Лекция XIII

6 мая 2017

Итераторы, или как переберать элементы в контейнерах с произвольным доступом

Итераторы

Уже было: динамический массив через **vector**

```
1
2 vector<double> real_arr;
3
4 for (usingned i = 0; i < 15; ++i) {
5   real_arr.push( sqrt(i) );
6 }
7
8 cout << real_arr[7] << '\n';
```

Статический массив фиксированного размера: **array**

```cpp
#include <array> // Здесь определяется класс

array<int, 10> tenth_arr;

// Обращение по индексу
for (unsigned i = 0; i < 10; ++i) {
  tenth_arr[i] = 10 * i + 5;
}
cout << tenth_arr[0] << '\n';

// Создание массива и инициализация начальными значениями
array<double, 5> fifth_real_arr = { 1.5, 2.77, 12.88 };
cout << fifth_real_arr[2] << '\n';
cout << fifth_real_arr[4] << '\n';
```

Итераторы

Статический массив фиксированного размера: **array**. Чем полезен?

```
1  void inc_by_2(int arr[5])
2  {
3    for (unsigned i = 0; i < 5; ++i) {
4      arr[i] += 2;
5    }
6  }
7
8  void inc_by_2(array<int, 5> arr)
9  {
10   for (int& elem : arr) { elem += 2; }
11 }
12
13 int arr1[] = {1, 2, 3, 4, 5}, arr2[] = {5, 6, 7};
14 array<int, 5> arr3 = {4, 6, 8, 10, 12};
15 array<int, 3> arr4 = { 1, 2, 3 };
16
17 inc_by_2(arr1);
18 inc_by_2(arr3);
19
20 // inc_by_2(arr4); // Даже не компилируется
21 inc_by_2(arr2); // Опасность!
```

Итераторы

Итератор - специализированный *объект* для данного контейнера, который умеет обходить **все элементы контейнера** единственным образом. Стандартные итераторы ведут себя подобно указателям, а именно:

- к итератору применима операция **инкремента**(++it) для перехода с **текущего** на **следующий** элемент;
- к итератору применима операция **декремента**(–it) для перехода с **текущего** на **предыдущий** элемент;
- к итератору применима операция **инкремента**(++it) для перехода с **текущего** на **следующий** элемент;
- к итератору можно **добавить целое число** n для перхода вперёд/назад (it + n, it - n) по элементам контейнера;
- для получения значения элемента, на который указывает итератор, применяется операция **разыменования**(*it).

Итераторы

Кроме того,

- библиотека контейнера предоставляет класс итератора (как правило через его класс);

```
1 vector<double> my_vec = { 1.5, 3.5, 5.5 };
2 // Итератор:
3 vector<double>::iterator vec_it;
```

- контейнер предоставляет метод для получения итератора, который указывает на его первый элемент;

```
4 vec_it = my_vec.begin();
5 // Или так:
6 vec_it = begin( my_vec );
```

- контейнер предоставляет метод, возращающий специальное значение, которое показывает, что достигнут конец контейнера (перебраны все элементы)

```
6 if ( vec_it != my_vec.end() ) {
7   cout << "Не все элементы перебрали\n";
8 }
```

Итераторы

Канонический пример на использование итераторов

```
1 vector<int> my_vec = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
2 vector<int>::iterator vec_it;
3
4 for (vec_it = my_vec.begin();
5      vec_it != my_vec.end(); ++vec_it) {
6   cout << " " << *vec_it;
7 }
8 cout << '\n';
```

Итераторы

Канонический пример на использование итераторов

```
1 vector<int> my_vec = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
2 vector<int>::iterator vec_it;
3
4 for (vec_it = my_vec.begin();
5      vec_it != my_vec.end(); ++vec_it) {
6   cout << " " << *vec_it;
7 }
8 cout << '\n';
```

в современном C++ не так актуален

```
1 vector<int> my_vec2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
2
3 for (const int& elem : my_vec) {
4   cout << " " << elem;
5 }
6 cout << '\n';
```

Итераторы

Тем не менее польза остаётся: вставка элементов в произвольное место динамического массива **vector** с помощью метода **insert** :

```
1  vector<int> my_vec = { 1, 2, 3, 4, 5, 6, 7, 8, 9,↩
       10 };
2  // Ставим итератор на третий элемент
3  vector<int>::iterator vec_it = my_vec.begin() + ↩
       2;
4
5  /*
6    Здесь *insert* принимает два
7    аргумента: итератор позиции
8    элемента и новый элемент для вставки
9  */
10 my_vec.insert(vec_it, 25);
11 cout << my_vec[2] << endl; // Печатает: 25
```

Вставка элементов в произвольное место динамического массива **vector** с помощью метода **insert** :

```
1  vector<int> my_vec2 = { 1, 2, 3, 4, 5, 6, 7, 8, ↩
       9, 10 },
2               vec2   = { 6, 7, 8 };
3  // Ставим итератор на третий элемент
4  vector<int>::iterator vec_it2 = my_vec2.begin() +↩
       2;
5
6  /*
7    А ещё *insert* принимает
8    умеет вставлять массивы
9    друг в друга
10 */
11 my_vec2.insert(vec_it2, vec2.begin(), vec2.end())↩
       ;
12 cout << my_vec2[3] << endl; // Печатает: 7
```

Итераторы

Более-менее полезный пример: добавление элементов в **vector** по возрастанию

```
1  void add_elem_asc_order(vector<int>& vec, int value)
2  {
3    vector<int>::iterator it = vec.begin(), end_it = vec.end();
4    while ( (it != end_it) && (value > *it) ) {
5      ++it;
6    }
7    if (it == end_it) {
8      vec.push_back( value );
9    } else {
10     vec.insert(it, value);
11   }
12 }
13
14 vector<int> my_vec;
15 for (unsigned i = 0; i < 105; ++i) {
16   add_elem_asc_order( my_vec, rand() % 101 );
17 }
18
19 for (int elem : my_vec) { cout << " " << elem; }
20 cout << '\n';
```

Итераторы и алгоритмы

C++ представляет библиотеку <**algorithm**>, к которой, в частности, определён набор функция для обобщённой работы с контейнерами произвольного доступа. Начнём с сортировки

```
(1) void sort(it_begin, it_end);
    void stable_sort(it_begin, it_end);
(2) void sort(it_begin, it_end, comparator);
    void stable_sort(it_begin, it_end, comparator);
```

1. сортировка элементов в порядке **возрастания**, начиная с итератора **it_begin** и заканчивая **it_end**;
2. сортировка элементов, порядок следования которых определяется функцией **comparator**.

stable_sort отличается от **sort** только тем, что *сохраняет порядок следования одинаковых по значению элементов*. Если это не важно (большинство вычислительных задач) - лучше выбирать **sort**, она не медленнее альтернативного варианта.

Пример на сортировку

```
 1 #include <algorithm>
 2
 3 bool compare_real_as_int_desc(double a, double b)
 4 { return int(a) > int(b); }
 5
 6 vector<double> my_reals = { 2.34, 1.555, 0.764, 5.67, 1.9,
 7                             1.88, 8.11, 4.56, 5.24, 4.5 };
 8
 9 std::sort(my_reals.begin(), my_reals.end(),
10                  compare_real_as_int_desc);
11 for (double elem : my_reals) {
12   cout << " " << elem;
13 }
14 cout << '\n';
15
16 // В качестве итераторов принимаются и стандратные указатели!
17 double simpl_arr[] = { 4.56, 1.23, 9.9, 8.8, 0.74 };
18 std::sort(simpl_arr, simpl_arr + 5);
19 for (unsigned i = 0; i < 5; ++i) {
20   cout << " " << simpl_arr[i];
21 }
22 cout << '\n';
```

Итераторы и алгоритмы

Далее, стандартные функции для поиска элемента по значению

```
(3) iterator find(it_begin, it_end, value);
(4) iterator find_if(it_begin, it_end, predicate);

predicate => bool (*predicate)(Type elem);
```

1. поиск первого элемента, равного **value**, между **it_begin** и **it_end**;
2. поиск первого элемента, между **it_begin** и **it_end** **comparator**, которого функция **predicate** вернула значение **true**.

Обе функции возвращают **итератор** на найденный элемент.

Пример на поиск

```
1  bool is_even(int elem)
2  { return (elem % 2) == 0; }
3
4  vector<int> my_ints = { 1, 3, 7, 15, 8,
5                          88, 11, 6, 4, 9 };
6
7  vector<int>::iterator found;
8  found = std::find(my_ints.begin(), my_ints.end(), 88);
9  if (found != my_ints.end()) {
10   cout << "число 88 нашлось в массиве\n";
11 }
12
13 found = std::find_if(my_ints.begin(), my_ints.end(), is_even);
14 if (found != my_ints.end()) {
15   cout << "первое чётное число: " << *found << '\n';
16 }
```

Тёмная сторона итераторов

Языком C++ не гарантируется никакое определённое поведение при выходе за границы контейнера

```
1 vector<int> my_vec5 = { 1, 3 };
2 vector<int>::iterator it5 = my_vec5.begin();
3 it5 += 3;
4 // Неопределено, вылетит программа с ошибкой
5 // на следующей строке, либо напечатает что-то
6 cout << "Непонятно что: "<< *it5 << '\n';
```

Итераторы и алгоритмы

В качестве резюме по итераторам:

- Контейнеры с произвольным доступом: динамический массив (<**vector**>), ассоциативные массивы (<**unordered_map**>, <**map**>, статический массив (<**array**>), множества (<**set**>, <**unordered_set**>), динамические списки (<**forward_list**> - однонаправленный список, <**list**> - двухнаправленный список).
- Контейнеры не поддерживающие итераторы: <**stack**>, <**queue**>, <**deque**.
- Функции из <**algorithm**>: выполнение заданного действия для каждого элемента - **for_each**; подсчёт элементов по значению - **count** и по заданной проверке - **count_if**.

P.S. а ещё итератор определён для класса **string**.

Автоматический вывод типа переменной компилятором, или когда хочется набирать меньше символов (только для C++)

auto для вывода типов

Компилятор C++ достаточно умён. Для него не проблема во многих случаях вывести тип переменной самостоятельно, без его явного указания. Но для этого переменной нужно обязательно **присвоить значение**. А вместо **типа данных** написать ключевое слово **auto**

```
1 auto int_var = 15;
2 // int_var становится переменной типа int
3
4 auto real_var = 15.6;
5 // int_var — переменной типа double
6
7 auto str = "Какая-то строка";
8 // str получает тип const char *
```

auto для вывода типов

Но для простых типов данных применение **auto** при определении переменных **строго не рекомендуется**. Когда же есть польза?

Первый пример: итераторы

```
1 // Как уже знаем:
2 vector<double> my_vec6 = { 1.5, 3.77 };
3 vector<double>::iterator it6 = my_vec5.begin();
4
5 // Можно записать так:
6 vector<double> my_vec6 = { 1.5, 3.77 };
7 auto it6 = my_vec5.begin();
```

Для стандартных контейнеров итераторы реализованы однообразно, следовательно вызов метода **begin** даёт ожидаемый результат. Указание типа перекладываем на компилятор.

auto для вывода типов

Второй пример: **for-range**

```
1  vector<double> my_vec7 = { 1.5, 3.77, 4.8, 7.9, ↩
       10.11 };
2
3  // Было:
4  for (double& elem : my_vec7) {
5    elem *= elem;
6  }
7
8  // Стало:
9  for (auto& elem : my_vec7) {
10   elem *= elem;
11 }
```

auto для вывода типов

Главный пример: динамическое создание объектов класса

```
1  class MySuperPoint2Dimension
2  {
3  public:
4    Point2D(int x_, int y_) : x{x_}, y{y_}
5    {}
6
7  private:
8    int x, y
9  };
10
11 // Для new нужен конструктор!
12 MySuperPoint2Dimension *p1 = new ↩
       MySuperPoint2Dimension(1, 5);
13
14
15 // С auto будет короче:
16 auto *p1 = new MySuperPoint2Dimension(1, 5);
```

Работа со временем, или контролируй каждую секунду.

Функции из <ctime>

Для базовых операций со временем в C++ проще пользоваться стандартной библиотекой языка C. Она включается:

```
1 // язык C++
2 #include <ctime>
3
4 // язык C
5 #include <time.h>
```

Функции из <ctime>

Первая операция: подсчёт времени выполнения участка кода. Для этого <**ctime**> предоставляет:

- тип данных **clock_t** представляющий собой **целочисленный** тип данных (зависящий от ОС);
- функцию **clock()**

```
1 clock_t clock();
```

, которая не принимает никаких аргументов и возращает **количество тактов**, прошедших с начала выполнения программы;

- ОС-зависимую константу **CLOCKS_PER_SEC** - хранящую количество тактов, выполняющихся за секунду (говоря языком физики, размерность этой константы есть - число_тактов / секунду).

Функции из <ctime>

Первая операция: подсчёт времени выполнения участка кода.

```
1  unsigned count_of_primes(unsigned n)
2  {
3    unsigned count = n-1;
4    for (unsigned i = 2; i <= n; ++i) {
5      for (unsigned j = sqrt(i); j > 1; --j) {
6        if ( (i % j) == 0 ) { --count; break; }
7      }
8    }
9
10   return count;
11 }
12
13 cout << "Вычисляем...\n";
14 clock_t cl_val = clock();
15 cout << "Количество простых чисел "
16         "меньше 1000000 равно " << count_of_primes(1000000)
17      << '\n';
18 cl_val = clock() - cl_val;
19 double sec_spent = double(cl_val) / CLOCKS_PER_SEC;
20 cout << "Вычисление произведено за " << cl_val
21      << " тактов (" << sec_spent
22      << " секунд)";
```

Функции из <ctime> I

Вторая операция: отображение времени в понятном виде. Для этого <**ctime**> предоставляет:

- тип данных **time_t** представляющий собой **целочисленный** тип данных (зависящий от ОС);
- структуру **tm**

```
1  struct tm {
2    int tm_sec,   // секунды,      0–59
3    int tm_min,   // минуты,       0–59
4    int tm_hour,  // часы,         0–23
5    int tm_mday,  // день месяца,  1–31
6    int tm_mon,   // месяц,        0–11
7    int tm_year,  // год,        от 1900
8    int tm_wday,  // день недели   0–6
9    int tm_yday,  // день с начала года 0–365
10   int tm_isdst, // спец. флаг
11 };
```

, которая используется в функциях для представления времени (не должна использоваться напрямую);

Функции из <ctime> II

- функцию **time**

```
1 time_t time(time_t *timer);
```

, интересную тем, что она и возвращает, и присваивает указателю **timer** одно и тоже значение: число секунд, прошедших с некоторого рубежа (на данный момент - "00:00:00 1 января 1970");

- две функции - **gmtime** и **localtime**

```
1 tm * gmtime(const time_t *timer);
2
3 tm * localtime(const time_t *timer);
```

, преобразующих время **timer** в объект структуры **tm** и возвращающих указатель-на-структуру. Различаются тем, что первая их них считает текущим часовым поясом UTC, а вторая - берёт локальный часовой пояс согласно настройкам ОС;

Функции из <ctime> III

- функцию **mktime**

```
1 time_t mktime(tm *tm_ptr);
```

, осуществляющею обратное преобразование: из объекта структуры **tm** в число секунд с укащанного выше рубежа;

- функцию **difftime**

```
1 double difftime(time_t end, time_t start);
```

, которая возращает число секунд между моментами **end** и **start**;

Функции из <ctime>

Отдельно стоит рассмотреть функцию **strftime**. Она позволяет представить время в человеко-читаемом формате.

```
1 size_t strftime(char *str_to_save, size_t max_size,
2                 const char *format, const tm *tm_ptr);
```

, где

- **str_to_save** - строка в стиле C, в которую будет записан результат;
- **max_size** - максимальное количество символов, которое попадёт в **str_to_save**;
- **format** - специальная строка в стиле C, задающая правила преобразования времени в строковое представление;
- **tm_ptr** - указатель-на-объект структуры **tm**, представляющий собой момент времени для отображения.

Функции из <ctime>

Функция **strftime**. Специальные **%-последовательности** для преобразования времени (указываются в строке **format**)

	Что означает?	Пример значений
%a	Аббревиатура дня недели	Sun, Mon, Thu *
%A	Полное название дня недели	Sunday, Monday *
%b	Аббревиатура месяца	Aug, Jun, Jul *
%B	Название месяца	June, Jule, March *
%d	Номер дня в месяце	01-31
%H	Час в 24-формате	00-23
%I	Час в 12-формате	01-12
%M	Минута	00-59
%S	Секунда	00-59
%p	Знак 12- или 24-часового формата	AM, PM
%m	Номер месяца	01-12
%W	Номер недели (с понедельника)	00-53
%y	Год, две последние цифры	00-99
%Y	Год	2017
%z	Смещение часовой зоны	+0600
%Z	Аббревиатура временной зоны	UTC *

* в третьем столбке означает зависимость от настроек локали в ОС

Функции из <ctime>

Вторая операция: отображение времени в понятном виде.

Примеры

```
1  time_t now = time( nullptr );
2  tm *tm_ptr = localtime( &now );
3  char buf[70];
4
5  strftime(buf, 70, "%I:%M:%S %p", tm_ptr);
6  cout << "12-часовое представление: " << buf << '\n';
7
8  strftime(buf, 70, "%H:%M:%S", tm_ptr);
9  cout << "24-часовое представление: " << buf << '\n';
10
11 strftime(buf, 70, "%d/%m/%y", tm_ptr);
12 cout << "ДД/ММ/ГГ: " << buf << '\n';
13
14 strftime(buf, 70, "%d/%m/%Y", tm_ptr);
15 cout << "ДД/ММ/ГГГГ: " << buf << '\n';
16
17 strftime(buf, 70, "Дата: %d/%m/%Y, %a %b", tm_ptr);
18 cout << buf << '\n';
19
20 string date_str = buf;
21 cout << date_str << '\n';
```