جلسه اول – مقدمه درس

کارایی برنامه:

در درس الگوریتم و ساختمان داده مورد بررسی قرار می‌گیرد. با موازی سازی تا حدی می توان کارایی را افزایش داد.

خوانایی برنامه (Readability)

کاربردپذیری

نگهداری برنامه (Maintenance)

امنیت برنامه

مقیاس‌پذیری (scalability) که در درس مهندسی اینترنت مورد بررسی است.

درستی عملکرد

کتابخانه‌های آماده

قابلیت تغییر برنامه

به عنوان مثال استفاده از global variable چندان توصیه نمی شود.

دستور Go to که در زبان های قدیمی تر موجود بود و کنترل اجرای برنامه را پیچیده می کرد.

تست نرم افزار را برای لحظه آخر نگه ندارید.

تاکید درس: نوشتن برنامه های خوب

Correctness: Testing, Debugging

Maintainability: Object orientation, coding style

Reusability: object orientation

از قطعات نرم‌افزار بتوانیم مجددا استفاده کنیم.

زبان اصلی: C++

JAVA هم کاربردی است.

زبان ها: C++، C#، Python، MATLAB، JAVA، R

C++ کارایی C (تاکید روی سرعت و استفاده بهینه از منابع) را با شی‌گرایی و انعطاف پذیری ترکیب کرده است.

کتاب:

Deitel: C++ How to program

Google Search

Stackoverflow.com

Cplusplus.com

Ramtung.ir/apnotes/html

جلسه دوم – مفاهیم مقدماتی زبان C++

سلام دنیا: یک برنامه‌ی بسیار ساده در زبان سی‌پلاس‌پلاس که صرفاً یک رشته را در خروجی می‌نویسد.

```
1. #include <iostream>
2. using namespace std;
3.
4. int main()
5. {
6.       cout << "Hello World\n";
7.       return 0;
8. }
```

Iostream هدرفایلی است که امکانات ورودی و خروجی C++ را در اختیار ما می‌گذارد.

قاعده قرار دادن h. قدیمی است و تقریبا منسوخ شده است.

Std مخفف استاندارد است و تمام امکانات استاندارد C++ در این فضای نام قرار دارد.

فضاهای نام برای پرهیز از تداخل بین کتابخانه‌های نرم افزاری مختلف مورد استفاده قرار می‌گیرند.

اگر خط دوم را ننویسم بایستی هربار بنویسیم std::cout

امصا تابع: نام تابع، نوع پارامتر ورودی، نوع پارامتر خروجی

در مورد خط چهارم فعلا بدانیم که امکان پاس کردن آرگومان به تابع main از طریق command line وجود دارد.

در C++ بلافاصله بعد از امضای تابع و در خط بعدی، آکلاد را باز می کنیم. این کار در جاوا درست بعد از امضای تابع و در همان خط انجام می گیرد.

در داخل بدنه‌ی تابع، یک تب (TAB) قرار می‌دهیم.

Cout مخفف console out است که اصطلاح اطلاق شده به همان ترمینال یا command line می باشد.

G++ کامپایلر C++ در لینوکس. فایل اجرایی در لینوکس پسوند out. و در ویندوز پسوند exe. دارد.

بازگرداندن صفر اطمینان از موفقیت آمیز بودن اجرای برنامه را تضمین می کند.

تایپ رشته – خواندن از ورودی: این مثال استفاده اولیه از تایپ رشته و نحوه‌ی خواندن از ورودی را نشان می‌دهد.

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

int main()
{
      string name;
      cout << "Please enter your name: ";
      cin >> name;
           cout << "Hello " << name << '\n';
           return 0;
      }
```

string هدرفایلی است که امکان کار روی رشته‌ها را به ما می‌دهد.

Name شی (object) از نوع string است.

تابع cin فقط کلمه اول از ورودی را می گیرد. کلمه دنباله ای از کاراکترها است که به فضای سفید محدود می‌شود.

فضای سفید: space، TAB، Enter

Cin کاراکترهای غیر فضای سفید را می‌خواند.

خواندن کل یک خط با تابع getline انجام می‌گیرد.

خواندن چند قلم از ورودی: در این مثال چند مقدار به دنبال هم از ورودی خوانده می‌شود. دقت کنید که اگر رشته‌ای از cin خوانده شود، یک کلمه از ورودی خوانده شده در آن متغیر قرار می‌گیرد. مثلاً اگر ورودی Gholam 29 در ورودی تایپ شود مقدار name برابر Gholam و مقدار age برابر ۲۹ خواهد بود.

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

int main()
{
	string name;
	int age;
	cout << "Please enter your name followed by your age: ";
		cin >> name >> age;
		cout << "Hello " << name << "!\n";
		cout << "Your age is " << age << endl;
	}
```

Endl ثابتی که در هدرفایل iostream تعریف شده است و با کاراکتر ‘\n’ مترادف است.

تایپ ده انگشتی

چنانچه return 0 را برنگردانید، C++ خود به صورت پیش‌فرض مقدار صفر را برمی‌گرداند.

البته این ویژگی چندان خوب نیست. لذا سعی کنید همواره بصورت explicit مقدار صفر را برگردانید و چنانچه تابع قرار است مقدار خروجی نداشته باشد، نوع خروجی را void تعریف کنید.

خواندن از ورودی در حلقه: این برنامه تعدادی کلمه را از ورودی می‌خواند و در صورتی که کلمه‌ای تکرار شود این موضوع را با نمایش پیغامی اطلاع می‌دهد.

خواندن ورودی تا آنجا ادامه می‌یابد که کاربر با ctrl-d (یا ctrl-z در ویندوز) خاتمه ورودی را مشخص کند.

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

int main()
{
	string previous = "";
	string current;
	while (cin >> current) {
		if (previous == current)
			cout << "repeated word: " << current << '\n';
		previous = current;
	}
	}
```

در برنامه بالا تا زمانی که اینتر نزنیم، C++ به خواندن از ورودی ادامه می‌هد و پس از آن پردازش خط بعد را آغاز می‌کند.

خواندن ورودی از فایل، پاس کردن به فایل اجرایی در لینوکس و نوشتن خروجی در فایل

a.out<masalan.txt>output.txt

نمایش مختویات فایل text در لینوکس

Cat output.txt

پاس کردن خروجی برنامه به تابع sort لینوکس (piping in linux)

a.out<masalan.txt|sort

از مشکلات آرایه این است که اندازه آن باید ثابت باشد. جهت حل این مشکل کتابخانه استاندارد C++ وکتورها را در اختیار ما قرار می دهد. تایپ vector جزئی از کتابخانه‌ی استاندارد سی‌پلاس‌پلاس است که کار با دنباله‌ای از عناصر را **بدون دغدغه‌های مدیریت حافظه** مهیا می‌کند.

محاسبه‌ی میانگین و میانه: در این برنامه تعدادی عدد اعشاری (که نماینده‌ی دما هستند) از ورودی گرفته می‌شود و میانگین و میانه‌ی آنها در خروجی نوشته می‌شود. تعداد این اعداد در زمان نوشتن برنامه نامعلوم است.

```
1. #include <iostream>
2. #include <vector>
3. #include <algorithm>
4. using namespace std;
5.
6. int main()
7. {
8.       vector<double> temps;
9.       double temp;
10.      while (cin >> temp)
11.            temps.push_back(temp);
12.
13.      double sum = 0;
14.      for (int i = 0; i < temps.size(); ++i)
15.            sum += temps[i];
16.
17.      cout << "Mean temperature: " << sum/temps.size() << endl;
18.      sort(temps.begin(), temps.end());
19.      cout << "Median temperature: " << temps[temps.size()/2];
20.      }
```

در سر فایل algorithm عده‌ای از توابع همچون sort که در برنامه بالا در خط ۱۸ مورد استفاده قرار گرفته، تعریف شده‌اند.

مقداردهی اولیه‌ی بردارها: این مثال نشان‌دهنده‌ی مقداردهی اولیه‌ی اندازه و عناصر بردار است.

```
1. #include <vector>
2. #include <string>
3. #include <iostream>
4. using namespace std;
5.
6. int main()
7. {
8.       vector<double> vec;
9.       // vec[0] = 10;
10.      vec.push_back(1.3);
11.      vec[0] = 12.4;
12.
13.      vector<int> v(6);
14.
15.      v[0] = 5; v[1] = 7;
16.      v[2] = 9; v[3] = 4;
17.      v[4] = 6; v[5] = 8;
18.
19.      vector<string> philosopher(4);
20.
21.      philosopher [0] = "Kant";
22.      philosopher [1] = "Plato";
23.      philosopher [2] = "Hume";
24.      philosopher [3] = "Kierkegaard";
25.
26.      //philosopher[2] = 99;
27.
28.      vector<double> vd(1000, 1.2);
29.      //vd[1000] = 4.7;
30.      }
```

در مثال بالا اگر خط ۹ را کامنت نکنیم با RUNTIME ERROR مواجه خواهیم شد. زیرا در لحظه‌ی تعریف، vector شامل صفر عنصر است و با هر pushback حافظه تخصیص می‌یابد.

Segmentation fault

خطای دسترسی غیرمجاز به حافظه در لینوکس است.

در خط ۶ یک بردار ساخته‌ایم که دارای ۶ عنصر integer است و مقدار اولیه‌ی هریک از این عناصر صفر است.

خط ۲۶ با خطای کامپایلر روبرو می‌گردد. چون ۹۹ رشته نیست. اما "۹۹" یا ’۹۹‘ رشته است.

در خط ۲۸، برداری ساخته‌ایم که ۱۰۰۰ عنصر اولیه آن با مقدار اعشاری ۱/۲ پر شده است. خط ۲۹ هم با خطای segmentation (دسترسی غیرمجاز به حافظه) روبرو است.

بردار دو بعدی:

```
Vector<vector<int>>twod(3);
Twod[0].push_back(12);
Twod[1].push_back(4);
Twod[2].push_back(65);
Cout<< Twod[1][0]<<endl;
Vector<vector<int>>twod(3,vector<int>(4));
```

وکتوری با ۳ عنصر اولیه بساز و هرکدام از عناصر اولیه‌ی آن را با یک بردار چهارتایی از نوع مقدار صحیح، مقدار اولیه بده.

تمرین: برنامه‌ای بنویسید که تعدادی کلمه را از ورودی بخواند و آن‌ها را در قالب یک لیست که با کاما جدا شده‌اند چاپ کند. به عنوان مثال اگر کلمات ورودی به ترتیب Aang و Kyoshi و Roku باشند خروجی چاپ شده باید دقیقاً به صورت [Aang, Kyoshi, Roku] باشد. توجه داشته باشید که تعداد کلمه‌ها مشخص نیست و تا انتهای ورودی بخوانید.

```cpp
#include <iostream>
#include <sstream>
#include <string>

int main()
{
	std::string line;
	std::getline(std::cin, line);
	std::istringstream stream{ line };
	std::string s;
	stream >> s;
	std::cout << "[" << s;
	while (stream >> s)
	{
		std::cout << ", " << s;
	}
	std::cout << "]";
	std::cout << std::endl;
	system("pause");
	return 0;
}
```

خط اول سرفایل iostream را به سورس اضافه می‌کند. این سرفایل بخشی از کتابخانه ورودی/خروجی است.

خط دوم سرفایل sstream را به سورس اضافه می‌کند که آن نیز بخشی از کتابخانه ورودی/خروجی است.

سرفایل string بخشی از کتابخانه رشته‌ها است.

با قطعه کد

```cpp
std::getline(std::cin, line);
```

ورودی را تا زمانی که کاربر Enter بزند، خوانده و در line ذخیره می‌کنیم.

```
     istream& getline (istream&  is, string& str, char delim);
(1)  istream& getline (istream&& is, string& str, char delim);

(2)  istream& getline (istream&  is, string& str);
     istream& getline (istream&& is, string& str);
```

Get line from stream into string

Extracts characters from *is* and stores them into *str* until the delimitation character *delim* is found (or the newline character, `'\n'`, for *(2)*).

لذا وقتی کاربر Enter می‌زند، داریم:

Line = “Farnaz Farshid Shabnam Golnar”

سپس آنچه را در بافر ورودی است به نام stream می‌نامیم و با line مقدار دهی می‌کنیم.

std::istringstream stream{ line };

با کد

```
stream >> s;
```

کلمات (آنچه محصور بین دو فضای سفید است) را از بافر ورودی استخراج می‌کنیم و در s می‌ریزیم.

خطوط ۱۱ و ۱۲

stream >> s;

std::cout << "[" << s;

اجرا می‌شوند و داریم s = Farnaz و چاپ می‌شود:

[Farnaz

سپس وارد حلقه while می‌شویم. **چون کاراکتر << فضای سفید را ندید می‌گیرد**، مقداری که در شروع حلقه در s ذخیره است عبارت است از s=Farshid و فاصله بین Farnaz و Farshid ندید گرفته شده است. لذا به دنبال [Farnaz کاراکتر ’ ,‘ و بعد از آن Farshid نوشته می‌شود:

[Farnaz, Farshid

در دومین تکرار حلقه S=Shabnam و داریم:

[Farnaz, Farshid, Shabnam

و در سومین تکرار s=Golnar و داریم:

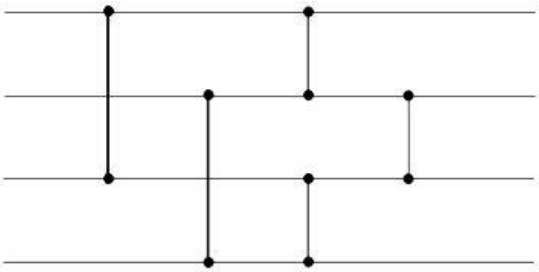
[Farnaz, Farshid, Shabnam, Golnar

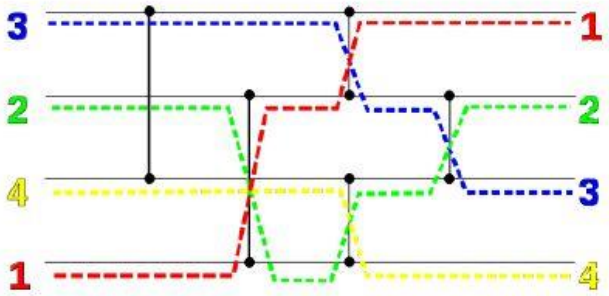
دیگر وارد حلقه نمی‌شویم. چون در جریان ورودی stream از نوع istringstream(input string stream) چیزی باقی نمانده است. از حلقه خارج شده و به دنبال مواد بالا، [نیز چاپ می‌شود.

[Farnaz, Farshid, Shabnam, Golnar]

جلسه سوم - طراحی بالا به پایین

شبکه‌های مرتب‌سازی و sorting network: شبکه‌ی مرتب‌سازی یک مدل انتزاعی ریاضی شامل شبکه‌ای از سیم‌ها و واحدهای مقایسه کننده است که برای مرتب‌سازی دنباله‌ای از اعداد ار آن استفاده می‌شود. هر مقایسه کننده دو سیم را به هم متصل می‌کند و مقادیر را با قرار دادن مقدار کوچکتر روی یکی از سیم‌ها و مقدار بزرگتر





روی سیم دیگر، مرتب می‌کند. هر سیم دارای یک مقدار می باشد، و هر مقایسه کننده دو سیم را به عنوان ورودی و خروجی می‌گیرد. زمانی که دو مقدار وارد مقایسه کننده می‌شود، مقایسه کننده مقدار کوچکتر را در سیم بالاتر، و مقدار بزرگتر را در سیم پایین قرار می‌دهد. به شبکه‌ای از سیم‌ها و مقایسه‌کننده‌ها که به طور صحیح تمام مقادیر ورودی را به صورت صعودی مرتب کنند یک شبکه مرتب‌سازی گفته می‌شود.

شکل زیر (چپ)، نشان‌دهنده یک شبکه مرتب‌سازی است. سیم‌ها در این شکل به صورت افقی و مقایسه کننده‌ها به صورت عمودی نشان داده شده‌اند. مراحل انجام مرتب‌سازی توسط این شبکه در شکل راست نشان داده شده است. فهم چگونگی صحیح عمل کردن این شبکه مرتب‌سازی آسان است. این را مدنظر داشته باشید که مقایسه کننده‌ها مقدار بزرگ را به سیم پایین و مقدار کوچک را به سیم بالا منتقل می‌کنند.

هدف این تمرین این است که عملکرد یک شبکه را روی یک دنباله‌ی ورودی شبیه‌سازی کنیم. در ورودی، شبکه و دنباله‌ی اعداد داده می‌شوند. خروجی برنامه تعیین می‌کند که آیا شبکه‌ی داده شده اعداد را به درستی مرتب می‌کند یا نه. توضیح این که لزوماً هر شبکه، به طور صحیح اعداد را مرتب نمی‌کند. مثال‌هایی از حالت‌های درست و نادرست در نمونه‌های زیر ذکر می‌شوند. علاوه بر این، ممکن است در توصیف شبکه‌ی ورودی خطاهایی وجود داشته باشد که برنامه‌ی شما باید آن‌ها را تشخیص دهد.

برای توصیف یک شبکه‌ی ورودی، از ماتریسی از کاراکترها استفاده می‌کنیم. به عنوان مثال شبکه‌ی نشان داده شده در شکل فوق توسط ماتریس زیر توصیف می‌شود:

a-c-

-bce

a-de

-bd-

هر سطر از ماتریس یک سیم را مشخص می‌کند. هر کاراکتر از یک سطر یا '–' است که نشان دهنده‌ی عدم وجود مقایسه کننده در آن بخش است یا با یک حرف لاتین مشخص میشود که در این صورت، باید در یکی (و تنها یکی) دیگر از کاراکترهای آن ستون کاراکتر مشابهی پیدا شود. به این ترتیب، فرض می شود یک واحد مقایسه‌کننده بین سیم های متناظر آن دو سطر وجود دارد. به عنوان مثال، در توصیف فوق یک مقایسه کننده بین دو سیم اول و سوم وجود دارد که با حرف a مشخص شده. ستون دوم نیز وجود یک مقایسه کننده بین سیم‌های دوم و چهارم است. ترتیب انتقال اعداد از چپ به راست فرض می شود.

در توصیف داده شده از شبکه ممکن است خطاهایی به شرح زیر وجود داشته باشد که برنامه‌ی شما باید آنها را تشخیص دهد:

- در یک ستون تنها یک مورد از یک حرف پیدا می‌شود یا این که بیش از دو مورد از آن حرف وجود دارد.
- در توصیف شبکه کاراکتری غیر از حروف کوچک لاتین و '–' وجود دارد.

دقت کنید که دو مقایسه کننده در دو ستون مختلف می توانند با یک حرف یکسان نمایش داده شوند، چون این امر ابهامی در عملکرد شبکه ایجاد نمی‌کند.

ورودی

خط اول هر مورد آزمون حاوی دو عدد صحیح N و K است که به ترتیب تعداد سطرها و تعداد ستون‌های ماتریس شبکه را مشخص می‌کنند. کنند. بعد از آن N خط پشت سر هم می‌آیند که هریک از K کاراکتر تشکیل شده‌اند. سپس در یک خط N عدد صحیح که به ترتیب روی سیم‌های 1 تا N قرار خواهند گرفت ذکر می‌شوند. در ورودی مسئله ممکن است تعداد بیش از یک مورد آزمون ذکر شود که هریک از قالب فوق پیروی می‌کند. آخرین خط ورودی شامل دو عدد صفر است.

خروجی

برای هر مورد آزمون، یک خط در خروجی بنویسید که با قالبی مانند آنچه در بخش «نمونه خروجی» آمده، یکی از سه نتیجه‌ی ممکن را مشخص کند: Not Sorted به این معنی که اعداد ورودی پس از پردازش توسط شبکه

مرتب نمی‌شوند، Sorted به این معنی که اعداد ورودی پساز پردازش توسط شبکه مرتب می‌شوند و Invalid Network به این معنی که ورودی داده شده قوانین مطرح شده در توصیف مساله را نقض می‌کند.

نمونه ورودی	نمونه خروجی
4 4 ac—e a—de b—df bc—f 1 3 2 0	Sorted
3 2 a— ab —b 1 3 0	Not␣sorted
3 2 a— ab ab 1 3 0 0 0	Invalid␣network

دقت کنید

- برنامه‌ی خود را فقط در قالب یک فایل به زبان سی پلاس پلاس تحویل دهید.
- به جز کتابخانه‌های استاندارد زبان سی پلاس پلاس و std_lib_facilities.h از هدرفایل دیگری استفاده نکنید.
- قبل از تحویل، برنامه‌ی خود را با انواع ورودی‌ها بیازمایید. حالت‌های مرزی ورودی‌های مسئله را در نظر بگیرید.

- به قالب خروجی دقت کنید. بزرگی و کوچکی حروف، فاصله های خالی و شکست خط‌ها مهم هستند. به هیچ وجه چیزی به جز آنچه در بخش خروجی اشاره شده در خروجی برنامه ننویسید (مثل پیغام‌هایی به کاربر برنامه)

مطالعه‌ی بیشتر

آیا می دانید قضیه‌ای وجود دارد که اگر شبکه‌ای هر دنباله‌ی ورودی از صفر و یک را به درستی مرتب کند، تمام دنباله‌های های دیگر از اعداد دلخواه را نیز به درستی مرتب می‌کند؟ شبکه‌های مرتب‌سازی در عین سادگی دارای مباحث نظری عمیق و جالبی هستند. برای مطالعه‌ی بیشتر می‌توانید به فصل ۲۷ کتاب زیر مراجعه کنید.

T. Cormen, C. Leiserson, R. Rivest, and C. Stein, “Introduction to Algorithms”, 2nd edition, MIT Press, 2001.

همچنین صفحه‌ی Sorting Networks از ویکی پیدیا نیز دارای پیوندهای مفیدی در این زمینه است. بخشی از توضیحات این بخش از ویکی پیدیای فارسی برداشته شده است.

برای درک عملکرد این شبکه، جدول زیر و اعداد ورودی به آن را در سمت چپ در نظر بگیرید.

	مرحله ۱	مرحله ۲	مرحله ۳	مرحله ۴	
۱	a	c	-	e	سیم ۱
۳	a	-	d	e	سیم ۲
۲	b	-	d	f	سیم ۳
۰	b	c	-	f	سیم ۴

- در مرحله‌ی اول دو کامپاراتور (مقایسه کننده) بین سیم ۱ و ۲ و سیم ۳ و ۴ داریم.
- در مرحله‌ی دوم یک کامپاراتور بین سیم ۱ و ۴ داریم.
- در مرحله‌ی سوم یک کامپاراتور بین سیم ۲ و ۳ داریم.
- در مرحله‌ی چهارم دو کامپاراتور بین سیم ۱ و ۲ و سیم ۳ و ۴ داریم.

مرحله‌ی اول: مقایسه‌ی سیم ۱ و ۲ باعث جابجایی نمی‌شود. مقایسه‌ی سیم ۳ و ۴ باعث می‌شود صفر در بالا و ۲ در پایین قرار بگیرد.

	مرحله ۴	مرحله ۳	مرحله ۲	نتیجه مرحله ۱	مرحله ۱	
سیم ۱	e	-	c	۱	a	۱
سیم ۲	e	d	-	۳	a	۳
سیم ۳	f	d	-	۰	b	۲
سیم ۴	f	-	c	۲	b	۰

مرحله‌ی دوم: مقایسه‌ی سیم ۱ و ۴ تاثیری ندارد.

	مرحله ۴	مرحله ۳	نتیجه مرحله ۲	مرحله ۲	نتیجه مرحله ۱	مرحله ۱	
سیم ۱	e	-	۱	c	۱	a	۱
سیم ۲	e	d	۳	-	۳	a	۳
سیم ۳	f	d	۰	-	۰	b	۲
سیم ۴	f	-	۲	c	۲	b	۰

مرحله‌ی سوم: مقایسه‌ی سیم ۲ و ۳ باعث می‌شود صفر در بالا روی سیم دوم و ۳ در پایین روی سیم سوم قرار بگیرد.

	مرحله ۴	نتیجه مرحله ۳	مرحله ۳	نتیجه مرحله ۲	مرحله ۲	نتیجه مرحله ۱	مرحله ۱	
سیم ۱	e	۱	-	۱	c	۱	a	۱
سیم ۲	e	۰	d	۳	-	۳	a	۳
سیم ۳	f	۳	d	۰	-	۰	b	۲
سیم ۴	f	۲	-	۲	c	۲	b	۰

مرحله‌ی چهارم: مقایسه‌ی سیم ۱ و ۲ باعث می‌شود صفر در بالا روی سیم اول و ۱ در پایین روی سیم دوم قرار بگیرد. مقایسه‌ی سیم ۳ و ۴ باعث می‌شود ۲ در بالا روی سیم سوم و ۳ در پایین روی سیم چهارم قرار بگیرد.

	نتیجه مرحله ۴	مرحله ۴	نتیجه مرحله ۳	مرحله ۳	نتیجه مرحله ۲	مرحله ۲	نتیجه مرحله ۱	مرحله ۱	
سیم ۱	۰	e	۱	-	۱	c	۱	a	۱

سیم ۲	۱	e	۰	d	۳	-	۳	a	۳
سیم ۳	۲	f	۳	d	۰	-	۰	b	۲
سیم ۴	۳	f	۲	-	۲	c	۲	b	۰

مارتین فولر، یکی از دانشمندان برجسته‌ی علوم کامپیوتر، bad smell (CodeSmell) ها را به ترتیب زیر اولویت‌بندی می‌کند.

Duplicated Code	Long Method
Large Class	Long Parameter List
Divergent Change	Shotgun Surgery
Feature Envy	Data Clumps
Primitive Obsession	Switch Statements
Parallel Inheritance Hierarchies	Lazy Class
Speculative Generality	Temporary Field
Message Chains	Middle Man
Inappropriate Intimacy	Alternative Classes with Different Interfaces
Incomplete Library Class	Data Class
Refused Bequest	Comments

فصل سوم کتاب Refactoring: Improving the Design of Existing Code

Comment is deodorant

refactoring smell code

smell code cheat sheet

code smell martin fowler

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

vector<int> read_numbers(int count) {
	vector<int> result;
	for (int i = 0; i < count; i++) {
		int number;
		cin >> number;
		result.push_back(number);
	}
	return result;
}

vector<string> read_network(int num_of_inputs) {
	vector<string> result;
	for (int i = 0; i < num_of_inputs; ++i) {
		string line;
		cin >> line;
		result.push_back(line);
	}
	return result;
}

void swap(int& a, int& b) {
	int temp = a;
	a = b;
	b = temp;
}

bool is_valid_network(vector<string> network, int num_of_stages) {
	for (int j = 0; j < num_of_stages; ++j) {
		for (int i = 0; i < network.size(); ++i) {
			if (network[i][j] == '-')
				continue;

			int count = 0;
			for (int k = 0; k < network.size(); k++)
				if (network[i][j] == network[k][j])
					count++;
```

```
                if (count != 2) {
                    return false;
                }
            }
        }
        return true;
}

void apply(vector<string> network, int j, vector<int>& numbers) {
        for (int i = 0; i < network.size() - 1; i++) {
            if (network[i][j] == '-')
                continue;

            for (int k = i + 1; k < network.size(); k++)
                if (network[i][j] == network[k][j])
                    if (numbers[i] > numbers[k])
                        swap(numbers[i], numbers[k]);
        }
}

bool sorted(vector<int> numbers) {
        for (int i = 0; i < numbers.size() - 1; i++)
            if (numbers[i] > numbers[i + 1])
                return false;
        return true;
}

void process_testcase(int num_of_inputs, int num_of_stages) {
        vector<string> network = read_network(num_of_inputs);
        vector<int> numbers = read_numbers(num_of_inputs);

        if (!is_valid_network(network, num_of_stages)) {
            cout << "Invalid network\n";
            return;
        }

        for (int j = 0; j < num_of_stages; j++)
            apply(network, j, numbers);

        if (sorted(numbers))
            cout << "Sorted";
        else
            cout << "Not sorted";
        cout << endl;
}
```

```
int main() {
	int num_of_inputs;
	int num_of_stages;

	cin >> num_of_inputs >> num_of_stages;
	while (num_of_inputs != 0 && num_of_stages != 0) {
		process_testcase(num_of_inputs, num_of_stages);
		cin >> num_of_inputs >> num_of_stages;
	}
	return 0;
}
```

خط ۶: تایپ بازگشتی یک تابع می‌تواند از نوع بردار باشد.

```
vector<int> read_numbers(int count) {
```

در خط ۵۵، تابع مذکور ستون j ام از شبکه را روی بردار numbers اعمال می‌کند. دقت کنید که بردار مذکور با ارجاع تعریف شده است.

```
void apply(vector<string> network, int j, vector<int>& numbers) {
```

منظور از خط ۹۴، ورودی‌های شبکه‌ی مرتب ساز است که عملا تعداد سطرهای جدول مربوطه می‌باشد.

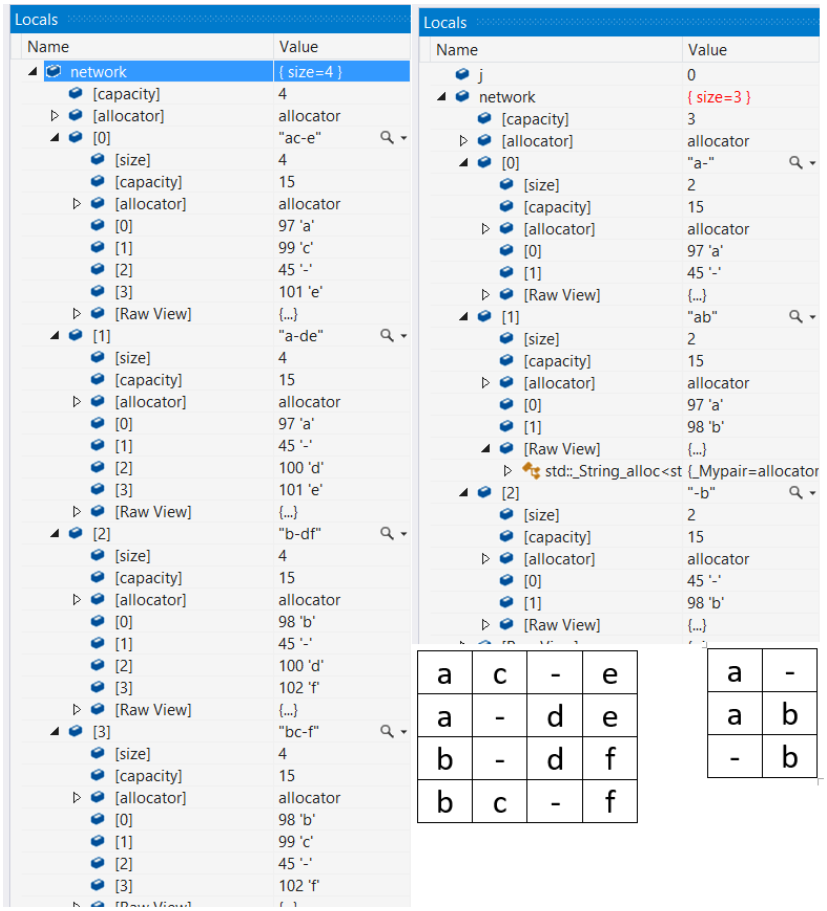
```
int num_of_inputs;
```

منظور از خط ۹۵، تعداد طبقات شبکه‌ی مرتب ساز است که عملا تعداد ستون‌های جدول مربوطه می‌باشد.

```
int num_of_stages;
```

ساختار داده:

در اینجا شبکه‌ی خود را برداری از رشته‌ها (vector<string>) تعریف کردیم که هر stage، یک ستون از شبکه است و شبکه‌ای که تابع read_network تشکیل می‌داد و به تابع apply پاس می‌شد به صورت زیر بود:



می‌توانستیم شبکه را طوری تعریف کنیم که هر stage یک سطر از شبکه باشد. در آن صورت به تابع appy به جای کل شبکه فقط یک عضو شبکه یا یک سطر (یک stage) را پاس می‌کردیم.

اما در آن صورت برای خواندن ورودی بایستی یک حلقه می‌نوشتیم و transpose می‌کردیم.

گاه ممکن است سخت کردن فرآیند اخذ و ذخیره سازی داده و در عوض ساده‌تر کردن پردازش به صرفه باشد. در این حالت پیاده‌سازی به صورت زیر خواهد بود:

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

using namespace std;

typedef vector<char> Stage;
typedef vector<Stage> Network;
typedef vector<int> Numbers;
```

```
Numbers read_numbers(int num_of_inputs) {
	Numbers result;
	for (int i = 0; i < num_of_inputs; i++) {
		int number;
		cin >> number;
		result.push_back(number);
	}
	return result;
}

Network read_network(int num_of_inputs, int num_of_stages) {
	Network net(num_of_stages);
	for (int i = 0; i < num_of_inputs; i++) {
		string row;
		cin >> row;
		for (int j = 0; j < row.length(); j++) {
			net[j].push_back(row[j]);
		}
	}
	return net;
}

int stages(Network net) {
	return net.size();
}

int inputs(Network net) {
	return net[0].size();
}

Stage ith_stage(Network net, int i) {
	return net[i];
}

void swap(int& a, int& b) {
	int temp = a;
	a = b;
	b = temp;
}

bool is_valid_network(Network network) {
	for (int j = 0; j < stages(network); ++j) {
		for (int i = 0; i < inputs(network); ++i) {
			if (network[j][i] == '-')
				continue;
```

```
                int count = 0;
                for (int k = 0; k < inputs(network); k++)
                    if (network[j][i] == network[j][k])
                        count++;

                if (count != 2) {
                    return false;
                }
            }
        }
        return true;
}

void pass_through(Numbers &nums, Stage stage) {
        for (int r = 0; r < stage.size(); r++) {
            if (stage[r] == '-')
                continue;

            for (int i = r + 1; i < stage.size(); i++)
                if (stage[r] == stage[i])
                    if (nums[r] > nums[i])
                        swap(nums[r], nums[i]);
        }
}

bool sorted(Numbers numbers) {
        for (int i = 0; i < numbers.size() - 1; i++)
            if (numbers[i] > numbers[i + 1])
                return false;
        return true;
}

void process_testcase(int num_of_inputs, int num_of_stages) {
        Network net = read_network(num_of_inputs, num_of_stages);
        Numbers numbers = read_numbers(num_of_inputs);

        if (!is_valid_network(net)) {
            cout << "Invalid network\n";
            return;
        }

        for (int j = 0; j < num_of_stages; j++)
            pass_through(numbers, ith_stage(net, j));

        if (sorted(numbers))
            cout << "Sorted";
```

```
	else
		cout << "Not sorted";
	cout << endl;
}

int main() {
	int num_of_inputs;
	int num_of_stages;

	cin >> num_of_inputs >> num_of_stages;
	process_testcase(num_of_inputs, num_of_stages);
	return 0;
}
```

در رابطه با قطعه کد زیر:

```
typedef vector<char> Stage;
typedef vector<Stage> Network;
typedef vector<int> Numbers;
```

کلمه‌ی کلیدی typedef مکانیسمی برای ایجاد نام دوم برای انواع داده‌ای از پیش تعریف شده فراهم می‌کند. نام انواع داده‌ای struct اغلب اوقات با typedef تعریف می‌شوند که نام نوع داده‌ی کوتاه‌تر، ساده‌تر و یا خواناتر را ایجاد می‌کند. برای مثال دستور:

```
typedef Card *CardPtr;
```

نام نوع جدید CardPtr را به عنوان نام دوم * Card تعریف می‌کند.

روش خوب برنامه‌نویسی: نام‌های typedef را با حروف بزرگ بنویسید تا تاکیدی بر این مطلب باشد که این نام‌ها، نام دوم نام‌های نوع دیگر هستند.

ایجاد نام جدید با typedef، نوع جدید ایجاد نمی‌کند. typedef فقط یک نوع نام جدید ایجاد می‌کند که می‌تواند در برنامه به عنوان نام دوم برای یک نام نوع موجود به کار گرفته شود.

نکته‌ای درباره‌ی قابلیت حمل برنامه‌ها: نام‌های دوم برای انواع داده‌ای کتابخانه‌ای را می‌توان با typedef ایجاد کرد تا برنامه‌ها را قابل حمل‌تر سازد. برای مثال برنامه می‌تواند با استفاده از typedef نام دوم Integer را برای اعداد صحیح چهار بایتی ایجاد کند. آن‌گاه از Integer می‌توان به عنوان نام دوم در سیستم‌هایی با عدد صحیح چهاربایتی به جای int استفاده کرد و نیز از آن می‌توان به عنوان نام دوم در سیستم‌های با چهار عدد صحیح

چهار بایتی به جای long int استفاده کرد که در آن مقادیر long int چهار بایت از حافظه را اشغال می‌کند. سپس، برنامه نویس به سادگی تمام متغیرهای صحیح چهار بایتی را به صورت نوع Integer اعلان می‌کند.

ممکن است در رابطه با تابع ith_stage سوال پیش بیاید که چرا در فراخوانی تابع pass_through همان عضو iام network را استفاده نکرده‌ایم و برای آن یک تابع جدا نوشته‌ایم؟

```
for (int j = 0; j < num_of_stages; j++)
      pass_through(numbers, ith_stage(net, j));

Stage ith_stage(Network net, int i) {
      return net[i];
}
```

علت همان افزایش خوانایی و encapsulation است. یک برنامه نویس دیگر وقتی کد ما را می‌خواند، خبر ندارد که network برداری از stageهاست. با این تابع سعی کرده ایم با network به مثابه شیئی برخورد کنیم که خبری از محتویات درون آن نداریم و می دانیم روی این شی می‌توان یک تابع فراخوانی کرد و مرحله‌ی iام آن را بدست آورد.

روش دیگر و باصرفه‌تر برای ساختار داده این است که stage خود را به جای برداری از کاراکترها، به صورت برداری از کامپراتورها یا برداری از زوج‌های مرتب ذخیره کنیم. به عنوان مثال شبکه‌ی زیر را در نظر بگیرید:

a	-	-	b
a	-	-	-
-	-	-	-
-	a	-	-
-	-	-	-
-	-	a	-
b	-	b	-
c	-	-	a
-	-	-	a
-	-	-	-
-	a	-	-
c	-	-	-
b	-	a	-
-	-	b	b

a	a	-	-	-	-	b	c	-	-	-	c	b	-

-	-	-	a	-	-	-	-	-	-	a	-	-	-
-	-	-	-	-	a	b	-	-	-	-	-	a	b
b	-	-	-	-	-	-	a	a	-	-	-	-	b

اکثر خانه‌های این شبکه، حاوی – است که عملا کاری انجام نمی‌دهند و به خاطر این – ها، قطعه کد:

```
if (network[i][j] == '-')
	continue;
```

در تابع apply، یا قطعه کد

```
if (stage[r] == '-')
	continue;
```

در تابع pass_through:

- برای stage 1، ۸ بار اجرا می‌شود.
- برای stage 2، ۱۲ بار اجرا می‌شود.
- برای stage 3، ۱۰ بار اجرا می‌شود.
- برای stage 4، ۱۰ بار اجرا می‌شود.

از سوی دیگر کد:

```
for (int i = 0; i < network.size() - 1; i++) {
	if (network[i][j] == '-')
		continue;

	for (int k = i + 1; k < network.size(); k++)
		if (network[i][j] == network[k][j])
			if (numbers[i] > numbers[k])
				swap(numbers[i], numbers[k]);
}
```

در تابع apply، سطر اول شبکه زیر را در هر stage، ۱ بار، سطر دوم را ۲ بار، ... سطر سیزدهم را ۱۳ بار و سطر چهاردهم را ۱۴ بار ملاقات می‌کند.

a	-	-	b
a	-	-	-
-	-	-	-
-	a	-	-
-	-	-	-
-	-	a	-

b	-	b	-
c	-	-	a
-	-	-	a
-	-	-	-
-	a	-	-
c	-	-	-
b	-	a	-
-	-	b	b

پس تعداد محاسبات انجام شده برابر است با:

$$4\times(14+13+...+2+1)=4\times\left(\frac{14\times13}{2}\right)=364$$

و یا کد:

```
for (int r = 0; r < stage.size(); r++) {
	if (stage[r] == '-')
		continue;

	for (int i = r + 1; i < stage.size(); i++)
		if (stage[r] == stage[i])
			if (nums[r] > nums[i])
				swap(nums[r], nums[i]);
}
```

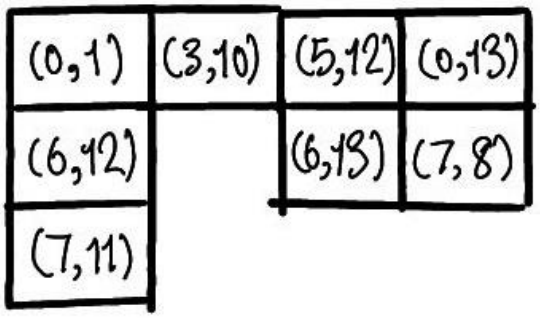
در تابع pass_through برای هر stage شبکه‌ی زیر، ۹۱ بار انجام می‌شود.

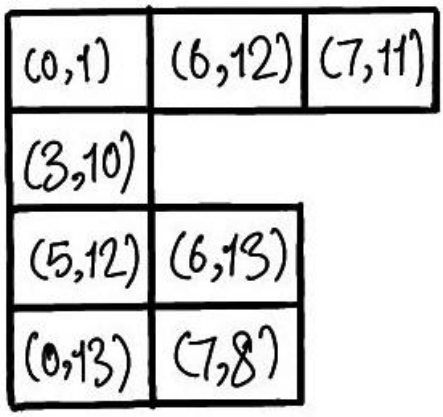
a	a	-	-	-	-	b	c	-	-	-	c	b	-
-	-	-	a	-	-	-	-	-	-	a	-	-	-
-	-	-	-	-	a	b	-	-	-	-	-	a	b
b	-	-	-	-	-	-	a	a	-	-	-	-	b

چون عضو اول stage را ۱ بار، عضو دوم را ۲ بار، ...، عضور سیزدهم را ۱۳ بار و عضور چهاردهم را ۱۴ بار ملاقات می‌کند.

$$14+13+...+2+1=\frac{14\times13}{2}=91$$

چنانچه شبکه را هنگام خواندن آن به یکی از صورت‌های زیر ذخیره کنیم:





برای stage1 فقط ۳ خانه، برای stage2 فقط یک خانه، برای stage3 و stage4 نیز فقط دو خانه و هرکدام از خانه ها نیز فقط یک‌بار دیدار خواهد شد. لذا در مجموع ۸ دیدار داریم نه ۹۱ یا ۳۶۴ دیدار و پردازش ساده‌تر و سریع‌تر خواهد بود. از سوی دیگر در میزان حافظه‌ی مورد نیاز جهت ذخیره سازی شبکه نیز صرفه جویی خواهیم کرد. البته خواندن ورودی‌ها اندکی دشوارتر خواهد بود.

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

using namespace std;

struct Comparator {
      int end1;
      int end2;
};

typedef vector<Comparator> Stage;
typedef vector<Stage> Network;
typedef vector<int> Numbers;

Comparator new_comparator(int, int);
Numbers read_numbers(int);
Network read_network(int, int);
Stage ith_stage(Network, int);
void swap(int&, int&);
void apply(Numbers&, Stage);
bool sorted(Numbers);
void process_testcase(int, int);
```

```
int main() {
	int num_of_inputs;
	int num_of_stages;

	cin >> num_of_inputs >> num_of_stages;
	process_testcase(num_of_inputs, num_of_stages);
	system("pause");
	return 0;
}

Comparator new_comparator(int e1, int e2) {
	Comparator c;
	c.end1 = e1;
	c.end2 = e2;
	return c;
}

Numbers read_numbers(int num_of_inputs) {
	Numbers result;
	for (int i = 0; i < num_of_inputs; i++) {
		int number;
		cin >> number;
		result.push_back(number);
	}
	return result;
}

Network read_network(int num_of_inputs, int num_of_stages) {
	Network net(num_of_stages);
	vector<string> temp_net;
	for (int i = 0; i < num_of_inputs; i++) {
		string line;
		cin >> line;
		temp_net.push_back(line);
	}

	for (int i = 0; i < num_of_stages; i++) {
		for (int j = 0; j < num_of_inputs; j++) {
			if (temp_net[j][i] == '-')
				continue;
			for (int k = j + 1; k < num_of_inputs; k++)
				if (temp_net[j][i] == temp_net[k][i])
					net[i].push_back(new_comparator(j, k));
		}
	}
	return net;
```

```
}

Stage ith_stage(Network net, int i) {
	return net[i];
}

void swap(int& a, int& b) {
	int temp = a;
	a = b;
	b = temp;
}

void apply(Numbers &nums, Stage stage) {
	for (int c = 0; c < stage.size(); c++) {
		if (nums[stage[c].end1] > nums[stage[c].end2])
			swap(nums[stage[c].end1], nums[stage[c].end2]);
	}
}

bool sorted(Numbers numbers) {
	for (int i = 0; i < numbers.size() - 1; i++)
		if (numbers[i] > numbers[i + 1])
			return false;
	return true;
}

void process_testcase(int num_of_inputs, int num_of_stages) {
	Network net = read_network(num_of_inputs, num_of_stages);
	Numbers numbers = read_numbers(num_of_inputs);

	for (int j = 0; j < num_of_stages; j++)
		apply(numbers, ith_stage(net, j));

	if (sorted(numbers))
		cout << "Sorted";
	else
		cout << "Not sorted";
	cout << endl;
}
```

در رابطه با قطعه کد زیر:

```
struct Comparator {
      int end1;
      int end2;
};
```

Structها (یا رکوردها) نوع داده‌ی مجتمع هستند یعنی آن‌ها را می‌توان با استفاده از عضوهایی از چند نوع مختلف که می‌توانند شامل structهای دیگری باشند تعریف کرد. تعریف struct (رکورد) زیر را در نظر بگیرید:

```
struct Card {
      char *face;
      char *suit;
}; // end struct Card
```

کلمه‌ی کلیدی struct، تعریف رکورد Card را معرفی می‌کند. شناسه‌ی Card نام struct (نام رکورد) است و در C++ برای اعلان متغیرهایی از نوع struct به‌کار می‌رود. در این مثال نوع struct، Card است. داده‌هایی (و احتمالا تابع‌ها به همان صورت که با کلاس‌ها است) که در داخل آکولادهای باز و بسته‌ی تعریف رکورد struct اعلان شده باشند، عضوهای struct نام دارند. نام عضوهای یک رکورد باید منحصر به فرد باشند اما دو struct مختلف می‌توانند عضوهای همنام داشته باشند بدون آن‌که در کار هم تداخل ایجاد کنند. هر تعریف struct باید با سمی کالن (;) به پایان رشد.

خطای رایج در برنامه نویسی: ننوشتن سمی کالن در پایان تعریف struct خطای دستوری است.

تعریف Card شامل دو عضو از نوع *char یعنی face و suit است. عضوهای struct می‌توانند متغیرهایی از انواع داده ای و اصلی (مانند int و double و غیره) یا مجتمع مانند آرایه‌ها، structهای دیگر و یا کلاس‌ها باشند. داده‌ی عضو در تعریف یک struct می‌تواند از چند نوع داده باشد. برای مثال رکورد employee می‌تواند حاوی عضوهای رشته‌ای کاراکتری برای نام خانوادگی، یک عضو int برای سن کارمند، یک عضو char شامل ‘M’ یا ‘F’ برای جنس کارمند، یک عضو double برای حقوق ساعتی کارمند و غیره باشد.

Struct نمی‌تواند حاوی یک نمونه از خودش باشد. برای مثال متغیر رکوردی Card را نمی‌توان در تعریف struct Card اعلان کرد. اما اشاره‌گر به رکورد Card را می‌توان در داخل تعریف Struct Card گنجاند. اگر رکوردی شامل عضوی باشد که آن عضو یک اشاره‌گر به همان نوع رکورد باشد، به آن **رکورد خودارجاعی** می‌گویند. در فصل ۲۱ با عنوان ساختمان داده‌ها، از ساختار مشابهی به نام کلاس‌های خودارجاعی استفاده کردیم که به کمک آن انواع متعددی از ساختارهای لیست پیوندی را ایجاد کردیم.

تعریف رکورد Card هیچ حافظه‌ای لز سیستم نمی‌گیرد بلکه نوع داده‌ی جدیدی ایجاد می‌کند که از آن برای اعلان متغیرهای رکوردی استفاده می‌شود. متغیرهای رکوردی مانند متغیرهای نوع دیگر اعلان می‌شوند. اعلان‌های زیر:

```
Card oneCard;
Card deck[52];
Card *CardPtr;
```

OneCard را یک متغیر رکوردی از نوع Card، deck را آرایه‌ای با ۵۲ عضو از نوع Card و CardPtr را به عنوان یک اشاره‌گر به رکورد Card اعلان می‌کند. متغیرهای یک نوع داده‌ی رکوردی معین را می‌توان با قرار دادن لیست نام متغیرها که با کاما از هم جدا شده‌اند بین آکولادهای باز و بسته تعریف رکورد و سمی کالن پایان تعریف رکورد اعلان کرد. برای مثال اعلان‌های بالا را می‌توان در تعریف رکورد زیر قرار داد:

```
struct Card {
      char *face;
      char *suit;
} oneCard, deck[52], *CardPtr;
```

نام رکورد اختیاری است. اگر تعریف رکورد حاوی نام رکورد نباشد متغیرهای نوع رکورد را می‌توان تنها بین آکلاد بسته‌ی تعریف رکورد و سمی کالن پایان تعریف رکورد اعلان کرد.

ملاحظاتی درباره‌ی مهندسی نرم‌افزار: هنگامی که نوع struct را ایجاد می‌کنید نام struct را مشخص کنید. به نام struct برای اعلان متغیرهای جدید از نوع struct اعلان پارامترهایی از نوع struct اندکی بعد در برنامه احتیاج است، در صورت استفاده از struct مانند کلاس C++ به نام struct برای مشخص کردن سازنده و نابودکننده احتیاج است.

تنها عملیات مجاز کتابخانه‌ای که می‌توان روی اشیا struct انجام داد عبارتند از جایگزینی یک شی struct در یک شی struct از همان نوع، گرفتن آدرس (&) یک شی struct، دسترسی به عضوهای یک شی struct به همان صورتی که عضوهای یک کلاس در دسترس قرار می‌گیرد و استفاده از عملگر sizeof برای طول اندازه‌ی یک struct.

اکثر عملگرها را مانند کار با کلاس‌ها می‌توان سربارگذاری کرد تا با اشیایی از نوع struct کار کنند.

خطای رایج در برنامه نویسی: مقایسه‌ی رکوردها (structها) خطای زمان کامپایل است.

Structها (رکوردها) را می‌توان با استفاده از لیست‌های مقدار اولیه، به همان صورتی که با آرایه‌ها عمل کردیم، مقدار اولیه دهیم. برای مثال، اعلان

```
Card oneCard = { "Three", "Hearts" };
```

متغیر OneCard را از نوع Card اعلان می‌کند و به عضو face مقدار اولیه‌ی "Three" و به عضو suit مقدار اولیه‌ی "Hearts" می‌دهد. اگر تعداد مقادیر اولیه‌ی داخل لیست کمتر از تعداد عضوهای داخل رکورد باشد به عضوهای باقیمانده با مقادیر پیش‌فرض‌شان مقدار اولیه داده می‌شود.

جمع بندی: به یک برنامه از دو جنبه می‌توان نگریست.

۱- کارهایی که برنامه قرار است انجام دهد (وظایف برنامه را باید به بلاک‌های کوچک بشکنیم)
۲- نگهداری داده‌های برنامه (پردازش را چگونه بشکنیم و داده‌ها را چگونه نگهداریم که تصمیم در مورد آن اثر مستقیم بر زمان و اجرای برنامه دارد)

درس ساختمان داده و الگوریتم: آرایش داده چگونه باشد تا پردازش سریع‌تر شود.

درخت دودویی: تعداد محاسبات متناسب با لگاریتم تعداد عناصر

در نوشتن برنامه‌ها اصرار داریم که از global variable استفاده نکنیم.

Global variable نگهداری برنامه را دشوار می‌کند.

اما به عنوان مثال اگر در برنامه‌ی مربوز به این جلسه شبکه را به صورت global تعریف می‌کردیم نیازی نبود که در is_valid_network شبکه را پاس کنیم و یا در apply لازم نبود که شبکه را پاس کنیم و یا هنگام خواندن شبکه در تابع read_network نیازی به return کردن نبود.

اما هدف از تقسم برنامه این است که به هر بلوک به صورت مستقل نگاه کنیم. تعریف global variable باعث می‌شود که اثر توابع روی هم فقط از طریق پارامترها نباشد و لذا خوانایی برنامه کاهش می‌یابد.

برای دریافت اینکه global variable کجا تغییر می‌یابد، بایستی کل برنامه را بخوانیم که این یعنی کاهش خوانایی برنامه.

Global variable باعث می‌شود نتوانیم تابع را در جایی دیگر استفاده کنیم و قابلیت نگهداری، خوانایی و استفاده مجدد را کاهش می‌دهد.

اصولا استفاده از global variable توصیه نمی‌شود مگر اینکه بدانید طول عمر کد کم است. مثلا اگر در مسابقه‌ی ACM شرکت می‌کنید نیازی به لحاظ کردن این موارد نیست. چون این کد قرار نیست maintain شود و شما در کمترین زمان جواب می‌خواهید.

جلسه چهارم – مبانی توابع بازگشتی

رد کردن پارامتر با مقدار: این برنامه‌ی خیلی ساده رد کردن پارامترها با مقدار (call by value) را نشان می‌دهد.

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

void f(int x)
{
	int i = 5;
	x = i + 1;
}

int main()
{
	int a = 10;
	f(a);
	cout << a << endl;
	return 0;
}
```

در خط ۱۲ مقدار a یعنی ۱۰ به تابع f فرستاده می‌شود. لذا پس از فراخوانی تابع f در خط ۱۳، کماکان مقدار متغیر a برابر ۱۰ است. چون متغیر به تابع فرستاده نشده و لذا تابع تغییری در متغیر ایجاد نکرده. بلکه مقدار متغیر، هنگام فراخوانی در متغیر x فرار گرفته و عملیات روی متغیر x درون تابع انجام شده است.

رد کردن پارامتر با ارجاع: این برنامه‌ی خیلی ساده رد کردن پارامترها با ارجاع (call by reference) را نشان می‌دهد. این نوع رد کردن با ارجاع، خاص زبان سی‌پلاس‌پلاس است و در زبان سی این کار با اشاره‌گرها صورت می‌گیرد.

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

void f(int x, int& y)
{
	int i = 5;
	x = i + 1;
	y = 18;
}

int main()
{
	int a = 10;
	int b = 11;
	f(a, b);
	cout << a << endl;
	cout << b << endl;
	f(a + 1, b + 1);

	return 0;
}
```

در خط ۴، پارامتر y با ارجاع رد می‌شود و تغییری که در خط ۸ ایجاد می‌شود، به بیرون منتقل می‌گردد یعنی b در خط ۱۷ برابر ۱۸ نمایش داده می‌شود.

در خط ۱۸ با compilation error مواجه می‌شویم:

error C2664: 'void f(int,int &)': cannot convert argument 2 from 'int' to 'int &'

چون b+1 مقدار است که در همان لحظه پاس به تابع تولید می‌شود و متغیر نیست و آدرس ندارد که آدرس به تابع داده شود.

فراخوانی تودرتوی توابع: این مثال ساده رد کردن پارامتر هنگام فراخوانی‌های متوالی توابع را بررسی می‌کند.

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

void g(int& y)
{
	int j = 2;
	y = j * 3;
}

void f(int x)
{
	g(x);
	cout << x << endl;
}

int main()
{
	int a = 10;
	f(a);
	cout << a << endl;
	return 0;
}
```

در خط ۱۲ پیش از فراخوانی تابع g مقدار x برابر ۱۰ است اما در خط ۱۳ به مقدار ۶ تغییر می‌کند. در خط ۲۰ مقدار a پس از فراخوانی کماکان برابر ۱۰ است.

توابع بازگشتی: حل مساله به صورت بازگشتی یعنی تبدیل مساله به مساله‌ای با اندازه‌ی کوچکتر از همان جنس

هر فراخوانی از تابع یک فریم از حافظه می‌گیرد.

محاسبه‌ی فاکتوریل: این برنامه به روش بازگشتی فاکتوریل یک عدد را محاسبه می‌کند. ایده‌ی اصلی در حل این مسئله بر این رابطه استوار است که برای $n>1$ داریم $n!=n*(n-1)!$

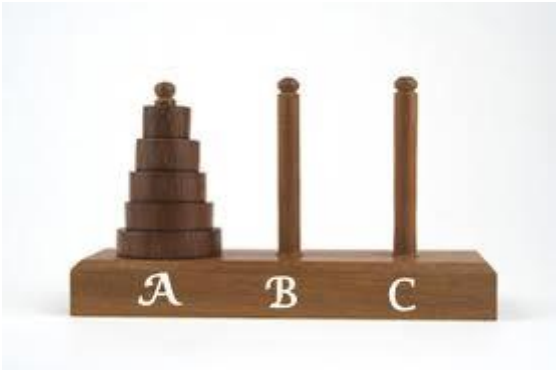
```
#include <iostream>
using namespace std;
int fact(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

```
int main()
{
    int num;
    cout << "Enter a positive integer number: ";
    cin >> num;
    int num_fact = fact(num);
    cout << num << "! = " <<  num_fact << endl;
}
```

سوالی که ممکن است پیش بیاید این است که با توجه به اینکه می‌دانیم، فاکتوریل تنها برای اعداد طبیعی تعریف می‌شود، چرا در خط ۵ به عوض استفاده از شرط n == 1 از شرط n <= 1 استفاده کرده‌ایم.

برنامه‌ها دارای دو ویژگی correctness و robustness هستند. correctness یعنی اینکه برنامه وظیفه‌ی موردنظر را به درستی انجام دهد اما robustness یعنی اینکه برنامه در قبال ورودی غیرمجاز رفتار معقول داشته باشد. شرط را بدین صورت نوشتیم تا اگر کاربر اعداد ۰ یا منفی را وارد کرد، سیستم دچار crash نشود.

برج هانوی (Tower of Hanoi): به شکل زیر توجه کنید:

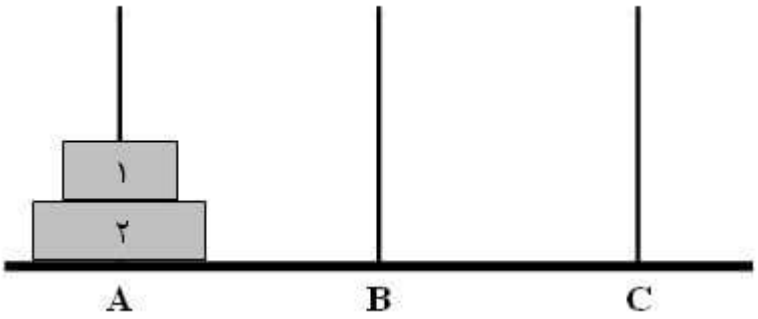


سه میله‌ی - میله‌ی مبدأ (A) ، میله‌ی کمکی (B) و میله‌ی مقصد (C) - و تعدادی دیسک در میله‌ی مبدأ داریم. هدف انتقال تمام دیسک‌ها از این میله به میله‌ی مقصد با رعایت دو شرط زیر است:

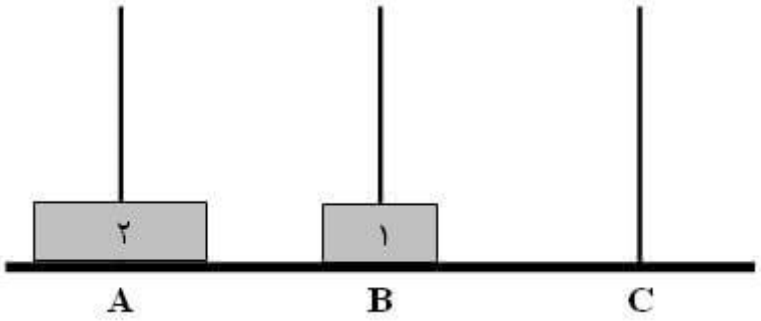
- در هر زمان فقط یک دیسک را می‌توان جابجا نمود.
- نباید در هیچ زمانی دیسکی بر روی دیسک با اندازه‌ی کوچکتر قرار بگیرد.

به طور حتم می‌توان با روش آزمون و خطا به نتیجه‌ی مطلوب رسید. اما هدف ما ارائه‌ی الگوریتمی برای انتقال دیسک‌ها با کمترین جابجایی ممکن است.

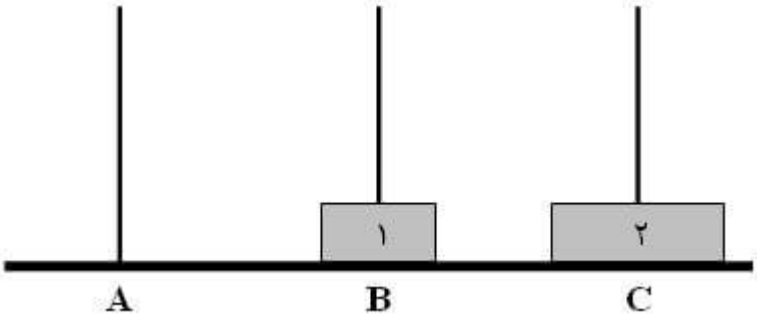
به عنوان مثال، اگر n = 2 باشد:



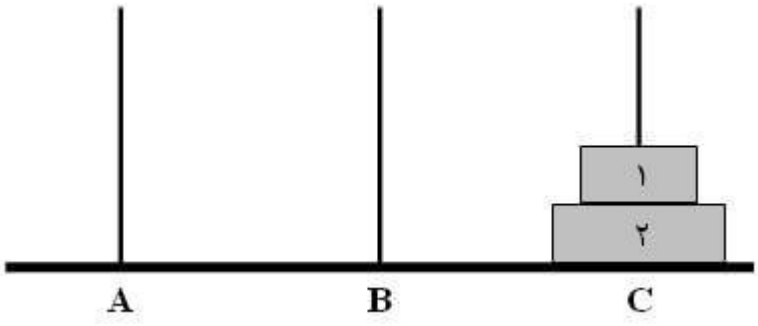
۱) دیسک ۱ را به میله‌ی B منتقل می‌کنیم ($A \to B$):



۲) دیسک ۲ را به میله‌ی C منتقل می‌کنیم ($A \to C$):



۳) دیسک ۱ را به میله‌ی C منتقل می‌کنیم ($B \to C$):



توجه داشته باشید که بر اساس قانون اول، نمی‌توان به غیر از بالاترین دیسک هر میله، به دیسک دیگری از آن دسترسی پیدا کرد.

حل بازگشتی مسئله‌ی برج هانوی: برای اینکه بتوان از روش بازگشتی تقسیم و حل (یا تقسیم و غلبه - Divide and Conquer) برای حل یک مسئله استفاده نمود، مسئله باید قابلیت خرد شدن به زیرمسئله‌هایی از همان نوع مسئله‌ی اصلی و اندازه‌ی کوچکتر را داشته باشد. این ویژگی در مورد مسئله‌ی برج هانوی صدق می‌کند.

ایده‌ی اصلی از آنجا ناشی می‌شود که برای جابجا کردن بزرگترین دیسک از میله‌ی A به میله‌ی C، ابتدا باید تمامی دیسک‌های کوچکتر به میله‌ی B منتقل شوند. پس از تمام شدن این مرحله، دیسک بزرگ را از میله‌ی A به میله‌ی C منتقل کرده و مجددا به کمک میله‌ی A تمامی دیسک‌های میله‌ی B را به میله‌ی C منتقل می‌کنیم. پس به طور خلاصه می‌توان گفت:

مرحله‌ی یک: n-1 دیسک بالایی میله‌ی مبدأ با شرایط ذکر شده و به کمک میله‌ی C به میله‌ی B منتقل می‌شوند.

```
hanoi(from, _using, to, num_of_discs - 1);
```

مرحله‌ی دو: بزرگترین دیسک از میله‌ی مبدأ به میله‌ی مقصد منتقل می‌شود.

```
move(from, to);
```

مرحله‌ی سه: n-1 دیسک میله‌ی B با کمک گرفتن از میله‌ی A به میله‌ی مقصد منتقل می‌شوند.

```
hanoi(_using, to, from, num_of_discs - 1);
```

می‌بینیم که توانستیم عملیات جابجا کردن n دیسک را به دو عملیات مشابه ولی با اندازه‌ی کمتر و یک عملیات ساده تقسیم کنیم. برنامه‌ی زیر، ترتیب حرکت‌ها را چاپ می‌کند:

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

// definition of the three pegs
vector<int> a;
vector<int> b;
vector<int> c;

void print_peg(vector<int> peg)
{
    for (int i = 0; i < peg.size(); i++)
        cout << peg[i] << ' ';
    cout << '\n';
}

void print_pegs()
{
    cout << "A: ";
    print_peg(a);
    cout << "B: ";
    print_peg(b);
    cout << "C: ";
    print_peg(c);
    cout << endl;
}

void move(vector<int>& from_peg, vector<int>& to_peg)
{
    to_peg.push_back(from_peg.back());
    from_peg.pop_back();
    print_pegs();
}


void hanoi(vector<int>& from, vector<int>& to,
           vector<int>& _using, int num_of_discs)
{
    if (num_of_discs == 1)
        move(from, to);
    else {
        hanoi(from, _using, to, num_of_discs - 1);
        move(from, to);
        hanoi(_using, to, from, num_of_discs - 1);
    }
}
```

```
int main()
{
    int num_of_discs;
    cout << "How many discs? ";
    cin >> num_of_discs;

    for (int i = num_of_discs; i >= 1; i--)
        a.push_back(i);

    print_pegs();
    hanoi(a, b, c, num_of_discs);
}
```

در این کد سه میله به نام‌های a و b و c داریم که میله‌ی a در ابتدا حاوی n قرص با اندازه‌های متفاوت است طوری که از بالا به پایین اندازه‌ی قرص‌ها زیاد می‌شود. هدف منتقل کردن تمام قرص‌ها به میله‌ی b است با این شرط که اولاً هربار فقط یک قرص منتقل شود و ثانیاً هیچ‌وقت یک قرص روی قرص کوچک‌تر قرار نگیرد. از میله‌ی c نیز می‌توان کمک گرفت.

واضح است که تابع بازگشتی فوق کمترین تعداد حرکت را چاپ می‌کند. چرا که برای جابجا کردن بزرگترین دیسک از پایین میله‌ی A، بقیه‌ی دیسک‌ها باید در میله‌ی B باشند. فقط در این صورت این دیسک جابجا می‌شود. در فراخوانی‌های بعدی، دیسک دوم از نظر بزرگی جابجا می‌شود و الی آخر. پس در این فراخوانی‌ها جابجایی بیهوده‌ای صورت نمی‌گیرد. نیز توالی حرکت‌ها برای هر n منحصر بفرد است. یعنی برای یک n مشخص، دو توالی متمایز از جابجایی‌ها وجود ندارد که تعداد جابجایی آنها کمتر یا مساوی این حالت باشد.

تحلیل پیچیدگی زمانی مسئله‌ی برج هانوی: در حالت کلی می‌خواهیم بدانیم اگر تعداد دیسک‌ها n باشد، کمترین تعداد حرکت برای جابجا نمودن دیسک‌ها چقدر است؟

فرض کنید T(n) تعداد حرکت‌های لازم جهت انتقال n دیسک به مقصد باشد. بر اساس توضیحات فوق، تعداد T(n-1) حرکت برای انتقال n-1 دیسک به میله‌ی کمکی، یک حرکت برای انتقال بزرگترین دیسک به میله‌ی مقصد و مجددا T(n-1) حرکت برای انتقال n-1 دیسک موجود در میله‌ی کمکی به میله‌ی مقصد نیاز است. پس:

$$T(n) = 2T(n-1)+1$$

برای حل این رابطه‌ی بازگشتی فرض کنید که

$$U(n) = T(n)+1$$

لذا:

$$U(n) = 2T(n-1)+1+1$$
$$U(n) = 2T(n-1)+2$$
$$U(n) = 2(T(n-1)+1)$$
$$U(n) = 2U(n-1) \Rightarrow U(n) = 2^n$$

پس:

$$T(n)+1 = 2^n \Rightarrow T(n) = 2^n - 19$$

مرتبه‌ی اجرایی این الگوریتم $O(2^n)$ است که چندان مرتبه‌ی مطلوبی به نظر نمی‌رسد. اما همانگونه که بحث شد، این روش حداقل تعداد حرکت‌های ممکن را می‌دهد؛ و هرگز نمی‌توان روش دیگری با مرتبه‌ی پایین‌تر برای حل آن یافت.

حل غیربازگشتی مسئله‌ی برج هانوی: مسئله‌ی برج هانوی علاوه بر روش تابع بازگشتی، راه حل‌های غیربازگشتی نیز دارد. تا به اینجا مشخص شده است که بهترین راه حل برای جابجا کردن n دیسک، به تعداد نمایی حرکت نیاز دارد. در نتیجه مرتبه‌ی راه حل‌های آن در بهینه‌ترین حالت - چه بازگشتی و چه غیربازگشتی - از مرتبه‌ی 2^n خواهد بود. اما آنچه که راه حل بازگشتی و غیربازگشتی را از هم متمایز می‌کند، مرتبه‌ی فضای مصرفی آن است.

حل بازگشتی مسئله، فراخوانی‌های تو در تو و فضای پشته از مرتبه‌ی $O(n)$ نیاز دارد. در حالی که می‌توان با استفاده از روش غیربازگشتی این مرتبه را به $O(1)$ کاهش داد. البته این مسئله تنها دلیل بررسی روش غیربازگشتی نیست. تبدیل مرتبه‌ی مصرف فضا از $O(n)$ به $O(1)$ زمانی که مرتبه‌ی اجرای الگورینم $O(2^n)$ است، چندان قابل توجه نیست. دلیل دیگر می‌تواند این باشد که برخی زبان‌های برنامه‌نویسی از فراخوانی بازگشتی توابع پشتیبانی نمی‌کنند و مجبور به استفاده ار روشهای غیربازگشتی هستند. اما دلیل اصلی این است که با بررسی این روش‌ها، تمرین کوچکی برای تبدیل الگوریتم‌های بازگشتی به غیربازگشتی انجام می‌دهیم.

جهت مطالعه‌ی بیشتر رجوع شود به:

- بررسی مسئله‌ی برج هانوی و روش‌های حل بازگشتی و غیربازگشتی آن به همراه کد به زبان C++

- Tower of Hanoi - Wikipedia, the free encyclopedia
- New fast iterative computer algorithms for the Tower of Hanoi puzzle
- Data Structures and Algorithms - TOWERS OF HANOI