ctrl-d.

Поэтому мы можем переписать предыдущую программу и для такого ввода, используя цикл *while*. Для этого нужно будет заранее завести переменную, в которую будем читать числа из входного массива.

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

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

Массив с ограничительным элементом

Иногда при вводе указывают, что считывать нужно до того момента, когда в массиве встретится определённый элемент. Например, -1.

Программа получается почти такой же, как в прошлом примере, отличается только в один if.

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

int main() {
    int sum = 0;
    int x;
    while (cin >> x) {
        if (x == -1) {
            break;
        }
        sum += x;
    }
    cout << sum << endl;
    return 0;
}
```

В одном файле много тестов

Иногда жюри хочет, чтобы ваша программа выполняла сразу много тестов из одного файла. Тогда говорится, что есть t тестовых наборов ($t < 1000$), и в каждом из них ввод уже задаётся одним из привычных способов.

В таком случае удобно создавать функцию *solve*() типа *void*, в которой будут проходить обычные вычисления. Тип *void* у функции значит, что она ничего не возвращает.

Напишем, например, программу, которая для каждого тестового набора двух чисел выводит наибольшее из них.

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

void solve() {
    int a, b;
    if (a > b) {
        cout << a << endl;
    }
    else {
        cout << b << endl;
    }
}

int main() {
    int t;
    cin >> t;
    for (int i = 0; i < t; i++) {
        solve();
    }
    return 0;
}
```

Ещё про ввод и вывод в C++

В олимпиадах иногда приходится работать с вводом-выводом по 10^6 и более чисел. С таким 10^6 чисел на моём ноутбуке *cin* справляется за 0.46 секунды. А в задачах обычно стоит ограничение на 1 секунду. Получается, довольно много времени уходит в никуда.

Вообще, в C++ стандартные ввод и вывод не такие уж и быстрые, потому что C++ изначально строился так, чтобы быть достаточно хорошо обратно совместимым с Си. А в Си есть своя библиотека с вводом-выводом. В итоге, библиотека ввода-вывода C++ была построена так, чтобы они могли нормально вместе использоваться. Нам же это не нужно.

Поэтому, чтобы ускорить ввод-вывод в C++, можно сделать следующим образом: сказать "я не хочу, чтобы что-то синхронизировалось с сишной библиотекой". В коде это будет записано с помощью "ios::sync_with_stdio(false);".

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

int main() {
    ios::sync_with_stdio(false);
    int a;
    for (int i = 0; i < 1000000; i++) {
        cin >> a;
    }
    return 0;
}
```

Это считывание работает уже за 0.182 секунды, что уже лучше.

Что ещё можно сделать для ускорения работы ввода-вывода?

По умолчанию каждый раз, когда вы читаете что-то со стандартного ввода, у вас выводится всё из стандартного вывода. Каждый вывод — это операция не дешёвая, поэтому вы можете написать после отключения синхронизации "cin.tie(0);". Теперь вывод будет происходить при заполнении буфера вывода или по завершении программы. Предыдущая программа при добавлении cin.tie(0) работает за 0.175 секунды.

И последнее — **не используйте** на олимпиадах *endl*. Вместо него используйте символ перевода строки ". Потому что *endl* также провоцирует вывод из буфера вывода, тем самым замедляет работу (быстрее

вывести один раз, но много, чем понемногу, но часто).

Следующая программа безо всяких оптимизаций работает за 4 секунды. Перебор. А с указанными оптимизациями получается 0.32 секунды.

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

int main() {
    ios::sync_with_stdio(false);
    cin.tie(0);
    int a;
    for (int i = 0; i < 1000000; i++) {
        cin >> a;
        cout << a << '\n';
    }
    return 0;
}
```

Строки

Ещё одна полезная разновидность линейных последовательностей – строки. Чтобы с ними работать в C++, нужно подключить заголовочный файл string "#include <string>".

Строки работают как и другие типы. Например, вы можете считывать и выводить их как и int переменные. cin считывает последовательность символов до первого пробела или иного разделителя.

Со строками можно работать также, как и с векторами. Те же самые методы присутствуют. Также строки в C++, в отличие от Python, можно изменять.

Что ещё есть полезного в строках: их можно складывать, используя оператор +, брать у них подстроки с помощью метода substr(), указывая, с какого символа начинается подстрока, и сколько символов нужно вывести.

Продемонстрируем всё это на примере кода.

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

int main() {
    string s; // пустая строка
    cin >> s;
    cout << s << '\n';
    s[2] = 'a';
    cout << s << '\n';
    string t = "eto stroka\n";
    cout << s + t;
    cout << s.substr(3, 2) << '\n'; // индексация с нуля
    return 0;
}
```

Про копирование в C++

Допустим, у вас есть вектор на 10^5 элементов. Передавать его просто так в функцию — дорого, так как нужно 10^5 элементов скопировать. В отличие от Python, где в функцию передаётся указатель на объект, в C++ этот объект полностью копируется. Соответственно, если его изменять в самой функции, в остальной части программы он не изменится.

Для того, чтобы передаваемый объект не копировался, и его можно было бы как-то менять, при передачи аргумента, справа от его типа нужно поставить знак амперсанда \. В таком случае, в функцию будет передаваться не сам объект, а ссылка на него, но взаимодействие с ним в функции никак не изменится.

Проиллюстрировать работу со ссылками можно на следующем примере.

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

void f(int x) {
    x *= 2;
}

void g(int& x) { // передаём число по ссылке
    x *= 2;
}

int main() {
    int x;
    cin >> x;
    f(x);
    cout << x << endl;
    g(x);
    cout << x << endl;
    return 0;
}
```

Если вам нужно эффективно передать объект в функию, но при этом не хотите, чтобы этот объект мог в ней измениться, в аргументе нужно написать "const obj_type& obj_name".

```cpp
#include <iostream>
#include <vector>

void func(const vector<int>& v) {
    v.push_back(4);
}

int main() {
    vector<int> v(3, 1);
    func(v);
    return 0;
}
```

Этот код выведет ошибку, так как мы попытались изменить константный объект.

range-based for

Если вы привыли к синтаксису Python, где можно циклом for проходиться по каким-нибудь контейнерам типо строк или списков, то в C++ вы можете использовать похожий приём.

Посмотрим на примере:

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

int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; i++) {
        cin >> v[i];
    }
    for (int x: v) { // x пробегает все элементы v в порядке,
    ↪  в каком они там лежат
        cout << x << '\n';
    }
}
```

Такая конструкция бывает полезна только когда вам нужно просто пройтись по всем элементам контейнера, потому что если вам нужен индекс элемента, вы его не получите. Так что, если нужен индекс элемента, используйте *for* со счётчиком.

Элементы в range-based *for* можно изменять, если вы также передадите их по ссылке. Для этого, опять же, к типу элементов нужно добавить амперсанд.

При этом наш код станет таким.

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

int main() {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int& x: v) { // x можно изменять в цикле
        cin >> x;
    }
    for (int x: v) { // здесь x - копии значений вектора
        cout << x << '\n';
    }
}
```

Функции для работы со строками

Для работы со строками есть ещё пара полезных функций.

Если вы считали строку и хотите её перевести в число, то в C++ вы можете использовать функцию *stoi*(), аргументом которой является эта строка. Для перевода в тип *long long* есть *stoll*(), для double есть *stod*().

Также есть обратное преобразование, его можно делать при помощи to_string(), которая принимает число любого типа.

```
#include <iostream>
#include <string>

int main() {
    string s;
    cin >> s;
    cout << stoi(s) << endl;
    cout << stoll(s) << endl;
    cout << stod(s) << endl;
    int x;
    cin >> x;
    cout << to_string(x) << endl;
    return 0;
}
```

Ещё два типа переменных

Тип *bool* — принимает значение $false$ или $true$ (0 или 1).

Также есть символьный тип *char*, занимает 1 байт памяти.

Напишем программу, которая считывает и выводит первый непробельный символ:

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

int main() {
    char c;
    cin >> c;
    cout << c << endl;
}
```

Иногда нам нужно узнавать, какой именно у нас есть символ: буква, цифра или что-то ещё. В C++ символы кодируются при помощи *ascii* - 256 возможных символов. Посмотреть таблицу этой кодировки можно, написав в консоли "man ascii".

Что полезного есть в этой кодировке? То, что все числа и буквы английского алфавита идут в привычном нам порядке.

Поэтому, если нам известно, что на вход приходит строчная буква английского алфавита, мы можем узнать её номер вычев просто первый символ этого алфавита.

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

int main() {
    char c;
    cin >> c;
    cout << c - 'a' << endl; // если ввели z, выведется 25
}
```

Для того, чтобы так перевести цифру, можно вычесть символ нуля: $c - '0'$.

Если нам нужно проверить, является введённый символ строчной буквой или цифрой, мы можем написать следующие if:

```
if ('a' <= c && c <= 'z') {
    cout << "lower\n"; // строчная буква
}
if ('0' <= c && c <= '9') {
    cout << "digit\n";
}
```

Ещё одним вариантом определить, какой символ содержится в элементе типа *char* — это использовать функции из заголовочного файла *cctype*.

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

int main() {
    char c;
    cin >> c;
    if (islower(c)) {
        cout << "lower\n";
    }
    if (isdigit(c)) {
        cout << "digit\n";
    }
}
```

Все остальные функции можно посмотреть самостоятельно в описании заголовочного файла.