**Использование стандартной сортировки в C++**

**Функции sort и stable\_sort**

Для сортировки массивов и векторов в STL есть функции

**sort**

 и

**stable\_sort**

 (последняя реализует устойчивую сортировку, которая не меняет порядок элементов массива, если они равны).

Для сортировки вектора A алгоритм сортировки нужно вызывать так:

**sort(A.begin(), A.end())**

Для сортировки массива A из n элементов функцию сортировки нужно вызывать так:

**sort(A, A + n)**

Но в некоторых случаях необходимо использовать более сложные, настраиваемые функции сортировки, использующие нестандартный порядок. Есть несколько способов задавать настраиваемый порядок сортировки.

**Функция-компаратор**

Алгоритм сортировки сравнивает элементы при помощи операции "меньше". Можно самостоятельно реализовать операцию "меньше" и использовать её в алгоритме сортировки. Например, давайте упорядочим числа по последней цифре, а если последние цифры равны, то порядок неопределён. В этом случае мы вводим между ними отношение порядка, обозначим его ≺. Например, следующие отношения будут верны, так как последняя цифра левого числа меньше, чем последняя цифра правого числа:

2≺3

12≺3

102≺73

98≺99

А следующие отношения порядка неверны:

3⊀2

3⊀3

7⊀22

103⊀113

Отношение порядка должно удовлетворять следующим свойствам:

1. a⊀a.

2. Если a≺b, то b⊀a.

3. Если a≺b и b≺c, то a≺c.

Для того, чтобы использовать функцию-компаратор, необходимо объявить функцию, которая получает на вход два сравниваемых значения и возвращает значение типа bool, при этом она возвращает true, если первый аргумент меньше второго аргумента, то есть обязан в упорядоченном массиве идти раньше второго.

Пример реализации такой функции для сортировки значений по последней цифре:

**bool cmp(int a, int b)**

**{**

**return a % 10 < b % 10;**

**}**

Эта функция передается в функцию sort третьим параметром:

**sort(a.begin(), a.end(), cmp);**

Если есть два элемента a и b, такие, что a⊀b и b⊀a, то с точки зрения сортировки эти элементы "равны". В нашем примере это два числа, оканчивающиеся на одинаковые цифры. Тогда их порядок не определен, если используется функция sort. Если же использовать функцию stable\_sort, то эта функция не переставляет равные элементы, то есть сохраняется тот же порядок, который был в массиве до сортировки.

Приведем еще один пример сортировки двух чисел - по возрастанию последней цифры числа, а если последние цифры равны - то по убыванию самих чисел.

**bool cmp(int a, int b)**

**{**

**return a % 10 < b % 10 || a % 10 == b % 10 && a > b;**

**}**

Заметим также, что в функцию-компаратор лучше передавать объекты не по значению, а по ссылке, в этом случае они не будут копироваться (что может занимать значительное время при передачи крупных объектов, например, строк, векторов и т.д.). Тогда функцию нужно объявлять так:

**bool cmp(const int & a, const int & b);**

**Лямбда-функции (С++11)**

В стандарте языка С++11 появились лямбда-функции - безымянные функции, которые не получают собственного имени, а используются только в месте объявления.

Распространенное использование лямбда-функций - это параметр-компаратор при сортировке. Вместо создания специальной функции-компаратора, можно описать функцию прямо в вызове функции sort.

Пример - отсортируем

**vector<int> a**

 по последней цифре числа:

**sort(a.begin(), a.end(), [](int x, int y) -> bool {return x % 10 < y % 10;});**

**Оператор "меньше" для классов и структур**

Если сортировка используется для каких-то классов или структур данных, то для них можно определить оператор "меньше", а не использовать функцию-компаратор. Тогда сортировка (если не задать компаратор) для сравнения будет использовать определенный оператор "меньше".

Например, рассмотрим структур fraction для хранения рациональных дробей, для простоты ограничимся только положительными дробями. Дроби будут храниться в виде двух целых чисел - числителя и знаменателя, поэтому у нашей структуры будут два поля: a и b.

Стуктура fraction определяется так:

**struct fraction {**

**int a, b;**

**}**

Есть разные способы определения для собственных структур и классов арифметических функций, покажем только один пример: функция с названием operator<, получающая на вход два аргумента (соответствующие левому и правому операнду, поэтому назовем их left и right), возвращающая значение типа bool. Операнды будем передавать по константным ссылкам:

**bool operator< (const fraction & left, const fraction & right) {**

**return left.a \* right.b < left.b \* right.a;**

**}**

**Структура pair**

В библиотеке STL есть шаблон класса

**pair**

.

Класс

**pair**

 - это два значения, то есть "пара". У объектов этого класса два поля, первое называется

**first**

, второе называется

**second**

. Например, класс

**pair**

 можно использовать для хранения точек плоскости (точка - это две координаты) или рациональных дробей (дробь - два числа).

Один экземпляр объектов класса

**pair**

 определяется так:

**pair <int, int> p;**

Теперь p — это структура с двумя полями, типа

**int**

 каждое, им можно присваивать значения:

**p.first = 1;**

**p.second = 2;**

Можно сразу присвоить значение "паре" целиком, здесь может оказаться полезным функция

**make\_pair**

, у которой два аргумента, и которая возвращает объект класса

**pair**

, поля которого равны двум аргументам. Например:

**p = make\_pair(1, 2);**

Можно создавать массивы и вектора из

**pair**

, например:

**vector<pair<int, int> > a;**

**pair**

 сортируются в лексикографическом порядке, то есть сначала они упорядочиваются по значению поля

**first**

, а при равном значении поля

**first**

 — по значению поля

**second**

. Поэтому для решения задачи сортировки, например, чисел по последней цифре можно создать

**pair**

, у которой поле

**first**

 будет равно последней цифре числа, а поле

**second**

 - самому числу. Рассмотрим два примера реализации считывания и создания такого массива:

**int n;**

**cin >> n;**

**vector<pair<int, int> > a(n);**

**for (int i = 0; i < n; ++i) {**

**cin >> a[i].second;**

**a[i].first = a[i].second % 10;**

**}**

А в следующем примере будем использовать функцию

**make\_pair**

 для создания пары и добавления ее в конец вектора:

**int n;**

**cin >> n;**

**vector<pair<int, int> > a;**

**for (int i = 0; i < n; ++i) {**

**int num;**

**cin >> num;**

**a.push\_back(make\_pair(n % 10, n));**

**}**

Элементами пары могут быть объекты разных типов, не только числа.

Другое типичное применение

**pair**

 в сортировке - сортировка данных с сохранением информации об их порядке. Например, пусть дана последовательность строк, их нужно отсортировать по алфавиту, но нужно для каждой строки запомнить ее номер во входных данных. В этом случае нужно использовать

**pair**

, у которой первое поле - строка, а второе поле - число, в котором будет храниться номер:

**int n;**

**cin >> n;**

**vector<pair<string, int> > a(n);**

**for (int i = 0; i < n; ++i) {**

**cin >> a[i].first;**

**a[i].second = i;**

**}**

**Структура tuple**

Допустим, в одной переменной нужно хранить не два, а большее число полей. Например, информация о человеке может содержать поля имя, фамилия, возраст, и необходимо как-то сортировать эти данные.

Возможные варианты решения:

1. Создание собственной структуры данных с указанием нужных полей. Например,

**struct person {**

**string lastname;**

**string firstname;**

**int age;**

**}**

В этом случае придется объявлять саму структуру, затем реализовывать операцию сравнения, что может занимать достаточно много кода.

2. Использовать pair, один из элементов которого также является pair.

Например, можно объявить переменную так:

**pair <string, pair<string, int> > person;**

**person.first = lastname;**

**person.second.first = firstname;**

**person.second.second = age;**

Это не требует объявления структуры, но достаточно неудобно обращаться к полям структуры через конструкции вида

**.second.first**

.

Начиная со стандарта C++11 в STL есть класс

**tuple**

 (кортеж), который предоставляет возможность создавать аналоги pair из любого количества полей. Например, для представления класса из трех полей типа

**string**

,

**string**

,

**int**

 можно объявить класс следующим образом:

**tuple <string, string, int> p;**

Для доступа к полям

**tuple**

 используется конструкция

**get**

 следующим образом:

**get<0>(p) = lastname;**

**get<1>(p) = firstname;**

**get<2>(p) = age;**

Параметр, передаваемый функции

**get**

 в угловых скобках — это номер поля, индексация начинается с нуля. Это значение должно быть константой, то есть определено на момент компиляции программы, нельзя в качестве этого значения использовать переменную.

Например, объявим класс

**person**

 как

**tuple**

 из трех полей при помощи

**typedef**

 (для упрощения последующего использования)

**typedef tuple <string, string, int> person;**

**int n;**

**cin >> n;**

**vector <person> p(n);**

**for (int i = 0; i < n; ++i) {**

**cin >> get<0>(p[i]) >> get<1>(p[i]) >> get<2>(p[i]);**

**}**

Или можно использовать функцию

**make\_tuple**

, аналогичную make\_pair:

**typedef tuple <string, string, int> person;**

**int n;**

**cin >> n;**

**vector <person> p;**

**for (int i = 0; i < n; ++i) {**

**string lastname, firstname;**

**int age;**

**cin >> lastname >> firsstname >> age;**

**p.push\_back(make\_tuple(lastname, firstname, age));**

**}**

Tuple сортируются также в лексикографическом порядке — сначала по первому полю, при равенстве первого поля — по второму, затем по третьему и т.д.

**Универсальная инициализация tuple (С++17)**

В стандарте C++17 появилась удобная возможность работы с tuple без использования функции get, если использовать универсальную инициализацию.

Присвоить элементу вектора типа tuple новое значение можно так:

**p[i] = {lastname, firstname, age};**

А распаковать значения типа tuple можно в три переменные следующим образом:

**auto [lastname, firstname, age] = p[i];**

Пример перебора всех значений вектора с распаковкой в три переменные:

**for (auto [lastname, firstname, age]: p)**

**{**

**cout << lastname << " " << firstname << " " << age << endl;**