مقدمه، تعریف

شاید تا به حال برایتان اتفاق افتاده باشد که بخواهید به دلایلی آرایه‌ای با طول متغیّر داشته باشید. در این موارد vector می‌تواند زندگیتان را نجات بدهد! با اینکه vector در اصل یک متغیّر معمولی است، اما نحوه‌ی استفاده از آن شباهت زیادی به آرایه دارد.

عناصری که درون vector هستند، همگی می‌بایست از یک نوع باشند. برای تعریف vector، بعد از اضافه کردن کتابخانه‌ی مربوطه با افزودن دستور `#include <vector>` به ابتدای فایل، می‌نویسیم:

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

بجای int نام هر نوعی از متغیرها (مثلاً: double یا pair<int, char> یا حتی vector<int>) می‌تواند قرار بگیرد.

پس از تعریف vector، آرایه‌ی مورد نظر به صورت پیش‌فرض خالی است و ما می‌توانیم با دستورهایی به آن عناصری را اضافه یا کم کنیم، و یا به بعضی از خانه‌های آن دسترسی داشته باشیم.

دسترسی به خانه‌های vector شباهت بسیاری به دسترسی به خانه‌های آرایه دارد. به این صورت که v[i] محتوای خانه‌ی iم است. (باز هم مثل آرایه: اندیس خانه‌های vector از صفر شروع می‌شوند.) دقت کنید که v[i] حتماً باید وجود داشته باشد در غیر این صورت با پیغام خطا مواجه خواهید شد.

توابع پرکاربرد

برخی توابع برای کار با vector: (فرض کنیم v یک vector است که تا به حال n عنصر را در خود جای داده)

- v.size()
  مقدار بازگشتی این تابع، تعداد عناصر درون آرایه (n) خواهد بود.
- v.push_back(x)
  این دستور عنصر x را به انتهای آرایه اضافه می‌کند. با این کار، طول آرایه یک واحد افزایش می‌یابد و v[n] برابر x خواهد بود.
- v.pop_back()
  این دستور، آخرین عضو آرایه (یعنی v[n-1]) را از vector حذف می‌کند. با این کار طول آرایه یک واحد کاهش می‌یابد.
- v.clear()
  با فراخوانی این تابع، تمام عناصر حذف خواهند شد و یک vector خالی خواهیم داشت.
- v.front() و v.back()
  back همان عنصر انتهای آرایه (v[n-1]) خواهد بود و front همان عنصر ابتدایی (v[0])
- v.resize(d)
  پس از اجرای این دستور طول آرایه برابر d خواهد بود. برای این کار، تعداد کافی از عناصر به انتها اضافه و یا از انتها حذف می‌شوند.

iterator چیست؟

در آموزش توابع کتابخانه‌ی **algorithm** با توابعی مانند sort آشنا شدید که برای کار کردن، دو اشاره‌گر[1] (یکی به ابتدا و دیگری به انتهای آرایه) دریافت می‌کردند و بر روی آن عملیاتی را انجام می‌دادند. برای مثال، در مورد مرتب کردن آرایه‌ی arr به طول n، از دستور زیر استفاده می‌کردیم:

```
sort(arr, arr+n);
```

اما اگر vectorی به نام v و به طول n داشته باشید، با بکار بردن دستور بالا با خطای کامپایل مواجه می‌شوید:

```
sort(v, v+n); //Compilation error
```

دلیل آن نیز واضح است. تابع sort انتظار دریافت دو اشاره‌گر دارد. معادل pointer برای داده‌ساختارهایی که در STL تعریف شده‌اند، iterator است. با آشنایی بیشتر در مورد داده‌ساختارهای متفاوت، با iterator نیز بیشتر آشنا خواهید شد. برای مثال به نحوه‌ی درست استفاده از تابع sort برای مرتب‌سازی عناصر یک vector دقت کنید:

```
sort(v.begin(), v.end());
```

دقت کنید که v.end() دقیقاً معادل v.begin()+n است و صرفاً برای راحتی از end() استفاده می‌کنیم.

تحلیل پیچیدگی: (n تعداد عناصر درون vector است)

تحلیل زمانی: در این مورد نیز vector شبیه آرایه است. اضافه و حذف کردن عضو از انتهای vector از $O(1)$ است. همین‌طور اضافه یا حذف کردن عضو در ابتدا و یا اواسط آن از $O(n)$ خواهد بود. (این کار با توابعی مانند insert و erase انجام می‌پذیرد، که به دلیل کم‌کاربرد بودن از توضیح آنها صرف‌نظر کردیم.)

حافظه‌ی مصرفی: vector از $O(n)$ حافظه مصرف می‌کند. در صورتی که نحوه‌ی کار vector را بررسی کنید، متوجه خواهید شد که مقدار دقیق حافظه‌ی اشغال‌شده، حداکثر ۲ برابر میزان حافظه‌ای است که توسط آرایه‌ای به طول n اشغال می‌شود.

مثال: کاربرد در ذخیره‌سازی گراف

فرض کنید می‌خواهید یک گراف با V رأس و E یال (ساده، با طوقه، بدون یال چندگانه) را در حافظه ذخیره کنید. برای این کار روش‌های مختلفی وجود دارد (ر.ک. کتاب‌های آموزشی الگوریتم) یکی از این روش‌ها، نگه داشتن لیست مجاورت رئوس است. پیاده‌سازی این روش را ابتدا با آرایه، و سپس vector بررسی می‌کنیم:

- **پیاده‌سازی با آرایه‌ی دوبعدی:**

  یک آرایه‌ی دو بعدی نگه می‌داریم، به صورتی که a[i][j] نشان‌دهنده‌ی j –مین همسایه‌ی رأس i است.

  از آنجا که تعداد همسایه‌های هر رأس حداکثر می‌تواند V باشد، و دقیقاً V رأس داریم، برای پیاده‌سازی این روش به یک آرایه‌ی $V \times V$ نیاز داریم. پس $O(V^2)$ حافظه مصرف می‌کنیم.

[1] Pointer

- **پیاده‌سازی با آرایه‌ای از** `vector`:

یک آرایه‌ی V تایی از `vector` نگه می‌داریم. عناصر درون iمین `vector`، نشان‌دهنده‌ی همسایه‌های رأس `i` خواهد بود. چون هر یال دقیقاً باعث اضافه شدن دو عنصر به تعداد کل عناصر است، حافظه‌ی مصرفی $O(E)$ خواهد بود. (در واقع درست‌تر است که بگوییم $O(V+E)$ چرا؟)

به همین دلیل در اکثر مواد (به خصوص در مورد گراف‌های کم‌پشت[2] مانند درخت‌ها) برای ذخیره‌سازی گراف به صورت لیست مجاورت از `vector` استفاده می‌کنیم.

تمرین:

۱. خروجی کد زیر چیست؟ بعد از حدس زدن، برنامه را اجرا کنید و نتیجه را با حدستان مقایسه کنید.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v;
    for(int i=0; i*i<=16; i++)
        v.push_back(i*i);
    for(int i=0; i<v.size(); i+=2)
        v.pop_back();
    cout << v.size() - v.back() << endl;
    return 0;
}
```

۲. بعد از تفکر (و اگر به نتیجه نرسیدید: مطالعه) در مورد اینکه `vector` چگونه کار می‌کند:

a. گفتیم که اضافه کردن عضو به انتهای `vector` از $O(1)$ است. در واقع، اینطور نیست. درست‌تر بود که می‌گفتیم: «اضافه کردن عضو به انتها، سرشکن از $O(1)$ است.» در مورد تحلیل سرشکن[3] الگوریتم‌ها مطالعه کنید. تحلیل زمانی توابع `vector` یک نمونه‌ی خیلی خوب از تحلیل سرشکن است.

b. شاید قبلاً تابع `swap` را دیده باشید. این تابع که به صورت `swap(a, b);` نوشته می‌شود، محتوای متغیرهای `a` و `b` را عوض می‌کند و در حالت عادی به اندازه‌ی کپی کردن کامل محتوای هر دو متغیر زمان می‌برد. اما در مورد داده‌ساختارهای STL، این تابع همواره در $O(1)$ عمل می‌کند. این اتفاق چگونه ممکن است؟

c. `deque` (بخوانید: دِک) نام داده‌ساختار دیگری است که در STL وجود دارد. این داده‌ساختار علاوه بر تمام خواص `vector`، حذف و اضافه‌ی یک عنصر به ابتدای آرایه را نیز در $O(1)$ انجام می‌دهد. در مورد چگونگی کارکرد این داده‌ساختار نیز فکر (مطالعه) کنید.

---

[2] Sparse Graph
[3] Amortized Analysis