

# **PROGRAMOVANIE**

**v C++**



Cimrman se však nespokojuje jen s kritikou, ale dává příklad vlastními tvůrčími činy. Tady je nutno přiznat, že řada jeho pohádek vzbudila odpor, zejména u dětí.

---

— *Divadlo Járy Cimrmana*  
*Dlouhý, Široký a Krátkozraký*



Moje deti sa ma raz spýtali, ako sa programuje. Asi čakali odpoveď na menej ako minútu, ja som sa však rozhodol im napísat krátky tutoriál. Krátky tutoriál sa nakoniec trochu rozrástol, zistil som, že to neviem povedať kratšie. Výsledok je tu, potesím sa, ak si ho niekto prečíta. Text aj riešenia úloh sú voľne prístupné na Githube na adrese <https://github.com/pocestny/programovanie.git>.



Aby sme mohli rýchlo začať programovať, spravíme si základnú kostru programu. Čo v nej je si vysvetlíme neskôr, zatiaľ ju budeme používať bez vysvetlenia:

```

1 #include <iostream>
2 using namespace std;
3 int main() {
4     // riadok začínajúci sa // je komentár
5     // tu bude nás program
6 }
```

Na spustenie sa dá použiť nejaké prostredie (napr. [Code::Blocks](#)<sup>1</sup>, [MS Visual Studio](#)<sup>2</sup>, ...), alebo linuxový terminál. V hocijakom editore (napr. [Atom](#)<sup>3</sup>) napíš súbor s programom, ktorý nazvi napr. `program.cc` a skompiluj ho napr. tak, že v termináli napíšeš `g++ program.cc -o program`. Týmto hovoríš komplátoru<sup>4</sup> `g++`, aby zobrajal program `program.cc` a vyrobil spustiteľný súbor (binárku) `program`. Potom môžeš spustiť `./program` Keby program mal chybu, komplátor vypíše chybovú hlášku a binárku nevyrobí. Keby si napríklad namiesto `using` napísal `fusakla`, chybová hláška by mohla vyzeráť nejak takto:

```
program.cc:2:1: error: 'fusakla' does not name a type
  2 | fusakla namespace std;
    | ^~~~~~
```

Vidíme, že chyba je v súbore `program.cc` na riadku 2 a nasleduje aj opis chyby. Prostredia ako [Code::Blocks](#) chybu zvyčajne vypíšu v samostatnom okne a zvýraznia v teste programu. Keď už vieš, ako program skomplílovať a spustiť, môžeme začať programovať.

Základný stavebný prvok programu je *výraz*. Výraz je príklad, ktorý sa počas behu programu vyráta a zistí sa jeho výsledok (*hodnota výrazu*). Napr.  $3*(2+5)$  je výraz, ktorého hodnota (výsledok) je 21. Druhý stavebný prvok je *pričaz*. Pričaz hovorí, čo sa má s hodnotou výrazu urobiť. Pričaz je vždy ukončený bodkočiarkou. Najjednoduchší pričaz nerobí nič:  $3*(2+5)$ ; hovorí *Vypočítaj priklad*  $3 \cdot (2 + 5)$  a výsledok zahod. Pričaz na vypísanie hodnoty na konzolu je `cout <<` takže `cout << 3*(2+5);` znamená *Vypočítaj priklad*  $3 \cdot (2 + 5)$  a výsledok vypíš. Špeciálny výraz je `endl`, ktorého hodnota je 'koniec riadku'.

Náš prvý program vyzerá takto:

```

1 #include <iostream>
2 using namespace std;
3 int main() {
4     cout << 3 * (2 + 5);
5     cout << endl;
6 }
```

Keď ho spustíš, vypíše sa [21](#).

Samozrejme nechceme, aby sme pre každý príklad písali samostatný program, ale chceme, aby jeden program rátal veľa rôznych príkladov. Na to slúžia *premenné*. Premenná je krabička, ktorá má meno a môže v nej byť uložená nejaká hodnota. Každú premennú treba najprv vyrobiť a pritom treba povedať, aké hodnoty v nej môžu byť uložené (tomu sa hovorí *typ* premennej). Základný typ je `int`. Premenné typu `int` vedia ukladať celé čísla (*integer*). Meno typu je zároveň aj príkaz na vyrobenie premennej. Takže `int x;` je príkaz, ktorý

<sup>1</sup><https://www.codeblocks.org/>

<sup>2</sup><https://visualstudio.microsoft.com>

<sup>3</sup><https://atom.io/>

<sup>4</sup>Kompilátor je program, ktorý z textu programu vyrobí spustiteľný kód. V linuxe sú najrozšírenejšie komplátory `g++` a `clang++`. Samozrejme, komplátor je program, ako každý iný a treba ho mať nainštalovaný, aby fungoval.

## Rýchly úvod

hovorí *Vyrob premennú, ktorá sa bude voľať x a budú sa v nej dať ukladať celé čísla*. Premennú si môžeš nazvať hocijako, ale meno sa musí skladať z veľkých a malých písmen (bez diakritiky) a podčiarkovníkov (\_). Môžu v ňom byť aj čísla, ale nesmie sa číslom začínať. Napr. u3\_prachDoma je dobré meno premennej. Príkaz = slúži na uloženie výsledku do premennej. Takže  $x = 3 * (2 + 5)$ ; hovorí *Vypočítaj príklad  $3 \cdot (2 + 5)$  a výsledok ulož do premennej, ktorá sa volá x*. Po vykonaní tohto príkazu teda v premennej x bude uložená hodnota 21. Samotné meno premennej je výraz (t.j. príklad), ktorého hodnota (t.j. výsledok) je číslo, ktoré je v nej práve uložené. Čo urobí nasledovný program?

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int x;
5     x = 2 + 5;
6     cout << 3 * x;
7     cout << endl;
8     x = 9 - 5;
9     cout << 3 * x;
10    cout << endl;
11 }
```

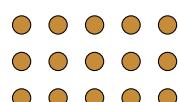
Prvý príkaz na riadku 4 vyrobí premenňu x, príkaz na riadku 5 vyráta príklad  $2 + 5$  a výsledok (teda 7) uloží do premennej x. Príkaz na riadku 7 ráta príklad *Vynásob číslo 3 a obsah premennej x*. V premennej x je práve uložená sedmička, takže výsledok príkladu je  $3 \cdot 7 = 21$  a to je číslo, ktoré sa vypíše a vzápäť sa vypíše koniec riadku. Nasledujúci príkaz na riadku 8 do premennej x uloží výsledok príkladu  $9 - 5$ , teda 4. Príkaz na riadku 9 ráta to isté, čo príkaz na riadku 6, t.j. *Vynásob číslo 3 a obsah premennej x*. Teraz je ale v premennej x uložené číslo 4, takže výsledok príkladu je 12.

Posledný príkaz z tejto časti je `cin>>`, ktorý čaká, kým na klávesnici napíšeš číslo (a stlačíš **Enter**) a toto číslo uloží do premennej. Kolko vypíše nasledovný program, ak mu napíšeš číslo 4? Skús si to najprv vyrátať sám a potom vyskúšať, či si mal pravdu.

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int x;
5     cin >> x;
6     x = x + 5;
7     cout << 3 * x;
8     cout << endl;
9 }
```

Zopakujme si to. Výraz je príklad, ktorý má výsledok (hodnotu). Videli sme v ňom vystupovať sčítanie, odčítanie, násobenie, čísla a mená premenných. Keď sa počíta výsledok a vo výraze je meno premennej, zoberie sa hodnota, ktorá je v nej práve uložená. Toto je dôležité: musíme rozlišovať, medzi tým, čo vieme v čase, keď píšeme program (*compile-time*) a tým, čo vie program, keď už pracuje (*runtime*). Napr. ak v predchádzajúcom programe zadáme číslo 4, vypíše sa  $3 \cdot (4 + 5)$ , teda 27. Ak zadáme číslo 1, vypíše sa  $3 \cdot (1 + 5)$ , teda 18. Pri písaní programu vieme, že ak napíšeme na vstupe hocjaké číslo a, pred výpisom bude v premennej x uložená hodnota  $a + 5$ , a preto sa vypíše výsledok príkladu  $3 \cdot (a + 5)$ . Nevieme ale dopredu povedať, aké číslo bude v premennej x uložené, to bude známe až počas behu programu.

**Úloha 1.** Máme obdĺžnik z kamienkov ako na obrázku vpravo. Napiš program, ktorý prečíta zo vstupu dve čísla: kolko kamienkov je v riadku a kolko v stĺpci obdĺžnika. Program má vypísať, kolko kamienkov je v celom obdĺžniku. Obrázok vyššie má 5 kamienkov v riadku a 3 riadky a je v ňom 15 kamienkov.



## Podmienky

2

Pretože v čase, keď píšeme program, nevieme, čo bude v ktorej premennej uložené, potrebujeme príkazy, ktoré nám umožnia urobiť rôzne veci, podľa toho, čo sa v premenných práve nachádza. Predtým si ti ale ešte predstavíme iný typ. Okrem typu `int`, ktorý znamená celé číslo, je aj typ `bool` (*boolean*), ktorý môže mať dve hodnoty: `true` (pravda) a `false` (nepravda). Rovnako, ako môžeme mať príklady (výrazy) s celými číslami, môžeme mať aj príklady s pravdivostnými hodnotami. Napr.  $3 > 5$  je výraz, ktorého hodnota je `false`. Tu je niekoľko operácií, ktoré vieme vo výrazoch používať. Operácie vľavo zoberú dve hodnoty typu `int` a vrátia novú hodnotu `int`. Mali by byť jasné, len si treba všimnúť, že delenie / je vždy celočíselné a zvyšok po delení sa dá zistíť pomocou %.

V pravej časti sú operácie, ktoré vrátia hodnotu `bool`. Niektoré z nich porovnávajú čísla typu `int` ( $<$ ,  $>$ ,  $\leq$ ,  $\geq$ ,  $\neq$ ,  $\equiv$ )<sup>1</sup> a iné ( $\&\&$ ,  $\|$ ,  $!$ ) kombinujú hodnoty typu `bool`:

výsledok je <code>int</code>		výsledok je <code>bool</code>	
<code>+</code>	<code>-</code>	<code>*</code>	<code>=</code>
<code>/</code>		<code>&gt;</code>	<code>&gt;=</code>
<code>%</code>		<code>&lt;</code>	<code>&lt;=</code>
		<code><math>\neq</math></code>	<code><math>\neq</math></code>
		<code><math>\&amp;\&amp;</math></code>	<code><math>\&amp;\&amp;</math></code>
		<code><math>\ </math></code>	<code><math>\ </math></code>
		<code>!</code>	<code>!</code>

Pri používaní logických spojok  `$\&\&$`  a  `$\|$`  má vždy  `$\&\&$`  prednosť pred  `$\|$`  (podobne ako násobenie pred sčítaním), t.j. `a  $\|$  b  $\&\&$  c` znamená `a  $\|$  (b  $\&\&$  c)`.

Príkaz `if` vyhodnotí podmienku (ktorá musí byť napísaná v zátvorkách) a vykoná iný príkaz iba vtedy, ak výsledok je `true`. Napr. `if (x > 5) x = 5;` hovorí *Zober si hodnotu teraz uloženú v premennej x a zisti, či je viac ako 5. Ak je to pravda, ulož do x hodnotu 5. Inak neurob nič.* Aby nebolo treba písať veľakrát tú istú podmienku, existuje tzv. zložený príkaz. Tieto dva programy sú rovnaké:

```
1 if (a > 4) x = a;  
2 if (a > 4) x = x + 5;  
3 if (a > 4) cout << x;
```

```
1 if (a > 4) {  
2     x = a;  
3     x = x + 5;  
4     cout << x;  
5 }
```

Vyrábanie zložených príkazov sa dá použiť kedykoľvek, nie len v príkaze `if`. Kedykoľvek napíšeme viac príkazov do kučeravých zátvoriek ({}), správajú sa ako jeden príkaz. Na každom mieste, kde môžeme použiť príkaz, môžeme použiť konštrukciu s kučeravými zátvorkami. Jedným takým miestom je aj začiatok programu. V našej schéme máme `int main()` a za ním celý program v kučeravých zátvorkách: celý program je vlastne jeden zložený príkaz. O vytváraní premenných si viac povieme neskôr, nateraz skús nevytvárať premenné vovnútri zložených príkazov<sup>2</sup>.

Príkaz `if` má aj rozšírenú verziu so slovom `else` (inak):

```
1 if (a >= 0)  
2     cout << a;  
3 else  
4     cout << -a;
```

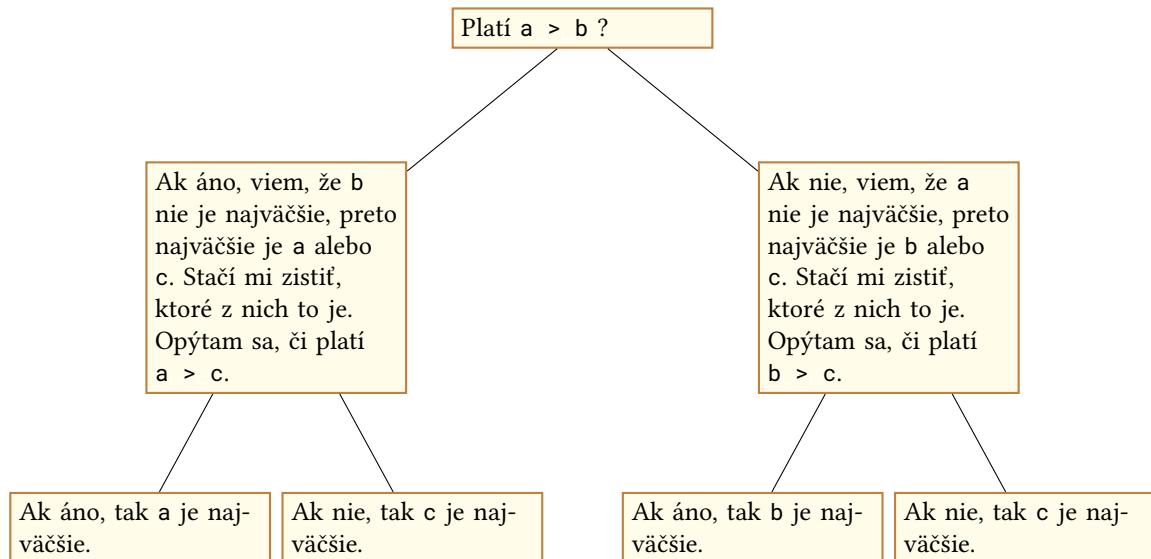
Opäť by sme namiesto každého príkazu mohli použiť zložený príkaz v kučeravých zátvorkách.

<sup>1</sup>Všimni si, že pre porovnanie na rovnosť musíme použiť dva znaky `==`, lebo `=` je príkaz na uloženie výsledku do premennej

<sup>2</sup>Alebo sa dobre zamysli, ako sa to asi môže správať.

## Podmienky

Vnáraním podmienok vieme vytvárať rozhodovacie stromy. Povedzme, že máme tri premenné  $a$ ,  $b$ ,  $c$  a chceme zistiť, ktorá je najväčšia. To vieme urobiť tak, že sa budeme postupne pýtať:



Toto viem zapísť do programu :

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int a, b, c;
5     cin >> a >> b >> c;
6     if (a > b) {
7         if (a > c)
8             cout << a;
9         else
10            cout << c;
11     } else {
12         if (b > c)
13             cout << b;
14         else
15             cout << c;
16     }
17     cout << endl;
18 }
```

Tu som použil skrátený zápis: namiesto `int a;` `int b;` `int c;` môžem napísať `int a, b, c;` a namiesto

```
1 cin >> a; cin >> b; cin >> c;
```

môžem písat `cin >> a >> b >> c;` Tiež si si už asi všimol, že nezáleží na tom, ako sú v texte programu medzery a riadky. Je rozumné mať text *odsadený* tak, aby sa dal dobre čítať: každý príkaz na zvláštnom riadku a vnorené príkazy majú na začiatku riadku viac medzier, aby začínali viac vpravo. Rovnako je dobré nazvať premenné tak, aby bolo z názvu jasné, čo je v nich uložené. Ale nie je to nutné. Aj toto<sup>3</sup> je správny program, ale nie je ľahké zistiť, čo vlastne robí:

<sup>3</sup>Program je zo súťaže [www.ioccc.org](http://www.ioccc.org) o najnečitateľnejší C/C++ program (<https://www.ioccc.org/2011/eastman/hint.html>).

```

1 #include <stdio.h>
2 #include <math.h>
3 #include <unistd.h>
4 #include <sys/ioctl.h>
5
6     main() {
7         short a[4];ioctl
8             (0,TIOCGWINSZ,&a);int
9             b,c,d=*a,e=a[1];float f,g,
10            h,i=d/2+d%2+1,j=d/5-1,k=0,l=e/
11            2,m=d/4,n=.01*e,o=0,p=.1;while (
12 printf("\x1b[H\x1B[?251"),!usleep(
13 79383)){for (b=c=0;h=2*(m-c)/i,f=-
14 .3*(g=(l-b)/i)+.954*h,c<d;c+=(b=++
15 b%e)==0)printf("\x1B[%dm\u2022",g*g>1-h
16 *h?c>d-j?b<d-c||d-c>e-b?40:100:b<j
17 ||b>e-j?40:g*(g+.6)+.09+h*h<1?100:
18 47:((int)(9-k+(.954*g+.3*h)/sqrt
19 (1-f*f))+(int)(2+f*2))%2==0?107
20 :101);k+=p,m+=o,o=m>d-2*j?
21 -.04*d:o+.002*d;n=(l+-
22 n)<i||l>e-i?p=-p
23 , -n:n;}}

```

**Úloha 2.** Napíš program, ktorý načíta tri čísla a vypíše prostredné z nich.

**Úloha 3.** Napíš program, ktorý načíta číslo mesiaca (1 pre január, 2 pre február, atď.) a vypíše číslo ročného obdobia (1 pre jar, 2 pre leto, 3 pre jesen a 4 pre zimu).

### 3 Cykly

Videli sme už príkazy na načítanie a vypísanie, uloženie výsledku do premennej a vyhodnotenie podmienky. Ďalšie užitočné príkazy sú tzv. *cykly*, ktoré nejaký príkaz opakujú viackrát. Predstavíme si jeden z nich: `while`. Príkaz `while` má, podobne ako `if`, v zátvorkách napísanú podmienku a za ňou príkaz. `while` robí to, že stále dokola vyhodnocuje podmienku a kým je splnená, vykonáva príkaz. Podobne ako pri `if` sa dá použiť zložený príkaz: `v {}` uzavretých viac príkazov sa správa ako jeden. Čo robí tento program?

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int x;
5     x = 1;
6     while (x < 11) {
7         cout << x << endl;
8         x = x + 1;
9     }
10 }
```

Najprv vyrobí premennú `x` a uloží do nej jednotku. Potom dokola robí toto: *Pozri sa, či  $x < 11$ . Ak áno, vypíš `x`, ulož do `x` číslo o 1 väčšie, ako tam bolo doteraz a celé to zopakuj.* Cyklus teda postupne vypíše riadky s číslami 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 a potom skončí. Jedna vec, na ktorú si treba dávať pri cykle pozor, je, že sa ľahko môže stať, že nikdy neskončí a program sa zacyklí. Keby sme napr. zabudli napísať riadok `x = x + 1;` `x` sa nikdy nezmení, `x < 11` bude platiť stále a program bude donekonečna vypisovať riadok s číslom 1.

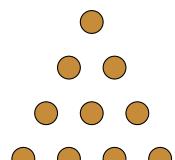
Urobme ešte jeden príklad. Dajme tomu, že najprv napišeš jedno prirodzené číslo  $n$  a potom  $n$  prirodzených čísel. Program má zistiť, ktoré z nich je najväčšie. Napr. ak by si na vstupe napísal 5 6 3 9 12 1, program má vypísať 12. Ako to naprogramovať? Najprv si vyrobíme premennú `int n`; kde si uložíme počet čísel `cin >> n`; Potom potrebujeme  $n$ -krát zopakovať toto: *Prečítaj číslo `x`. Ak je väčšie, ako najväčšie číslo, ktoré sme doteraz videli, tak najväčšie videné číslo bude odteraz `x`. Inak sa najväčšie videné číslo nezmení.* Ako môžeme niečo zopakovať  $n$ -krát? Urobíme cyklus, v ktorom vždy odráťame z `n` jednotku. Keď nám v `n` ostane nula, cyklus skončíme.

Naprogramovať zvyšok už nie je ľažké: budeme mať premennú `int max`; v ktorej si budeme pamätať doposiaľ najväčšie videné číslo. Vnútri cyklu načítame jedno číslo zo vstupu a porovnáme ho s `max`. Ostáva vyriešiť jediný problém: čo dať na začiatku do premennej `max`? A čo sa stane, ak do nej nedáme nič? V tom prípade hovoríme, že premenná má *ne definovanú hodnotu* a to znamená, že v nej môže byť uložené čokoľvek. Nikdy nepíš program, ktorý používa premennú, do ktorej predtým nič nezapísal. V našom prípade vieme, že budeš zadávať iba prirodzené čísla. Keď teda na začiatku do `max` uložíme nejaké záporné číslo, máme istotu, že už prvé načítané číslo bude väčšie. Toto je užitočná technika, ktorej sa hovorí *zarázka (sentinel)*. Celý program teda bude vyzerať takto:

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int n, max, x;
5     max = -1;
6     cin >> n;
7     while (n > 0) {
8         cin >> x;
9         if (x > max) max = x;
10        n = n - 1;
11    }
12    cout << max << endl;
13 }
```

**Úloha 4.** Používateľ napiše najprv jedno prirodzené číslo  $n$  a potom  $n$  prirodzených čísel. Napiš program, ktorý vypíše ich súčet.

**Úloha 5.** Znova máme kamienky, tentokrát uložené do trojuholníka. Napiš program, ktorý prečíta zo vstupu jedno číslo: kolko kamienkov je v spodnom riadku. Program má vypísať, kolko kamienkov je v celom trojuholníku. Vieš ho napiisať aj bez použitia cyklu?



**Úloha 6.** Napíš program pre takúto úlohu. Používateľ postupne zadáva čísla. Keď napíše párne číslo, vypíš to isté číslo. Keď napíše nepárne, vypíš o jedno väčšie. Keď napíše -1, program skončí.

**Úloha 7.** Používateľ najprv zadá číslo  $n$  a potom  $n$  rôznych čísel. Napíš program, ktorý zistí, ako ďaleko sú od seba najväčšie a najmenšie číslo.

**Úloha 8.** Používateľ najprv napíše prirodzené číslo  $a$  a potom začne písť prirodzené čísla (nevieme, kolko). Nakoniec napíše -1. Napíš program, ktorý zistí, kolkokrát je medzi napísanými číslami číslo  $a$ . Napr. pre vstup 3 9 3 8 7 3 6 6 7 3 3 7 8 -1 má byť odpoveď 4, lebo trojka sa v zadaných číslach nachádza štyrikrát.

**Úloha 9.** Zoberme si takúto hru: myslí si číslo. Ak je párne, vydel' ho dvoma. Ak je nepárne, vynásob ho troma a pripočítaj 1. Toto opakuj, až kým sa nedostaneš k jednotke. Napríklad keď začneš s číslom 12 budeš postupne dostávať čísla 6, 3, 10, 5, 16, 8, 4, 2, 1.<sup>1</sup>

Napíš program, ktorý prečíta zo vstupu jedno číslo  $a$  a vypíše, ako dlho trvá, kým sa dostaneš k jednotke a aké najväčšie číslo pri tom stretneš. Pre 12 má program vypísať 9 a 16, lebo k jednotke sa dostaneš po 9 krokoch a najväčšie číslo, ktoré stretneš je 16. Pre 27 má vypísať 111 a 9232.

(Zistíť, či je číslo párne, sa dá jednoducho pomocou operácie %, ktorá dáva zvyšok po delení. Ak teda chceš urobiť nejaký príkaz, iba ak  $n$  je párne, stačí napísať `if (n % 2 == 0) ...`)

**Úloha 10.** Používateľ postupne zadáva čísla. Napíš program, ktorý po každom zadanom číslu vypíše súčet všetkých čísel od začiatku až doteraz. Keď používateľ zadá -1, program skončí.

**Úloha 11.** Fibonacciho čísla sú postupnosť čísel, ktorú vieme zostrojiť takto: prvé dve Fibonacciho čísla sú 0, 1. Ďalšie číslo vyrobím tak, že sčítam dve posledné. Celá postupnosť teda vyzerá 0, 1, 1, 2, 3, 5, 8, 13, 21, ...<sup>2</sup> Napíš program, ktorý pre zadané  $n$  vyráta  $n$ -té Fibonacciho číslo.

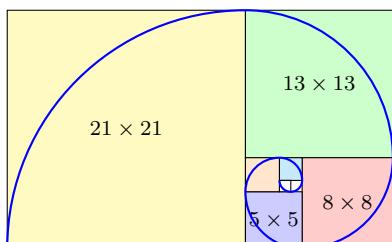
Podobne ako podmienky, aj cykly môžu byť vnorené: príkaz, ktorý sa vykonáva v cykle, môže opäť obsahovať cyklus. Pozrime na tento príklad: používateľ na vstupe napíše číslo  $n$ . Program má vypísať šachovnicový vzor rozmerov  $n \times n$ , pričom sa v ňom striedavo opakujú 0 a 1. Napríklad pre vstup 5 by program vypísal

```
01010
10101
01010
10101
01010
```

Ako naprogramovať niečo takéto? Začiatok je ľahký: budeme mať premennú `int r`, v ktorej vždy bude uložené, ktorý riadok robíme. Začneme s tým, že do nej uložíme nulu a v cykle ju budeme postupne zväčšovať až po  $n$ . V rámci cyklu potrebujeme vypísať jeden riadok. Riadok sa skladá z  $n$  znakov, preto môžeme použiť inú premennú, `int s`, v ktorej bude uložené, ktorý stĺpec v riadku práve robíme. Na začiatku riadku uložíme do  $s$

<sup>1</sup>V matematike je známa Collatzova hypotéza, ktorá hovorí, že nech začneš z hocjakého čísla, vždy sa nakoniec k jednotke dostaneš. Matematikov veľmi hnevá, že stále nevedia Collatzovu hypotézu dokázať. Na druhej strane, zatiaľ ani nenašli žiadne číslo, ktoré by sa po čase k jednotke nedostalo, a to už vyskúšali všetky čísla až po 295147905179352825856.

<sup>2</sup>Fibonacciho čísla sú veľmi zaujímavé. Pomer dvoch za sebou idúcich Fibonacciho čísel sa blíži k tzv. *zlatému rezu*. V prírode sa často vyskytujú v rôznych útvarech, napr. pri špirálach:



## Cykly

---

nulu a v cykle budeme  $n$ -krát pripočítavať jednotku a budeme striedavo vypisovať nulu a jednotku. Vystrieďame ich ľahko: ak  $x$  je 0, tak  $1-x$  je 1 a naopak. Posledná otázka: ako zistíme prvý znak v riadku? Premenná  $r$  udáva číslo riadku (začína od 0), takže stačí na začiatku zobrať zvyšok po delení 2. Celý program vyzerá takto:

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int r, s, x, n;
5     cin >> n;
6     r = 0;
7     while (r < n) {      // tento cykus sa zopakuje raz pre každý riadok
8         x = r % 2;        // x je začiatočný znak
9         s = 0;
10        while (s < n) {   // tento cyklus vypisuje znaky v riadku
11            cout << x;
12            x = 1 - x;      // zmení vypisovaný znak
13            s = s + 1;
14        }                  // koniec cyklu pre znaky v riadku
15        cout << endl;
16        r = r + 1;
17    }                      // koniec cyklu pre riadky
18 }
```

**Úloha 12.** Napíš program, ktorý načíta jedno číslo  $n$  a vypíše štvorec z núl orámovaný jednotkami, ktorý má  $n$  riadkov. Pre  $n = 5$  má vzor vyzeráť takto:

```
11111
10001
10001
10001
11111
```

**Úloha 13.** Napíš program, ktorý načíta jedno číslo  $n$  a vypíše štvorec z núl a jednotiek, ktorý má  $n+1$  riadkov. Pre  $n = 5$  má vzor vyzeráť takto:

```
00000
00001
00011
00111
01111
11111
```

**Úloha 14.** Napíš program, ktorý načíta jedno číslo  $n$  a vypíše trojuholníkový vzor z núl a jednotiek, ktorý má  $n$  riadkov. Pre  $n = 5$  má vzor vyzeráť takto:

```
000010000
000111000
001111100
011111110
111111111
```

Chceme napísať takýto program: najprv napíšeš číslo  $n$  a potom  $n$  prirodzených čísel. Nakoniec napíšeš ešte jedno číslo  $a$ . Program má vypísať najmenšie zo zadaných čísel, ktoré je väčšie ako  $a$  (alebo vypíše  $a$ , ak žiadne z čísel nie je väčšie). Napríklad:

Vstup  
5  
1 10 100 1000 10000  
427

Výstup  
1000

Problém je v tom, že keď načítavame čísla zo vstuпу, ešte nevieme, ako ich bude treba spracovať, takže by sme si ich potrebovali uložiť do premenných. Keby sme počet čísel poznali v čase písania programu (napr. by bolo povedané, že vždy bude presne päť čísel), mohli by sme si vyrobiť premennú pre každé číslo. Počet čísel je ale známy až počas behu programu. Použijeme preto *pole* premenných. Pole je vlastne veľa premenných, ktoré majú spoločné meno a rozlišujú sa číslom (tzv. *indexom*). Pole sa vyrobí podobne, ako jedna premenná, ale v hranatých zátvorkách je uvedený počet premenných. Tak napr. `int x[10]`; vyrobí pole premenných  $x[0]$ ,  $x[1]$ , až  $x[9]$ , z ktorých každá vie mať uložené celé číslo<sup>1</sup>. Výhoda je, že keď chceme použiť nejakú premennú z poľa, nemusíme písat jej meno priamo, napr.  $x[5]$ , ale na mieste indexu môžeme použiť hocjaký výraz. Takže môžeme napísať  $x[2+3]$  a tiež dostaneme hodnotu z premennej  $x[5]$ . A samozrejme, výraz môže obsahovať aj mená premenných, takže  $x[i+3]$  znamená *Pozri sa, čo je práve teraz uložené v premennej i, pripočítaj k tomu 3, výsledok zober ako číslo premennej z poľa a vrát hodnotu, ktorá je v nej uložená*. Ak by v  $i$  bola uložená dvojka, tak  $x[i+3]$  vráti hodnotu uloženú v premennej  $x[5]$ . Pole nie je premenná, ale veľa premenných. Preto aj príkaz priradenia funguje na jednotlivých premenných, ale nie na celom poli. Ak by sme mali premenné `int a[10], b, c[10], d; môžeme napísať b=d; aj a[d+3]=b; aj c[b+a[d]]=a[d]+c[3]; ale nemôžeme napísať a=c; a už vôbec nie a=b`; Rovnako, keďže pole nie je premenná, tak nemôžeme používať ani operácie ako  $a+c$  alebo  $a==c$ , ale musíme ich robiť na jednotlivých premenných poľa.

Predpokladajme, že máme pole `int a[10]`; a premennú `int x`. Čo spraví príkaz `x=a[22]`? Kompilátor by mal hádam vyhlásiť chybu, ale on nie. Premenné sú totiž krabičky, ktoré sú v pamäti uložené jedna za druhou. Príkazom `int a[10]`; komplilátor vyhradí v pamäti miesto na 10 premenných typu `int` a zapamätá si, kde je prvá z nich,  $a[0]$ . Ich počet sa nikde nezapamäta. Keď napíšeš  $a[5]$ , znamená to *piata premenná typu int za premenou a[0]*. Ak napíšeš  $a[22]$ , program sa pozrie do pamäte, kde by bola 23. premenná z poľa, keby to pole bolo dosť dlhé. Ak je kratšie, môže na tom mieste byť hocikto (napríklad nejaká iná premenná) a program sa začne správať veľmi podivne. Chyby tohto typu sa v programe veľmi ľahko hľadajú. Preto je dobré dávať veľký pozor, na aké miesta poľa pristupuješ.

Pri pristupovaní do poľa je príjemné, že výrazy typu `bool` sa vydelenie len dovtedy, kým nie je jasný výsledok, potom sa s počítaním prestane. Napr. pri výraze `false && ( zlozita_vec )` je hned po vydelení prvého `false` jasné, že výsledok bude `false`, preto sa `zlozita_vec` ani nevykoná. To umožňuje napísať do jednej podmienky kontrolu, či je index do poľa správny napr. napíšem `if (i>=0 && i<10 && a[i]==5)` a viem, že z poľa sa bude čítať iba vtedy, ak je premenná  $i$  zo správneho rozsahu.

Vráťme sa k nášmu príkladu. S pomocou poľa a cyklu si vieme zapamätať všetky zadané čísla:

```
1 int i, n;
2 cin >> n;           // prečítam počet čísel
3 int x[n];            // vyrábime pole potrebnej veľkosti
4 i = 0;
5 while (i < n) { // v cykle prečítame všetky prvky poľa
6     cin >> a[i];
7     i = i + 1;
8 }
```

<sup>1</sup>Všimni si, že prvá premenná má číslo 0, preto ak vyrábime 10 premenných, posledná má číslo 9.

Ked' už budeme mať všetky čísla uložené v poli, prečítame zo vstupu číslo  $a$ . Ak chceme zistiť najmenšie číslo, ktoré je väčšie ako  $a$ , urobíme si ďalšiu premennú  $\min$ , v ktorej si budeme pamätať doteraz nájdené číslo. V cykle sa pozrieme na každú premennú z poľa a zistíme, či je jej hodnota väčšia ako  $a$ . Ak áno a je zároveň menšia ako  $\min$ , aktualizujeme  $\min$ . Posledná vec: na začiatku potrebujeme do  $\min$  dať vhodnú zarázku. Môžeme použiť najväčšie číslo, ktoré si môžeme zistiť priebežne pri ukladaní vstupného poľa. Celý program by vyzeral takto:

```

1 #include <iostream>
2 using namespace std;
3 int main() {
4     int n, i, a, min;
5     cin >> n;
6     int x[n];          // vyrobíme pole správnej dĺžky
7     i = 0;
8     min = 0;
9     while (i < n) { // načítavame vstup a zároveň počítame maximum
10        cin >> x[i];
11        if (x[i] > min) min = x[i];
12        i = i + 1;
13    }
14    cin >> a;          // prečítame a
15    i = 0;
16    while (i < n) { // v cykle nájdeme hľadané číslo
17        if (x[i] > a && x[i] < min) min = x[i];
18        i = i + 1;
19    }
20    cout << min << endl;
21 }
```

**Úloha 15.** Používateľ zadá číslo  $n$ , potom  $n$  čísel a nakoniec číslo  $a$ . Napiš program, ktorý zistí, kol'kokrát bolo zadané číslo  $a$ . Napríklad

Vstup

9 3 9 3 1 3 2 3 4 5 3

Výstup

4

**Úloha 16.** Používateľ zadá číslo  $n$ , potom  $n$  čísel a nakoniec číslo  $a$ . Napiš program, ktorý vypíše všetkých  $n$  čísel, ale zväčšených o  $a$ .

**Úloha 17.** Používateľ zadá číslo  $n$  a potom zoznam  $n$  kariet s číslami. Dvaja hráči,  $A$  a  $B$  hrajú takúto hru. V každom kole sa  $A$  pozrie na prvú kartu a  $B$  na poslednú kartu (ak je v zozname iba jedna karta, je zároveň prvá aj posledná; ak je zoznam prázdný, hra sa skončila). Ak  $A$  vidí väčšie číslo, výsledok kola je 1 a  $B$  zahodí kartu z konca, ak  $A$  vidí menšie číslo, výsledok je -1 a  $A$  zahodí kartu zo začiatku, a ak vidia rovnaké, výsledok je 0 a zahodí sa karta z konca aj zo začiatku. Napiš program, ktorý prečíta začiatočný zoznam a vypíše postupne výsledky všetkých kôl. Napr. pre  $n = 3$  a vstup 1 2 3 sú výsledky kôl -1 -1 0.

**Úloha 18.** Používateľ zadá číslo  $n$  a potom  $n$  rôznych čísel. Napiš program, ktorý najprv vypíše najmenšie číslo a potom vždy najmenšie číslo, ktoré je väčšie, ako posledné vypísané. (Všimni si, že program vlastne vypíše všetky zadané čísla v utriedenom poradí.)

**Úloha 19.** Používateľ zadá číslo  $n$  a potom pole  $n$  prirodzených čísel. Napiš program, ktorý zistí najväčšie také číslo  $k$ , že sa v poli nachádza aspoň  $k$  čísel veľkosti aspoň  $k$ . Napr. pre vstup 9 3 9 1 1 2 2 3 4 5 je odpoveď 3, lebo sú tam aspoň tri čísla väčšie alebo rovné 3 (je tam 5 takých: 3, 9, 3, 4, 5), ale nie sú tam štyri čísla väčšie alebo rovné 4 (sú tam len tri: 9, 4, 5).

## Ďalšie cykly

5

Poznáme už príkaz cyklu `while` a úplne by nám stačil, aby sme s ním mohli naprogramovať všetko, čo chceme. Sú ale aj ďalšie príkazy cyklov, ktoré môžeme použiť, aby sme dostali kratší a čitateľnejší program. Veľakrát sme potrebovali prejsť pomocou cyklu postupne niečo prechádzkať. Na to sme vždy mali premennú, ktorú sme na začiatku nastavili (napr.) na nulu a v cykle sme ju zvyšovali nejak takto:

```
1 int i;
2 i = 0;
3 while (i < n) {
4     // niečo urob
5     i = i + 1;
6 }
```

Na skrátenie programu a zlepšenie čitateľnosti je možné vyrobenie premennej a priradenie do nej urobiť naraz, t.j. môžeme napísť

```
1 int i = 0;
2 while (i < n) {
3     // niečo urob
4     i = i + 1;
5 }
```

Tento typ programu sa dá čitateľnejšie zapísat iným príkazom cyklu, `for`. Príkaz `for` má v zátvorkách tri časti, oddelené bodkočiarkami. Prvá časť je príkaz, ktorý sa má vykonať na začiatku, druhá časť je podmienka ako vo `while` cykle a tretia časť je príkaz, ktorý sa má vykonať po každom vykonaní cyklu. Predchádzajúci príklad by mohol vyzeráť:

```
1 int i;
2 for (i = 0; i < n; i = i + 1) {
3     // niečo urob
4 }
```

Rovnako častý je aj príkaz `i = i + 1`, ktorý hovorí *Zober to, čo je uložené v premennej i, pričítaj 1 a výsledok ulož nasäť do i*. Tento príkaz je tak častý, že má vlastnú skratku, stačí napísat<sup>1</sup>`i++`. Cyklus tvaru `for (i = 0; i < n; i++) { ... }` je veľmi častý.

Príkaz `while` vyhodnocuje podmienku na začiatku, pred vykonaním príkazu. Niektory sa viac hodí, aby sa podmienka vyhodnocovala na konci. Na to slúži príkaz `do-while`. Dajme tomu, že chceme čítať čísla zo vstupu a vypisovať o jedno väčšie, až kým neprečítame -1. Môžeme to napísť takto:

```
1 int x;
2 do {
3     cin >> x;
4     cout << x + 1 << endl;
5 } while (x >= 0);
```

<sup>1</sup>Zvýšenie hodnoty premennej o 1 sa zvykne volať *inkrementovanie* a zníženie o 1 *dekrementovanie*. Zápis `i++` sa dá použiť aj ako výraz, napr. `if (i++ < 10) ...` znamená *Najprv zober hodnotu z premennej i. Premennú i vzápäť zvýš o 1 a pôvodnú hodnotu porovnaj s 10*. Podobne sa dá použiť aj `++i`, ktorý inkrementuje premenňu pred použitím. Ak je to vo forme príkazu, je to jedno, ale výraz `if (++i < 10) ...` znamená *K premennej i najprv pričítaj 1 a potom ju porovnaj s 10*. Rovnaký zápis funguje aj pre odčítanie, t.j. `--i` a `i--`. Ak chceme pridať inú hodnotu, ako 1, môžem napísat `+=`, napr. `i += 3` namiesto `i = i + 3`.

## Ďalšie cykly

---

Niekedy je dobré prerušíť vykonávanie cyklu. Na to slúžia príkazy **break** (skončí vykonávanie celého cyklu) a **continue** (skončí vykonávanie jednej iterácie a prejdi na ďalšiu). Napr. nasledujúci program kopíruje čísla zo vstupu na výstup, až kým neprečíta -1 a potom skončí:

```
1 while (true) {  
2     cin >> x;  
3     if (x == -1) break;  
4     cout << x << endl;  
5 }
```

Podobne nasledujúci program vypisuje iba nepárne čísla, až kým neprečíta -1:

```
1 do {  
2     cin >> x;  
3     if (x % 2 == 0) continue;  
4     cout << x << endl;  
5 } while (x != -1);
```

# Algoritmické úlohy a príkazový riadok

6

Všetky úlohy, ktoré si doteraz robil, vyžerali podobne: používateľ niečo napiše na vstupe (napríklad 7 čísel väčších ako 3), program s tým má urobiť presne popísanú činnosť (napríklad nájsť najväčšie číslo) a výsledok vypísat. Takýmto úlohám, ktoré zo vstupu robia výstup podľa presne stanovených pravidiel sa hovorí *algoritmické úlohy*. Sú dôležité preto, lebo pri nich vieme presne povedať, kedy je program správny: musí vypísat správny výstup pre všetky možné vstupy. Keby sme napr. v úlohe o hľadaní najväčšieho čísla mali program, ktorý skoro vždy napiše skutočne najväčšie číslo, ale ak je náhodou jedno z čísel na vstupe 12343, tak program sa zacyklí, tak takýto program by bol zlý (aj keby väčšinou fungoval správne). Aj keď programuješ väčší program, napr. hru, je dobré si ho rozdeliť na menšie, algoritmické, časti, ktoré sa dajú samostatne skontrolovať.

Ako sa dajú algoritmické úlohy kontrolovať? Väčšinou nie je možné spustiť program na všetky možné vstupy, lebo ich je privelá. Keďže majú presne definované, ako má vyzeráť výstup, je možné o nejakom programe aj matematicky dokázať, že je správny. To býva ale ľahšie, preto sa väčšinou programy *testujú*. Snažíme sa vybrať rôzne vstupy a na nich program spustíme. Keď na všetkých vstupoch dáva správne výsledky, veríme, že je dobrý. Nájsť dobrú sadu testovacích vstupov býva samo osebe dosť náročné. Je veľa miest, kde sa dajú algoritmické úlohy trénovať (napr. [testovac.ksp.sk](https://testovac.ksp.sk)<sup>1</sup>, [SPOJ](https://www.spoj.com)<sup>2</sup>, [CodeChef](https://www.codechef.com)<sup>3</sup>, [CodeForces](https://www.codeforces.com)<sup>4</sup> a ďalšie). Tieto stránky majú úlohy s možnosťou testovania: odošleš svoj program, na stránke sa skompiluje, spustí na veľa rôznych vstupoch a oznámi sa ti výsledok. Je aj veľa súťaží v riešení algoritmických úloh (napr. [KSP](https://www.ksp.sk)<sup>5</sup>, [slovenská<sup>6</sup>](https://www.slovenska6.sk) a [medzinárodná<sup>7</sup>](https://www.medzinardonad7.sk) olympiáda, ...).

Ako si môžeš testovať programy aj sám? Keď program číta vstup pomocou `cin >>` a vypisuje na výstup pomocou `cout <<`, používa pri tom tzv. *standardný vstup a výstup*. Za normálnych okolností je to klávesnica a obrazovka, ale je možné ich *presmerovať*. Keď v linuxovom termináli spustíš `./program <vstup >výstup`, program bude čítať vstup namiesto z klávesnice zo súboru `vstup` a vypisovať namiesto obrazovky do súboru `výstup`. Môžeš si teda napišať rôzne vstupné súbory, ktoré si nazveš napr. `1.in`, `2.in` atď. a k nim si pripravíš výstupné súbory `1.out`, `2.out` atď. so správnymi odpoveďami. Potom môžeš spustiť `./program <1.in >1.prg` a v súbore `1.prg` bude výstup programu zo vstupu `1.in`. Teraz stačí porovnať, či sú `1.prg` a `1.out` rovnaké. To sa dá aj automaticky, pomocou programu `diff`: napišeš `diff 1.prg 1.out`. Ak sú rovnaké, nevypíše sa nič, inak sa vypíše, kde sa líšia.

Ďalšia možnosť spájania vstupu a výstupu je tzv. *pipe*. Ak napišeš `./program1 | ./program2` tak sa spustí `program1` a jeho výstup sa použije ako vstup pre `program2`. Z príkazového riadku môžeš automaticky spustiť veľa testov. Príkaz

```
for f in *.in; do ./program <$f | diff - 'basename $f .in'.out; done
```

spustí `program` pre všetky súbory s príponou `.in` a pre každý porovná výstup so súborom `.out` s rovnakým menom.

Ak zistíš, že v programe je chyba, treba ju hľadať. Prostredia ako `Code::Blocks` umožňujú program spúšťať príkaz po príkaze a sledovať, ako sa pritom menia hodnoty premenných. To isté vie robiť aj program, ktorý sa volá *debugger*, napr. `gdb`. Pre naše účely ale budú celkom stačiť kontrolné výpisy: na dôležité miesta v programe pridáš vypisovanie premenných a keď spustíš program, vidíš, kde sa čo mení. Kontrolné výpisy nakoniec nezabudni vypnúť. Je dobré ich nemazať ale dať do komentárov, možno ich ešte v budúcnosti použiješ.

<sup>1</sup><https://testovac.ksp.sk>

<sup>2</sup><https://www.spoj.com>

<sup>3</sup><https://www.codechef.com>

<sup>4</sup><https://www.codeforces.com>

<sup>5</sup><https://www.ksp.sk>

<sup>6</sup>[http://oi.sk/](https://oi.sk/)

<sup>7</sup><https://ioinformatics.org/>

## 7 Čísla, znaky, reťazce

Doteraz sme hovorili, že typ `int` obsahuje celé čísla. Je to ale trošku zložitejšie. Skús si napísat takýto program:

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int i, n = 1000;
5     for (i = 0; i < 20; i++) {
6         cout << n << endl;
7         n = 10 * n;
8     }
9 }
```

Čo sa deje? Zo začiatku program vypisuje vždy desaňásobky čísla, tak ako by si čakal: `1000, 10000, 100000, ...`. Po čase sa ale namiesto násobkov desiatky začnú objavovať čudné čísla. Prečo? Súvisí to s tým, ako sú čísla v počítači uložené. Pamäť sa skladá zo súčiastok, ktoré môžu byť zapnuté alebo vypnuté, takže celá pamäť počítača sa dá predstaviť ako väčšie pole premenných, z ktorých každá môže byť 0 alebo 1. Takéto premenné sa volajú *bity*. Keď chceme zapísat iné číslo ako 0 alebo 1, musíme ho uložiť do viacerých bitov. Robíme to pomocou dvojkovej sústavy.

Keď zapisujeme čísla, spravidla to robíme v desiatkovej sústave. Pre čísla 0 až 9 máme znaky (čísllice). Keď potrebujeme zapísat desiatku, na ktorú už nemáme znak, zabalíme desať jednotiek do jedného balíka a počítame počet balíkov. Číslo 42 teda znamená *štyri balíky po desať a ešte dve*. Keď sa nám minú čísllice pre balíky, začneme robiť balíky balíkov; počítame stovky a tak ďalej. My to robíme po desať, ale to isté môžeme robiť s akýmkoľvek číslom. Keď to budeme robiť s dvojkou<sup>1</sup>, budeme namiesto balíkov po desať robiť balíky po dvoch. Teda budeme počítať 0, 1, potom 10 (t.j. *jeden balík po dvoch a nič iné*), 11, 100 (t.j. *jeden balík po štyroch a nič iné*), 101, 110, 111 (t.j. jeden balík po štyroch, jeden balík po dvoch a ešte jedna), 1000, atď. Napr. číslo 90 zapíšeme v dvojkovej sústave 1011010, pretože  $90 = 1 \cdot 64 + 0 \cdot 32 + 1 \cdot 16 + 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 0 \cdot 1$ .

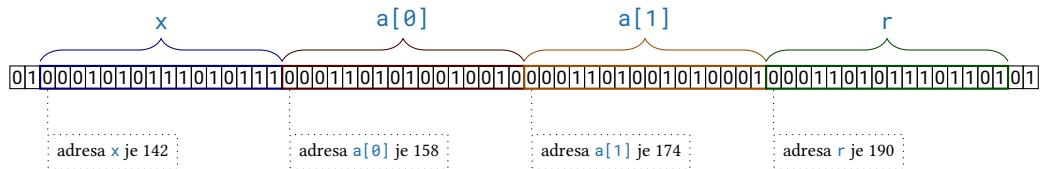
64	32	16	8	4	2	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	0	1	1	0	1	0

Názvy premenných sú len vecou programu; v skomplilovanej binárke sú inštrukcie pre procesor, ktoré hovoria, ako sa majú meniť hodnoty na jednotlivých miestach v pamäti. Miesto v pamäti sa volá *adresa*. Keď v programe napišeš `int x;` komplilátor si nájde nejaké voľné miesto v pamäti, kde bude uložená premenná `x`, zapamätá si jej adresu, a potom kedykoľvek v programe napišeš napr. `x=a`, tak komplilátor do binárky zapíše inštrukciu pre procesor, aby počas behu programu nastavil bity v pamäti na mieste, kde je `x` uložené, podľa dvojkového zápisu `a`. Problém je v tom, že rôzne veľké čísla majú rôzne dlhé zápis. Keby napr. za premennou `x` mala byť uložená premenná `y`, jej adresa by sa počas behu programu menila podľa toho, aké veľké číslo je v premennej `x` práve uložené. To by bolo veľmi nepraktické. Preto sa to rieši tak, že na každú premennú sa vyhradí fixná dĺžka. Keby si napísal

```
1 int x = 5591, a[2], r = 6893;
2 a[0] = 6802;
3 a[1] = 6737;
```

<sup>1</sup>Často sa používa aj 16-ková (tzv. hexadecimálna) sústava, kde mám číslice  $0, 1, 2, \dots, 8, 9, a, b, c, d, e, f$ , takže  $a_{16} = (10)_{10}$ ,  $b_{16} = (11)_{10}$ , až  $f_{16} = (15)_{10}$  a počítam so základom 16, napr.  $(beef)_{16} = b_{16} \cdot 16^3 + e_{16} \cdot 16^2 + e_{16} \cdot 16 + f_{16} = 11 \cdot 16^3 + 14 \cdot 16^2 + 14 \cdot 16 + 15 = (48879)_{10}$ . V C++ môžeš priamo písat číslo v 16-kovej sústave, ak pred neho napišeš `0x`. Napr. `0xffff` je `255` a `0xfeeddadbeef` je `17518595784431`.

a veľkosť `int` by bola 16 bitov, v pamäti by to mohlo vyzeráť napr. takto<sup>2</sup>



Skutočná veľkosť `int` sa lísi podľa systému, ale na väčšine súčasných počítačov je to 32 bitov. Pretože prvý bit je vyhradený na znamienko (+/-), najväčšie číslo, aké si môžeš v premennej typu `int` zapamätať, je  $1 + 2 + 2^2 + 2^3 + \dots + 2^{30} = 2147483647$ . Keby si sa pokúsil ešte pripočítať 1, prvý bit by sa nastavil na 1 a dostał by si záporné číslo.

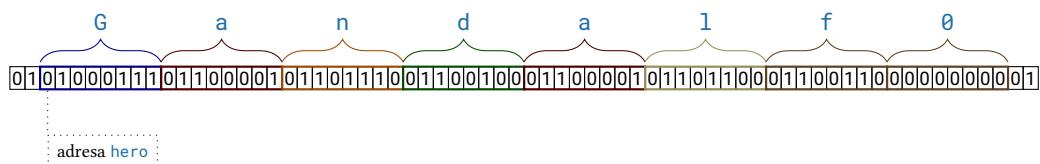
Preto existujú rôzne typy čísel: okrem `int` je `long`, ktorý je spravidla dvakrát tak dlhý, t.j. má 64 bitov, takže najväčšie číslo je  $1 + 2 + 2^2 + \dots + 2^{62} = 9223372036854775807$ . Oba typy môžeš použiť vo verzii bez znamienka (`unsigned int`, `unsigned long`) vtedy sa najväčšie číslo zdvojnásobí, ale nedajú sa pamätať záporné čísla. Vyskúšaj si napísat

```

1 #include <iostream>
2 using namespace std;
3 int main() {
4     unsigned int x = 0;
5     cout << x << endl;
6     x = x - 1;
7     cout << x << endl;
8 }
```

Ako sa dá zapamätať text? Samozrejme, ako čísla. Máme typ `char` (resp. `unsigned char`) ktorý je dlhý 8 bitov (t.j. 1 byte). Funguje rovnako ako typ `int`. Každý znak anglickej abecedy má priradenú číselnú hodnotu podľa [ASCII tabuľky](#)<sup>3</sup>. V programe sa dá použiť bud príslušné číslo, alebo znak v apostrofoch. Keď máš premennú `char x`; tak `x=107`; a `x='K'`; urobia to isté. Keďže je to číslo, dá sa s ním normálne rátať, napr. `'a'+3=='d'` je `true`.

Text sa pamäta ako pole znakov, napr. `char hero[8]`; Textu, zloženému zo znakov, sa zvykne hovoriť *reťazec* (*string*). Príkaz na vypísanie, `cout <<` za normálnych okolností neakceptuje polia, ale pre pole znakov má výnimku a vypíše ho ako text. Takisto, zadávať hodnoty `hero[0]=71`; `hero[1]=97`; ... je nepraktické, preto existuje výnimka: pri vyrábaní poľa znakov sa dá priamo inicializovať textom, napr. `char hero[8] = "Gandalf"`; vytvorí pole



Je konvencia, že posledný znak reťazca je nula, tak sa ľahko zabezpečí, že do poľa viem uložiť aj kratší text než na aký bolo rezervované. Len treba myslieť na to, rezervovať pole aspoň o 1 dlhšie ako najdlhší plánovaný reťazec. Pri priradení reťazca do poľa znakov sa dá dĺžka poľa vyniechať, ak napíšeš `char hero[] = "Gandalf"`; pole `hero` sa vytvorí s dĺžkou 8. Pre vypisovanie existuje aj skratka, že do príkazu `cout <<` sa napíše priamo reťazec v dvojitéch úvodzovkách, napr.:

<sup>2</sup>Keď sme hovorili o poliach, vravel som, že treba dávať pozor, aby si nepísal za koniec poľa. Keby si v tejto situácii napísal `a[2]=55`, prepísala by sa ti premenná `r` a ty by si netušil, kde sa tá chyba nabrať.

<sup>3</sup><http://www.csc.villanova.edu/~tway/resources/ascii-table.html>

## Čísla, znaky, reťazce

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     char hero[8] = "Gandalf";
5     cout << hero << " is " << endl;
6 }
```

Príkaz na vypisovanie vypisuje zaradom znaky a skončí, keď narazí na znak 0, takže môže vypísať aj kratšie reťazce. Čo vypíše tento program?

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     char hero[8] = "Gandalf";
5     hero[2] = 0;
6     cout << "Hero" << endl;
7 }
```

Pripradenie do poľa znakov naraz sa dá ale urobiť iba pri jeho vytváraní, takže napísať neskôr `hero = "Batman"`; sa nedá.

S načítavaním textu zo vstupu je to podobné, môžeš napísať

```
1 char mojText[20];
2 cin >> mojText;
```

Zo vstupu sa začnú čítať znaky až po prvú medzeru, tab, alebo koniec riadku<sup>4</sup> a budú sa ukladať do poľa `mojText`. Toto sa ale prudko neodporúča robiť, lebo ak je na vstupe viac znakov, ako je dĺžka poľa, začnú sa prepisovať ďalšie premenné. Vyskúšaj si písat rôzne vstupy do tohto programu:

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     char mojText[5];
5     char x = '!';
6     cin >> mojText;
7     cout << mojText << " " << x << endl;
8 }
```

Vstup:	Výstup:
pes	pes ! nič zvláštne
pes a macka	pes ! pohoda, načítava sa po prvú medzeru
macka	macka znak 0 z konca reťazca prepísal premennú x
ABCDEFG	ABCDEFG F vypisuje sa až po znak 0, premenná x je prepísaná šiestym znakom

Keď na vstupe napíšeš veľmi dlhý text, asi spôsobiš, že program skončí s chybou `Segmentation fault`: to je chyba, keď sa program snaží písat do takej časti pamäte, kde nemá prístup. Načítavanie reťazcov je teda lepšie riešiť inak, ale o tom budeme hovoriť neskôr. Nateraz nám toto stačí.

**Úloha 20.** Používateľ zadá číslo  $n$ , potom  $n$  čísel. Cieľom je vypísať "súčtovú pyramídu", napr.

<sup>4</sup>Znaky medzera, tabulátor a nový riadok sa volajú *whitespace*. Toto je dôležité: `cin >>` prestane na prvom whitespace znaku. Aj pri načítavaní čísel sa whitespace znaky preskakujú, takže je jedno, či sú vstupné čísla oddelené medzerami alebo koncami riadkov. Iná je situácia, ak sa načítava typ `char`, vtedy sa whitespace nepreskakuje.

Vstup:

```
6
1 2 3 4 5 6
```

Výstup:

```
1 2 3 4 5 6
3 5 7 9 11
8 12 16 20
20 28 36
48 64
112
```

**Úloha 21.** Na vstupe je najprv číslo  $n$ , ktoré udáva počet kôp. Potom nasleduje niekoľko príkazov. Príkaz sa skladá z kľúčového slova a prípadných parametrov. Príkazy môžu byť:

- **pridaj kam kolko** pridá kolko na kopu číslo kam
- **uber odkiaľ kolko** uberie kolko z kopy číslo kam
- **max?** zistí, aká veľká je najväčšia kopa
- **koniec** skončí načítavanie

Napište program, ktorý načíta vstup a pri každom príkaze **max?** vypíše veľkosť najväčšej kopy. Napr.

Vstup:

```
3
pridaj 1 10
uber 1 6
max?
pridaj 2 7
pridaj 2 3
max?
koniec
```

Výstup:

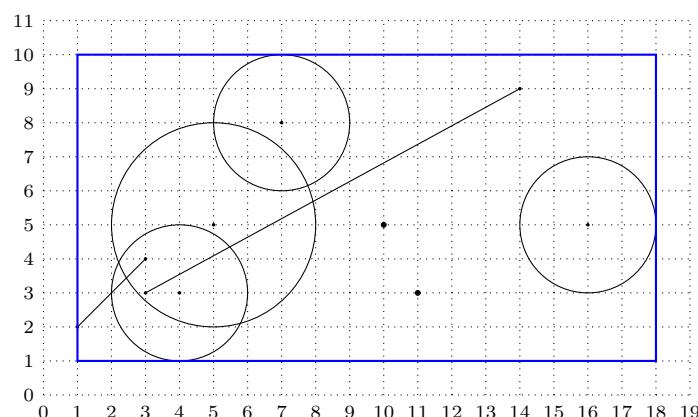
```
4
10
```

**Úloha 22.** Na stole sú rôzne geometrické útvary: body, kruhy a úsečky. Napiš program, ktorý z opisu predmetov vyráta najmenší obdĺžnik (ktorého strany sú rovnobežné s okrajmi stola), v ktorom sú všetky útvary. Na začiatku vstupu je počet útvarov  $n$ , Nasleduje  $n$  riadkov, každý z nich je jedna z možností

- **bod x y** bod so súradnicami  $[x, y]$
- **kruh x y r** kruh so stredom v bode  $[x, y]$  a polomerom  $r$
- **usecka x1 y1 x2 y2** úsečka z bodu  $[x_1, y_1]$  do bodu  $[x_2, y_2]$

Vstup:

```
8
kruh 5 5 3
usecka 1 2 3 4
kruh 7 8 2
kruh 4 3 2
bod 10 5
bod 11 3
usecka 3 3 14 9
kruh 16 5 2
```



Výstup:

```
1 1 18 10
```

## 8 Dvojrozmerné polia

Po troch vysvetľujúcich kapitolách podme naspäť k programovaniu. Chceme riešiť takúto úlohu: na vstupe sú dve čísla  $r$  a  $s$  a potom tabuľka čísel s  $r$  riadkami a  $s$  stĺpcami. Úlohou je napísanie program, ktorý tabuľku otočí takto:

Vstup:

```
3 4  
0 1 2 3  
4 5 6 7  
8 9 10 11
```

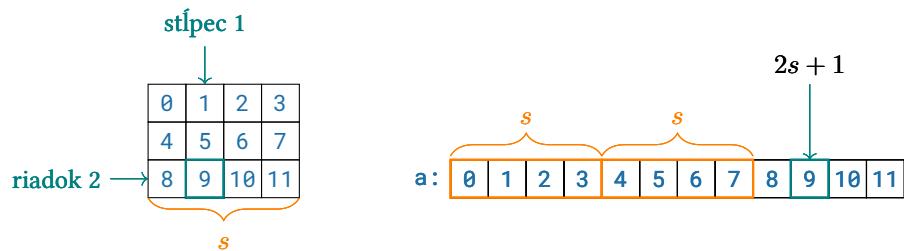
Výstup:

```
0 4 8  
1 5 9  
2 6 10  
3 7 11
```

Jeden prístup je prečítať celý vstup do jedného poľa veľkosti  $rs$ :

```
1 int r, s, i;  
2 cin >> r >> s;  
3 int a[r * s];  
4 for (i = 0; i < r * s; i++) cin >> a[i];
```

Fajn, ale ako ho vypísať? Keď si očísľujeme riadky aj stĺpce od 0, tak vidíme, že číslo v riadku  $i$  a stĺpci  $j$  bude na pozícii  $is+j$ : pred ním je  $i$  celých riadkov po  $s$  čísel a ešte  $j$  čísel z riadku  $i$ . Napr. v riadku 2 a stĺpici 1 je číslo 9:



Vypisovanie vieme urobiť tak, že v jednom cykle (v premennej  $j$ ) prechádzame po stĺpcoch a pre každý stĺpec ideme v druhom cykle po všetkých číslach tohto stĺpca a vypíšeme ich:

```
1 for (j = 0; j < s; j++) {  
2     for (i = 0; i < r; i++)  
3         cout << a[i * s + j] << " ";  
4     cout << endl;  
5 }
```

Aby sa nám takéto výpočty zjednodušili, môžeme použiť dvojrozmerné pole, ktoré vytvorí tabuľku premenných. Príkaz `int a[3][4];` vyrobí premenné

```
a[0][0]    a[0][1]    a[0][2]    a[0][3]  
a[1][0]    a[1][1]    a[1][2]    a[1][3]  
a[2][0]    a[2][1]    a[2][2]    a[2][3]
```

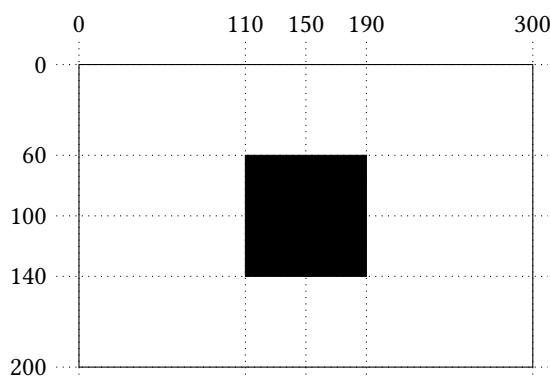
Celý program bude vyzerať takto:

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int r, s, i, j;
5     cin >> r >> s;
6     int a[r][s];
7
8     for (i = 0; i < r; i++)    // cyklus pre riadky
9         for (j = 0; j < s; j++) // cyklus pre stĺpce
10            cin >> a[i][j];
11
12    for (j = 0; j < s; j++) { // vypíšeme vymenené riadky a stĺpce
13        for (i = 0; i < r; i++)
14            cout << a[i][j] << " ";
15        cout << endl;
16    }
17 }
```

**Úloha 23.** Na vstupe sú čísla  $r$ ,  $s$ ,  $d$  a tabuľka čísel s  $r$  riadkami a  $s$  stĺpcami. Napíšte program, ktorý zistí, či sa v tabuľke vyskytuje štvorec  $d \times d$  zo samých núl.

## 9 Čiernobiele obrázky

Dvojrozmerné pole núl a jednotiek sa dá chápať ako čiernobiely obrázok, kde 1 je biela a 0 čierna. Aby sa dali lepšie pozerať, môžeš si stiahnuť súbor [obrazok.h](#)<sup>1</sup>, v ktorom som ti pripravil<sup>2</sup> príkaz na zapísanie dvojrozmerného poľa do obrázku. Ukažem ti, ako sa s ním pracuje. Povedzme, že chceme urobiť obrázok rozmerov  $300 \times 200$ , ktorý bude mať v strede čierny štvorec rozmerov  $80 \times 80$  takto:



Začni tým, že si súbor [obrazok.h](#) ulož do toho istého adresára, kde máš program a napiš:

```
1 #include <iostream>
2 #include "obrazok.h"
3
4 int main() {
5     int a[200][300], i, j;
6     for (i = 0; i < 200; i++)
7         for (j = 0; j < 300; j++)
8             if (i >= 60 && i <= 140 && j >= 110 && j <= 190)
9                 a[i][j] = 0;
10            else
11                a[i][j] = 1;
12    zapis_cb_png(200, 300, a, "vystup.png");
13 }
```

V druhom riadku hovoríš, že sa má použiť súbor [obrazok.h](#). V poslednom riadku používaš príkaz [zapis\\_cb\\_png](#), ktorý je naprogramovaný v tom súbore a má tri parametre, ktoré sa pišu v zátvorkách: počet riadkov, počet stĺpcov, dvojrozmerné pole núl a jednotiek a meno výstupného súboru. Zvyšok by mal byť jasný: premennou i prechádzame v cykle cez všetky riadky, pre každý riadok premennou j prechádzame cez všetky stlpce. Pre každé poličko poľa zistíme, aká farba tam má byť, a zapíšeme. Keď program skompliluješ a spustíš, vyrobí ti súbor [vystup.png](#), ktorý sa dá pozrieť v prehliadači obrázkov.

**Úloha 24.** Napiš program, ktorý načíta čísla  $n$  a  $d$  a vytvorí štvorcový obrázok rozmerov  $n \times n$  s čiernym rámkom hrúbky  $d$ .

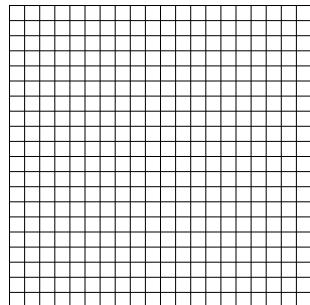
**Úloha 25.** Napiš program, ktorý precítia číslo  $d$  a spraví obrázok rozmerov  $d \times d$  s čiernym trojuholníkom takto:



<sup>1</sup><https://github.com/pocestny/programovanie/raw/master/materialy/obrazok.h>

<sup>2</sup>V skutočnosti som len použil program [LodePNG](#), ktorý je voľne prístupný a vie zapisovať obrázky vo formáte PNG.

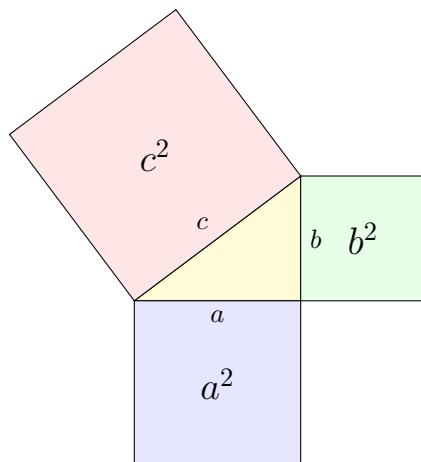
**Úloha 26.** Napíš program, ktorý prečíta čísla  $n$ ,  $d$  a vytvorí obrázok so štvorcovou mriežkou  $n \times n$  a rozostupmi  $d$ . Napr. pre  $n = 301$  a  $d = 15$  bude obrázok:



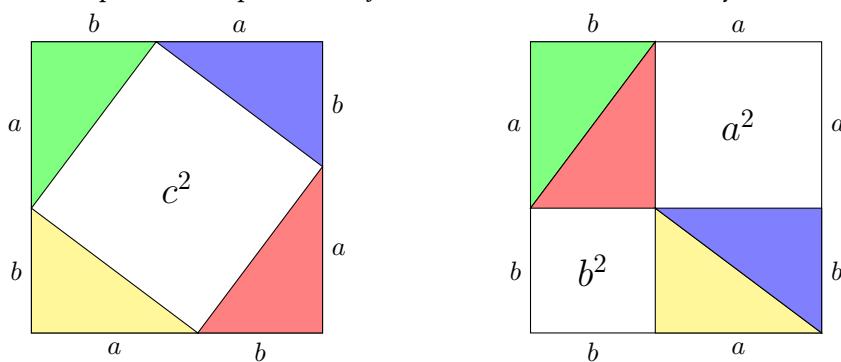
**Úloha 27.** Napíš program, ktorý prečíta číslo  $d$  a vytvorí obrázok šachovnice ( $8 \times 8$  políčok, v ľavom dolnom rohu je čierne políčko) rozmerov  $8d \times 8d$ , ktorej jedno políčko má rozmery  $d \times d$ .

### Malá odbočka: Pytagorova veta, Euklidova vzdialenosť a kruhy

Poznáš Pytagorovu vetu? Nie je nová, starí Gréci ju poznali viac ako 300 rokov pr. Kr. Asi vieš, že obsah štvorca so stranou  $x$  sa dá vypočítať ako  $x \cdot x = x^2$ . Pytagorova veta hovorí, že v pravouhlom trojuholníku sa obsah štvorca nad preponou rovná súčtu obsahov štvorcov nad odvesnami. Alebo kratšie,  $c^2 = a^2 + b^2$ :

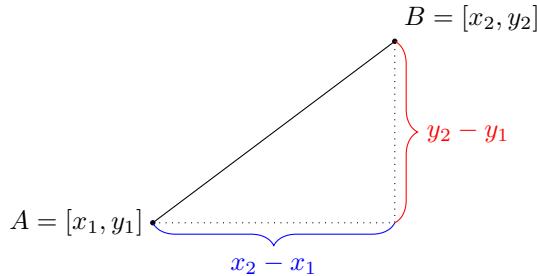


Prečo to platí ľahko vidno z nasledujúceho obrázka. Vľavo sú štyri rovnaké trojuholníky naukladané do štvorca. Pri každom bode v strede strany sú dva rôzne uhly z trojuholníkov, ktoré dokopy dajú  $90^\circ$  (súčet uhlrov v trojuholníku je  $180^\circ$ ). Preto biela vec vovnútri je štvorec. Vo veľkom štvorci sa iba premiestnili farebné trojuholníky a vznikol obrázok vpravo, takže plocha bielej časti v oboch obrázkoch musí byť rovnaká:



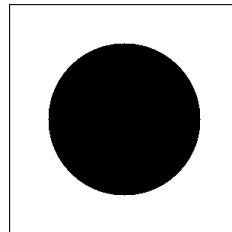
## Čiernobiele obrázky

Pythagorova veta zároveň hovorí<sup>3</sup>, že ak mám v rovine dva body, bod  $A$  so súradnicami  $[x_1, y_1]$  a bod  $B$  so súradnicami  $[x_2, y_2]$ , tak ich vzdialenosť je  $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ :



No a napokon kruh so stredom v bode  $S = [x, y]$  a polomerom  $r$  je tvorený bodmi  $X = [x', y']$ , pre ktoré vzdialenosť od  $S$  je nanajvýš  $r$ , takže platí  $(x - x')^2 + (y - y')^2 \leq r^2$ . S týmto už ľahko vyriešiš túto úlohu:

**Úloha 28.** Na vstupe sú čísla  $a$  a  $r$ . Napiš program, ktorý vytvorí obrázok rozmerov  $a \times a$ , ktorý bude mať v strede kruh s polomerom  $r$ . Napr. pre  $a = 300$ ,  $r = 100$  bude obrázok vyzerať takto:



## Projekt: Celulárny automat

Máme kolóniu baktérií, ktoré žijú na čiare. Na začiatku je živá iba jedna baktéria:

.....\*.....

V každej ďalšej generácii sa kolónia vydáva podľa nasledovných pravidiel:

1. ak je na poličke baktéria, ktorá má dvoch susedov, tak skape, lebo nemá dosť miesta
2. ak je prázdne poličko, ktoré nesusedí s baktériou, ostane prázdne
3. vo všetkých ostatných prípadoch na poličku bude baktéria

Prvé štyri generácie budú vyzerať takto:

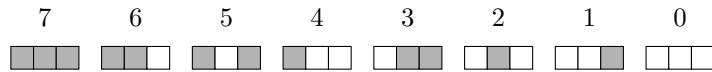
0: .....\*.....  
1: .....\*\*\*.....  
2: .....\*\*.\*\*.....  
3: .....\*\*\*\*\*.....  
4: .....\*\*.....\*\*.....

**Úloha 29.** Napiš program, ktorý načíta  $n$  a  $t$ , pričom  $t < n$  a vypíše jeden riadok dĺžky  $2n + 1$  z bodiek a hviezdičiek, ktorý bude znázorňovať, ako bude kolónia vyzerať po  $t$  generáciach.

**Úloha 30.** Napiš program, ktorý precíta  $t$  a vykreslí obrázok rozmerov  $(2t + 1) \times t$ , v ktorom  $i$ -ty riadok bude zodpovedať  $i$ -tej generácii: ak je na poličke baktéria, bude tam čierny pixel, inak biely. Prvý riadok bude celý biely, iba v strede (na pozícii  $[0][t]$ ) bude čierny pixel.

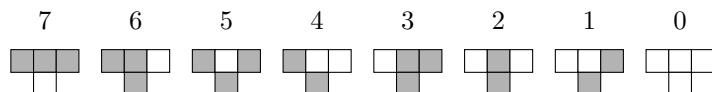
<sup>3</sup>kedže  $c^2 = a^2 + b^2$ , tak  $c = \sqrt{a^2 + b^2}$ . Znak  $\sqrt{x}$  sa číta *odmocnina z x* a znamená číslo, ktoré keď umocním na druhú (t.j. vynásobím samé so sebou) dostanem  $x$ , t.j.  $(\sqrt{x})^2 = \sqrt{x} \cdot \sqrt{x} = x$ . Napríklad  $\sqrt{36} = 6$ .

Môžeme ďalej skúmať, ako by vyzeral náš obrázok, keby kolónia mala iné pravidlá. Políčko má osem možností, ako môže vyzeráť jeho okolie:



Ak si všimneš, očísloval som ich tak, že ak plné políčko je jednotka a prázdne políčko je nula, tak políčka vlastne tvoria číslo okolia zapísané v dvojkovej sústave, napr. okolie s číslom 5 je **101**. Inak povedané, ak mám tri čísla (nuly a jednotky)  $x, y, z$ , tak číslo okolia je  $4x + 2y + z$ .

Pravidlo pre rast kolónie musí pre každé okolie povedať, či v ňom v ďalšej generácii bude baktéria alebo nie. Naše pravidlo vyzerá takto:



Každé pravidlo viem opísť pomocou 8 nul alebo jednotiek, ktoré hovoria výsledok pre jednotlivé okolia. Naše pravidlo je **01111110**. Keď si predstavíme, že to je zápis nejakého čísla v dvojkovej sústave, každé z 256 možných pravidiel kolónie môžeme označiť číslom z rozsahu 0 až 255. Naše pravidlo je

128	64	32	16	8	4	2	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0	1	1	1	1	1	1	0

$64 + 32 + 16 + 8 + 4 + 2 = 126$ . Našim cieľom teraz bude preskúmať rôzne pravidlá. Ako ľahko pracovať s číslom pravidla? Na to pomôžu tzv. *bitové operácie*. Sú to operácie nad celými číslami (podobne ako **+, -, \*, /, %**), ale pracujú s jednotlivými bitmi čísla v dvojkovej sústave.

<b>&amp;</b>	AND	výsledok je číslo, ktoré má bit nastavený na 1, ak obe čísla majú príslušný bit 1	<b>10 &amp; 9 == 8</b>	<b>8 4 2 1</b> 10:1 0 1 0 9: 1 0 0 1 ----- 8: 1 0 0 0
<b> </b>	OR	výsledok je číslo, ktoré má bit nastavený na 1, ak aspoň jedno číslo má príslušný bit 1	<b>10   9 == 11</b>	<b>8 4 2 1</b> 10:1 0 1 0 9: 1 0 0 1 ----- 11: 1 0 1 1
<b>^</b>	XOR	výsledok je číslo, ktoré má bit nastavený na 1, ak práve jedno číslo má príslušný bit 1	<b>10 ^ 9 == 3</b>	<b>8 4 2 1</b> 10:1 0 1 0 9: 1 0 0 1 ----- 3: 0 0 1 1

Ešte sa hodia dve bitové operácie. Operácia **<<** posúva binárny zápis čísla dolava.  $x << a$  pripíše  $a$  núl na koniec dvojkového<sup>4</sup> zápisu  $x$ . To je to isté, ako keby som ho  $a$ -krát za sebou vynásobil dvomi. Napr. 5 je v dvojkovej sústave **101**.  $5 << 3$  preto bude číslo s dvojkovým zápisom **101000**, čo je  $5 \cdot 8 = 40$ . Opačná operácia **>>** posúva číslo doprava, t.j. zoberie celú časť po delení dvomi. Napr. 45 je v dvojkovej sústave **101101**. Keď ho trikrát vydelím dvomi a vždy zoberiem celú časť, dostanem **10110** (čo je 22), **1011** (čo je 11) a **101** (čo je 5). Preto  $45 >> 3$  je 5.

S týmito operáciami viem jednoducho vyhodnocovať pravidlá. Predpokladajme, že mám pravidlo  $r$  a niekde v poli mám hodnoty políčok  $a[i-1], a[i], a[i+1]$ . Číslo okolia (od 0 do 7) dostanem tak, že vyrátam  $z = 4 * a[i-1] + 2 * a[i] + a[i+1]$ . Bit na pozícii  $z$  v pravidle mi hovorí, či bude políčko v ďalšej generácii obsadené. Preto stačí vyrátať  $r \& (1 << z)$ , t.j. jednotku posunúť o  $z$  pozícii doprava a urobiť operáciu AND s pravidlom. Ak je výsledok 0, políčko má byť prázdne. Ak je výsledok nenulový (t.j. rovný  $1 << z$ ), na políčku bude baktéria.

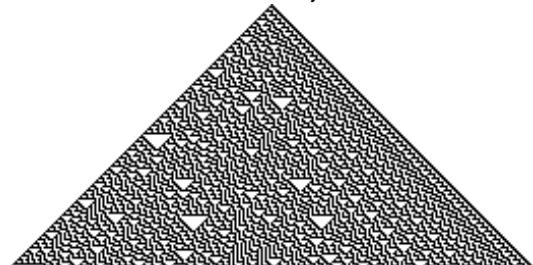
<sup>4</sup>Tu sa prejaví výhoda 16-kovej sústavy. Pretože  $16 = 2^4$ , jedna cifra v 16-kovej sústave priamo zodpovedá štyrom binárnym cifrám. Takže napr. **0xff»8** je **255»8**, t.j.  $255 \cdot 2^8 = 255 \cdot 256 = 65280$ , čo je **0xffff00**.

## Čiernobiele obrázky

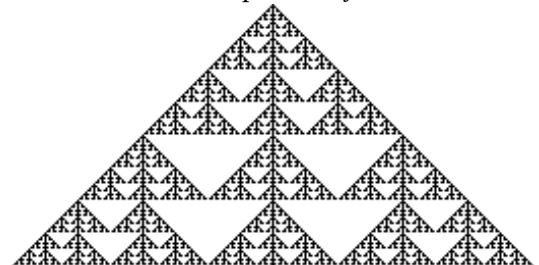
---

**Úloha 31.** Napiš program, ktorý načíta číslo pravidla  $r$  a počet iterácií  $t$  a vytvorí obrázok (rozmerov  $2t + 1 \times t$ ), ako vyzerá vývoj kolónie s pravidlom  $r$  počas  $t$  generácií.

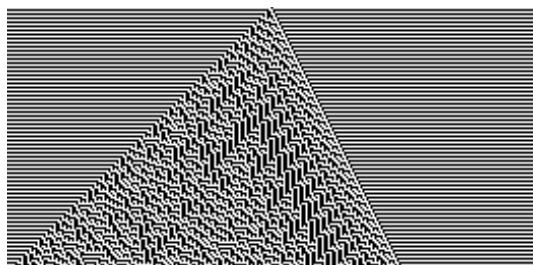
Môžeš skúmať rôzne pravidlá: niektoré kolónie rýchlo vymrú, niektoré vytvoria jednoduché útvary (napr. čiaru), iné robia tzv. deterministický chaos, a ešte iné rôzne pravidelné obrazce. Niekoľko pravidiel je tu<sup>5</sup>:



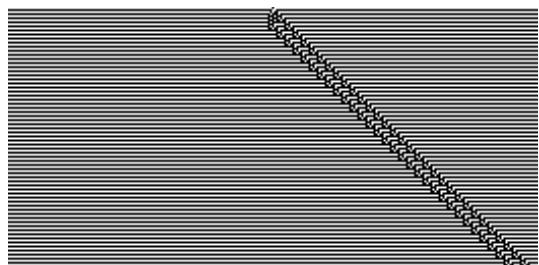
pravidlo 86



pravidlo 150



pravidlo 89



pravidlo 107

Rovnako dobre sa dá skúsať, čo sa stane, ak kolónia nezačína z jednej baktérie, ale z niekoľkých, ako sa správajú dve rastúce kolónie, ktoré do seba narazia, čo ak je pravidlo z väčšieho okolia a pod.

---

<sup>5</sup>Všimni si, že párne pravidlo vytvorí biele pozadie a nepárne pravidlo pásikavé. Je to náhoda? Vieš povedať, prečo to tak je?

Kapitolu 9 som začal tým, že som ti pripravil súbor, v ktorom bol nový príkaz `zapis_cb_png`. Teraz ti idem ukázať, ako si nové príkazy, ktorým sa hovorí *funkcie*, môžeš vyrobiť aj ty. Možnosť vyrobiť si vlastné funkcie je veľmi dôležitá: umožňuje ti rozdeliť si veľký program na malé časti, ktoré potom môžeš používať viackrát.

Najjednoduchšia funkcia vyzerá takto:

```

1 #include <iostream>
2 using namespace std;
3
4 int sedem() {
5     return 7;
6 }
7
8 int main() {
9     cout << sedem() << endl;
10 }
```

Všimni si, že sa veľmi podobá na zápis, ktorý používame na program. Nie je to náhoda. Celý program je v skutočnosti len kopa funkcií, ibaže jedna z nich je špeciálna: tá, ktorá sa volá `main` sa začne vykonávať ako prvá. Každá funkcia je samostatný program, ktorý má svoj vlastný svet.

Funkcia má meno<sup>1</sup> a typ. V našom programe sme spravili funkciu, ktorá sa volá `sedem`. Typ `int` pred jej menom hovorí, že táto funkcia predstavuje výraz, ktorého výsledok je celé číslo. Guľaté zátvorky za menom slúžia na parametre. Zatiaľ žiadne nemáme, ale zátvorky aj tak treba písť, aby bolo jasné, že je to funkcia a nie premenná. Potom nasleduje zložený príkaz, t.j. v kučeravých zátvorkách napísané príkazy.

Funkciu vieme v nejakej inej funkcií použiť (*zavolať*) tak, že napišeme jej meno a parametre, v našom prípade `sedem()`. Pretože naša funkcia má typ `int`, vieme, že každé jej zavolanie vráti celé číslo a môžme ju použiť vo výrazoch rovnako, ako by sme používali číslo. Môžeme napr. urobiť `int x = 3 + sedem();` a pod. Keďkoľvek treba vyhodnotiť výraz, v ktorom sa volá funkcia, všetko sa preruší a spustí sa tá funkcia. Príkaz `cout << sedem() << endl;` v našom programe hovorí *Začni vypisovať, mal by si vypísať číslo. Preruš robotu, spusti funkciu sedem() a počkaj, kým skončí. Keď skončí, ako výsledok ti dá číslo. Pokračuj vo vypisovaní, vypíš toto číslo a potom znak konca riadku.*

Príkazy, ktoré sa vo funkcií vykonávajú sa volajú *telo funkcie*. Špeciálny príkaz `return` vyraz; urobí to, že skončí funkciu a výsledok funkcie (tzv. *návratová hodnota*) bude hodnota výrazu vyraz. Typ výrazu musí byť rovnaký ako návratový typ funkcie, ktorý sme zadali pri jej definícii. Napr. naša funkcia `sedem()` vracia typ `int`, preto výraz v príkaze `return` musí mať hodnotu typu `int`. Funkcie ale môžu mať ako návratovú hodnotu ľubovoľný typ. Ak by som mal funkciu `bool zle() { return false; }`, môžem ju použiť všade, kde môžem mať výraz typu `bool`, napr. `if (zle()) {}`.

Každá funkcia musí skončiť príkazom `return`<sup>2</sup>. Funkcia je samostatný svet, takže môže mať vlastné premenné. Tie sú úplne nezávislé od iných funkcií (hovoríme im *lokálne premenné*). Čo urobí tento program?

```

1 #include <iostream>
2 using namespace std;
3
4 int pridaj() {
5     int x;
6     cin >> x;
```

<sup>1</sup>Podobne ako pri premenných, meno funkcie môže byť čokoľvek z veľkých a malých písmen, podčiarkovníkov a čísel, ale nesmie začínať číslom.

<sup>2</sup>Slušné by bolo, aby sme aj na konci hlavného programu písali `return 0;`, ale keďže už potom program skončí, tak to nevadí, keď to vynecháme.

## Funkcie

```
7     return x + 7;
8 }
9
10 int main() {
11     int x = pridaj();
12     cout << x + pridaj() << endl;
13 }
```

Hlavný program vytvorí premennú `x`. Zavolá funkciu `pridaj()` a jej výsledok má uložiť do `x`. Keď sa zavolá funkcia `pridaj()`, vznikne nový svet, v ktorom sa vytvorí iná premenná `x` a načíta sa do nej číslo zo vstupu. Dajme tomu, že používateľ napíše 2. Potom sa funkcia `pridaj()` skončí, jej svet zanikne, a ostane len výsledok, číslo 9. Pracovať pokračuje hlavný program, do svojej premennej `x` uloží výsledok volania funkcie, t.j. 9. Pokračuje príkazom `cout <<`, na to potrebuje vyhodnotiť výraz `x + pridaj()`. V `x` má 9, opäť preruší robotu a zavolá `pridaj()`. Zase vznikne nový svet s premennou `x`, do ktorej sa prečíta hodnota. Dajme tomu, že teraz je na vstupe 3. Funkcia skončí, jej svet zanikne a ostane číslo 10. Hlavný program teraz dokončí rátanie výrazu `9 + 10` a vypíše 19.

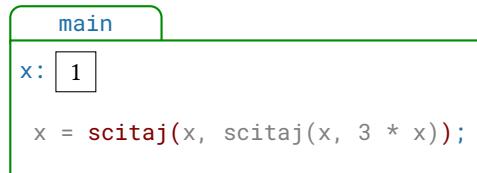
Doteraz sme nehovorili o parametroch. Parametre umožňujú používať tú istú funkciu na rátanie rôznych vecí. Sú to premenné, ktoré sú lokálne premenné funkcie, ale ich hodnota sa dá nastaviť zvonka pri volaní funkcie. Parametre a ich typy sú uvedené v zátvorkách za menom funkcie, takže napr.

```
1 int scitaj(int x, int y) {
2     int z = x + y;
3     return z;
4 }
```

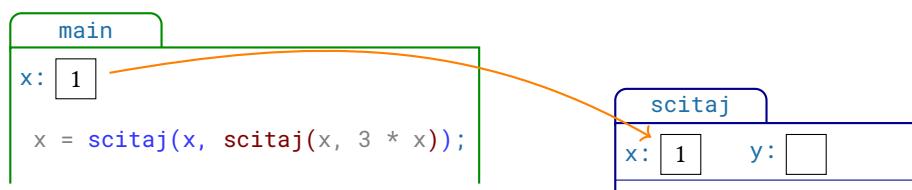
je funkcia, ktorá má dva parametre a oba sú celé čísla. Vo vnútri funkcie sa k nim dá správať ako k normálnym premenným. Majme hlavný program:

```
1 int main() {
2     int x = 1;
3     x = scitaj(x, scitaj(x, 3 * x));
4     cout << x << endl;
5 }
```

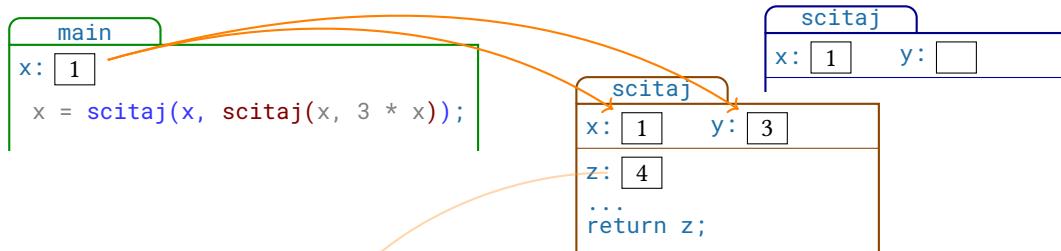
Skús sa zamyslieť, čo vypíše. Máš to? Podme to pozrieť detailne. Hlavný program sa spustí, vytvorí si svoju premennú `x` a dá do nej 1. Potom ide vyhodnocovať výraz, ktorého výsledok má uložiť do `x` a zistí, že je to volanie funkcie.



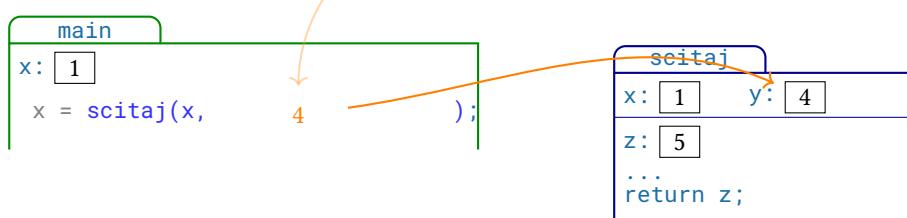
Vyrobí sa teda nový svet funkcie `scitaj` a hlavný program chce nastaviť hodnoty parametrov. Prvý parameter nastaví podľa aktuálneho obsahu `x`, ale pri druhom zistí, že je to výraz, v ktorom je opäť volanie funkcie.



Vytvorí sa teda ďalší, nezávislý, svet funkcie `scitaj`, nastavia sa mu hodnoty parametrov a vyráta sa výsledok, ktorý sa potom vráti tomu, kto funkciu zavola.



Teraz druhý svet zanikne, ostane z neho iba výsledok 4, a tým sa nastaví parameter v prvom svete:



Napokon zanikne aj prvý svet, ostane hodnota 5 a tá sa zapíše do premennej `x` v hlavnom programe. Takže si to zhrnme: funkcia je samostatný program, ktorý má výstupnú hodnotu<sup>3</sup>. Dá sa použiť vo výrazoch. Pri volaní funkcie sa vytvorí nový nezávislý svet, nastavia sa hodnoty parametrov<sup>4</sup> a vypočíta sa výstupná hodnota.

**Úloha 32.** Napíš funkciu `max` s troma parametrami, ktorá vráti najväčší z nich.

**Úloha 33.** Napíš funkciu `int dlzka(int n) {...}`, ktorá pre zadané číslo `n` vráti počet cifier (napr. `dlzka(546)` je 3). Pomocou nej napíš program, ktorý prečíta zo vstupu číslo `n` a vypíše zarovnanú tabuľku s mocninami čísel 2 až `n`. Napr. pre `n = 7` by výstup vyzeral

1	2	4	8	16	32	64
1	3	9	27	81	243	729
1	4	16	64	256	1024	4096
1	5	25	125	625	3125	15625
1	6	36	216	1296	7776	46656
1	7	49	343	2401	16807	117649

Doposiaľ sme mali každú premennú vyrobenu v nejakej funkcií (či už `main` alebo v nejakej inej). Dajú sa však vyrobiť aj premenné, ktoré žiadnej funkcií nepatria a sú viditeľné z každého sveta: hovorí sa im *globálne premenné*. Najľahšie sa to ukáže na píklaide:

```

1 #include <iostream>
2 using namespace std;
3
4 int a;
5
6 void pridaj(int kolko) { a = a + kolko; }
7
8 int main() {

```

<sup>3</sup>Tu len pripomienim: úplne na začiatku sme hovorili, že najjednoduchší príkaz `3*(2+5);` len vyráta hodnotu 21 a výsledok zahodí. To je, samozrejme, zbytočné, ale niekedy môžeš chcieť zahodiť hodnotu funkcie. Napríklad ak zoberiem funkciu `pridaj` z predchádzajúceho príkladu, tak volanie `pridaj();` načíta číslo zo vstupu a ignoruje ho.

<sup>4</sup>Zatiaľ vieme ako parametre funkciám posielat jednoduché premenné, ale nie polia. Polia sa tiež dajú ako parametre použiť, ale o tom až neskôr.

## Funkcie

```
9  int i;
10 a = 0;
11 for (i = 0; i < 10; i++) pridaj(i);
12 cout << a << endl;
13 }
```

Tu sme vyrobili globálnu premennú `a`. Keďže je vyrobená skôr, ako definujeme funkciu `pridaj`, telo funkcie môže k premennej `a` pristupovať<sup>5</sup>. Tu som použil ako návratový typ nový typ: `void`. Je to prázdny typ, ktorým hovoríme, že funkciu chceme vždy použiť iba v situácii, keď výslednú hodnotu zahodíme. Takéto funkcie nemusia končiť príkazom `return`. Po spustení programu vypíše  $0 + 1 + 2 + \dots + 9 = 45$ .

Poznámka o viditeľnosti: rovnako ako pri funkciách sa samostatný svet vyrába aj pri zložených príkazoch. To znamená, že svety môžu zapadať jeden do druhého: svet globálnych premenných, v ňom svet funkcie, v ňom svet zloženého príkazu,... Vždy, keď program potrebuje použiť nejakú premennú,, vyberie sa premenná s požadovaným menom z najbližšieho sveta, v ktorom taká je. Napr. program

```
1 #include <iostream>
2 using namespace std;
3
4 int x = 5;
5
6 void p() { cout << x; }
7
8 int main() {
9     int x = 7;
10    int a = 9;
11    cout << x;
12    p();
13    {
14        int x = 6;
15        cout << x << a;
16        p();
17    }
18    cout << x << endl;
19 }
```

vypíše<sup>6</sup> 756957: funkcia `main` má svoju vlastnú `x`, preto sa použije tá. Funkcia `p` vo svojom svete nemá premennú `x`, preto sa použije globálna `x`. Zložený príkaz má svoj vlastný svet, v ktorom má svoju vlastnú `x`, ale `a` sa použije zo sveta `main`.

Globálne premenné môžu byť rovnako dobre aj polia.  
Tu je ale problém, akú veľkosť má pole mať. Pretože ho vyrábam skôr, ako sa spustí hlavný program, nemôžem čakať, kým zo vstupu dostanem veľkosť pola. Najedenoduchší spôsob je povedať si dopredu ohraničenie na veľkosť pola, vytobiť dostatočne veľké pole a potom z neho použiť len potrebný kus. Toto nie je ideálny prístup (okrem iného mrhá pamäťou), ale nateraz nám postačí. Napr. program vpravo:

```
1 #include <iostream>
2 using namespace std;
3
4 int a[1000];
5
6 int main() {
7     int n = -1;
8     do {
9         n++;
10        cin >> a[n];
11    } while (a[n] > 0);
12 }
```

<sup>5</sup>Keby sme vymenili riadky 4 a 6, globálna premenná `a` by vo funkciu `pridaj` nebola viditeľná; premenná musí byť vždy vyrobená v programe skôr, ako sa použije.

<sup>6</sup>Všimni si, že C++ má tzv. *static scope*: ktorý je najbližší svet sa rozhoduje podľa toho, ako je to v programe zapísané. Takže vo volaní `p()` z riadku 16 sa použije globálna premenná `x` a nie premenná `x` zo sveta zloženého príkazu na riadku 14.

číta do poľa `a` vstupné kladné čísla a skončí, keď sa prvýkrát prečíta niečo  $\leq 0$ . V premennej `n` (ktorá je lokálna funkcií `main`) je uložená "skutočná" dĺžka poľa. Ak by bolo na vstupe viac ako 1000 čísel, stanú sa hrozné veci.

**Úloha 34.** Napíš funkciu s parametrom `int n`, ktorá v globálnom poli `a` (o ktorom predpokladáme, že obsahuje kladné čísla) nájde najväčší prvek spomedzi prvých `n`, prepíše ho v poli na `-1` a vráti jeho hodnotu (ešte predtým, ako ho prepísala). Potom s pomocou nej napíš program, ktorý do poľa prečíta kladné čísla zo vstupu a vypíše ich v utriedenom poradí.

Na ľahšiu prácu s globálnymi premennými je v súbore `obrazok.h` vyrobená funkcia `zapis_cb_png_vyrez`, ktorá má dodatočné štyri parametre. Tie hovoria, ktoré riadky a ktoré stĺpce sa majú do obrázka zapísť. Program

```

1 #include <iostream>
2 #include "obrazok.h"
3 using namespace std;
4
5 const int N = 2000;
6 int a[N][N];
7
8 int main() {
9     // tuto niečo nakresli
10    zapis_cb_png_vyrez(N, N, a, "vystup.png", 50, 80, 300, 200);
11 }
```

vypíše do súboru `vystup.png` výrez z poľa `a` tvorený riadkami 50 až 299 a stĺpcami 80 až 199. Pretože `a` je teraz vyrobené ako globálna premenná, jeho rozmery musia byť známe už v čase komplikácie. Na to je špeciálny modifikátor `const`, ktorý sme použili pri vytváraní premennej `N` a ktorý hovorí, že `N` sa v programe nebude meniť (môžeš si vyskúšať, že to nejde).

**Úloha 35.** Napíš funkciu `void stvorec(int r, int s, int d)`, ktorá v globálnom poli vyplní jednotkami štvorec  $d \times d$  začínajúci na riadku  $r$  a stĺpco  $s$ . S jej pomocou napíš program, ktorý načíta tri čísla  $a, b, c$  také, že  $a + b + c < 100$  a vyrobí obrázok rozmerov  $100 \times 100$ , v ktorom budú tri štvorce rozmerov  $a \times a$ ,  $b \times b$  a  $c \times c$  položené jeden na druhom v strede obrázka. Napr. pre vstup  $40, 30, 20$  bude obrázok vyzerať takto:



## 11 Rekurzia

Volanie funkcie pri vyhodnocovaní výrazu, ako sme ho opísali v minulej kapitole, nemusí byť len z hlavného programu, ale z hocijakej funkcie, tento program vypíše 16:

```
1 #include <iostream>
2 using namespace std;
3
4 int a(int x) { return x + 1; }
5
6 int b(int x) { return a(x) * a(x); }
7
8 int main() { cout << b(3) << endl; }
```

Jediná vec, ktorú treba dodržať je, podobne ako pri premenných, že každé použitie (volanie) musí byť až vtedy, keď funkcia bola v programe vyrobená. Keby si napr. v predchádzajúcom programe prehodil riadky s definíciou funkcií **a** a **b**, nefungoval by.

Špeciálne zaujímavá je situácia, keď funkcia vo svojom tele volá sama seba. Takéto volanie je možné a hovorí sa mu *rekurzia*. Na prvý pohľad to možno vyzerá divne, ale nie je to. Treba len, podobne ako pri cykle, zabezpečiť, aby sa volanie nezacyklilo ako táto funkcia:

```
1 int a(int x) { return 1 + a(x + 1); }
```

Čo sa začne diať, ak zavoláš **a(0)**? Najprv sa spraví nový svet funkcie **a**, v ktorom sa **x** nastaví na 0 a začne sa vykonávať telo funkcie. V ňom sa spraví nový nezávislý svet funkcie **a**, v ktorom sa **x** nastaví na 1 a začne sa vykonávať telo funkcie. V ňom sa spraví nový nezávislý svet funkcie **a**, v ktorom sa **x** nastaví na 2 a začne sa vykonávať telo funkcie. V ňom sa spraví nový nezávislý svet funkcie **a**, v ktorom sa ...

Keď ale funkciu naprogramuješ správne "dno", niektoré rekurzívne zápisy sú veľmi prirodzené. Zober si napríklad hľadanie najväčšieho spoločného deliteľa. Euklides vedel, že ak máme dve čísla  $a$  a  $b$  také, že  $a > b$ , a obidve sú deliteľné číslom  $k$ , to znamená že  $a = xk$  a  $b = yk$  pre nejaké  $x$  a  $y$ . Potom  $a - b = xk - yk = (x - y)k$ , a teda aj  $x - y$  je deliteľné číslom  $k$ . Preto každý spoločný deliteľ  $a$  a  $b$  je zároveň spoločným deliteľom  $a - b$  a  $b$ . Najväčšieho spoločného deliteľa preto vieme nájsť takto:

```
1 int NSD(int a, int b) {
2     if (a == b) return a;
3     if (a == 1 || b == 1) return 1;
4     if (a > b) return NSD(a - b, b);
5     return NSD(b - a, a);
6 }
```

Keď zavoláme napr **NSD(4, 6)**, toto volanie následne zavolá **NSD(2, 4)**, toto zase **NSD(2, 2)** a toto volanie už ďalší svet **NSD** nevytvorí, ale vráti 2. Potom už reťazovo všetky ostatné volania dobehnú a vrátia 2. Ako sa presvedčíme, že sa táto funkcia nikdy nezacyklí? Všimni si, že keď funkciu zavoláme s hocijakými  $a > 0$ ,  $b > 0$ , každé ďalšie volanie zmenší súčet  $a + b$ . Zároveň ani  $a$ , ani  $b$  neklesne na nulu (to by museli byť v predchádzajúcom volaní rovnaké, ale vtedy by sa už ďalšie volanie nevyrobilo). Keďže súčet nemôže klesať donekonečna, vidíme, že funkcia sa nikdy nezacyklí.

**Úloha 36.** Napíš program, ktorý načíta zo vstupu jedno prirodzené číslo a vypíše ho otočené<sup>1</sup> Má to ale háčik: v celom programe môžeš použiť iba jedinú premennú (tú, do ktorej načítaš vstup).

<sup>1</sup>To znamená, že najprv sa vypíše posledná cifra, t.j. zvyšok po delení desiatimi a potom otočené zvyšné cifry, t.j. otočená celočíselná časť po delení desiatimi.

S rekurzíu sa často stretneš pri definícii rôznych výrazov. Napríklad takto:

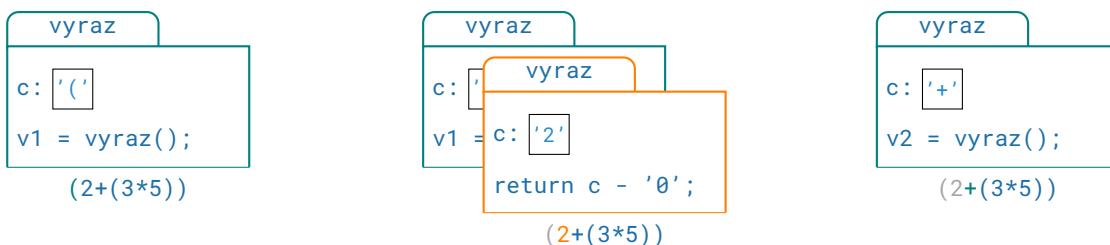
**Úloha 37.** Zjednodušený výraz je bud' jednocierné číslo, alebo má tvar  $(\clubsuit + \heartsuit)$ , alebo  $(\clubsuit * \heartsuit)$ , kde  $\clubsuit$  a  $\heartsuit$  sú zjednodušené výrazy. Na vstupe zjednodušený výraz končí znakom  $\$$ . Napiš program, ktorý prečíta zo vstupu zjednodušený výraz a vyráta jeho hodnotu.

Zo zadania vidíme, že zjednodušené výrazy sú napr.  $4\$$  alebo alebo  $(2+((3*2)+(1+(6*3))))\$$ , ale nie  $12\$$  (nie je jednocierné), ani  $5+1\$$  (nie je v zátvorkách), ani  $(3+8+1)\$$ . Na čítanie vstupu použijeme premennú typu `char c`; (pozri kapitolu 7). Príkaz `cin >> c`; prečíta jeden znak zo vstupu<sup>2</sup>. Ak je tento znak číslica, t.j. `c >= '0' && c <= '9'`, tak hodnota výrazu je číslo, ktoré táto číslica reprezentuje, t.j. `c - '0'`; (tu využívame, že v ASCII tabuľke idú znaky číslíca za sebou). Ak prvý znak nebola číslica, musela to byť otváracia zátvorka. Hodnotu výrazu potom získame tak, že rekurzívne vyhodnotíme prvý výraz, prečítame znak operácie, rekurzívne vyhodnotíme druhý výraz a vrátime výsledok (predtým ale nezabudneme prečítať zatváraciu zátvorku). Výsledok by mohol vyzerať takto:

```

1 int výraz() {
2     char c;
3     cin >> c;
4     if (c >= '0' && c <= '9') return c - '0';
5     int v1 = výraz(); // rekurzívne volanie
6     cin >> c;          // operátor
7     int v2 = výraz(); // druhé rekurzívne volanie
8     int výsledok;
9     if (c == '+')
10         výsledok = v1 + v2;
11     else
12         výsledok = v1 * v2;
13     cin >> c;          // zatváracia zátvorka
14     return výsledok;
15 }
```

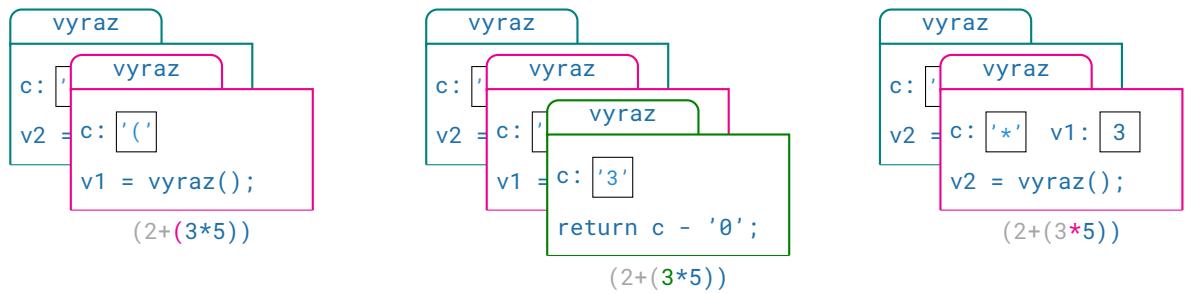
Podme sa pozrieť, čo sa stane, ak zavoláš `výraz()` a na vstupe bude  $(2+(3*5))$ . Najprv sa vytvorí svet pre funkciu `výraz()`, v nôm sa načíta znak `'('` do premennej `c`, zistí sa, že to nie je číslica, tak sa príde na príkaz `int v1 = výraz();`. Tu sa spracovanie preruší a vytvorí sa nový svet (nezávislej inštancie) funkcie `výraz()`. Ten má svoju vlastnú premenňu `c`, do ktorej sa načíta znak `'2'` zo vstupu. Kedže je to číslica, vráti sa hodnota `2` a svet zanikne. Teraz sa výpočet vrátil do prvého sveta, ktorý pokračuje tam, kde prestal. Do premennej `c` sa načíta ďalší znak zo vstupu, teraz je to `'+'` a príde sa na príkaz `int v2 = výraz();`.



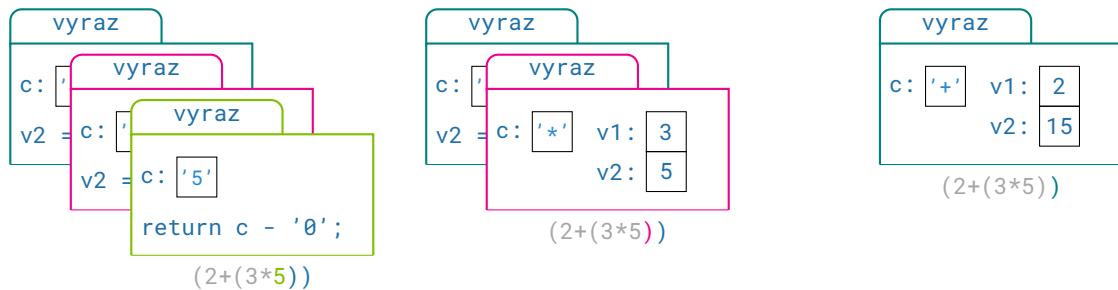
Opäť sa práca preruší, vytvorí sa nezávislý svet a načíta sa znak. Kedže je to `'('`, pokračuje sa až po volanie `v1 = výraz();`. Opäť sa vytvorí nový svet (dva svety už čakajú prerušené) a spustí sa funkcia `výraz`. Prečíta sa číslica `'3'` a do predchádzajúceho sveta vráti sa číslo `3`. Druhý svet pokračuje, prečíta znak `*` a príde na volanie `int v2 = výraz();`

<sup>2</sup>Ako sme v kapitole 7 hovorili, pri načítavaní sa preskakujú whitespace znaky. Toto správanie sa dá prepnúť, ak z `cin` načítaš špeciálnu hodnotu `noskipws`, t.j. `cin >> noskipws`; spôsobí, že sa budú načítať všetky znaky vrátane medzier.

## Rekurzia



Teraz sa zase spraví nový svet, ktorý načíta číslo 5. Dočíta sa zátvorka a vráti sa výsledok 15. Pôvodný svet dočíta poslednú zátvorku a vráti výsledok 17.



Počas rekurzie vzniká veľa na sebe nezávislých svetov (v našom príklade boli na sebe najviac tri, ale pre iné vstupy ich môže byť oveľa viac) tej istej funkcie, ktoré všetky čakajú, kým ten aktuálny práve ráta. Počet čakajúcich svetov sa niekedy volá *hlbka rekurzie*.

Keby si chcel mať viacičerné čísla, uvedom si, že napr. hodnota čísla '**4567**' je desaťkrát hodnota čísla '**456**' plus hodnota cifry '**7**'.

**Úloha 38.** Uprav predchádzajúce riešenie tak, aby pracovalo s viacciernými číslami.

Niekedy je dobé mať globálne premenné (napr. pole), do ktorého rekurzívna procedúra postupne generuje všetky možnosti: skúsi prvú možnosť, rekurzívne doplní zvyšok, skúsi druhú možnosť, atď. Napríklad chcem pre zadané  $n$  vypísať všetky  $n$ -bitové čísla v dvojkovej súštave, napr. v poradí ako vpravo. Vypísať všetky čísla môžem tak, že najprv vypíšem všetky, ktoré sa začínajú nulou a potom všetky, ktoré sa začínajú jednotkou. Všetky  $n$ -bitové čísla, ktoré sa začínajú nulou, viem vypísať tak, že vypíšem všetky  $n-1$ -bitové čísla, akurát že pred každým napišem nulu. Ako to naprogramovať s pomocou rekurzie?

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

Urobím si pole, v ktorom bude  $n$  čísel. Začнем tak, že na prvé miesto napišem nulu a rekurzívne sa zavolám na zvyšok. Budem ale potrebovať písť na iné miesta v poli, preto moja funkcia bude mať ako parameter, na ktoré miesto poľa má zapisovať. Ked' sa v rekurzii dostanem až na koniec poľa, jednoducho ho vypíšem. Mohlo by to vyzeráť napr. takto:

```

1 #include <iostream>
2 using namespace std;
3
4 int a[100];
5 int n;
6
7 void vypis(int k) {
8     if (k == n) {

```

```

9   for (int i = 0; i < n; i++) cout << a[i] << " ";
10  cout << endl;
11 } else {
12   a[k] = 0;
13   vypis(k + 1);
14   a[k] = 1;
15   vypis(k + 1);
16 }
17 }
18
19 int main() {
20   cin >> n;
21   vypis(0);
22 }
```

Čo ak teraz chcem, aby som čísla vypisoval v takom poradí, že sa vždy zmení iba jedna cifra<sup>3</sup>, napr:

```

0 0 0
0 0 1
0 1 1
0 1 0
1 1 0
1 1 1
1 0 1
1 0 0
```

Všimni si, že ten výpis má podobnú vlastnosť, ako ten minulý: najprv idú všetky čísla, ktoré sa začínajú nulou, a potom všetky také, ktoré sa začínajú jednotkou. Akurát, že tie jednotkové musím vypisovať v opačnom poradí. Ako to jednoducho v rekurzii docielim? Namiesto toho, aby som najprv napísal nulu a potom jednotku stačí, aby som najprv nechal číslo, ktoré tam bolo pôvodne (predpokladám, že na začiatku mám v poli samé nuly) a potom opačné číslo. Premysli si, prečo je to tak. Výsledok by teda mohol byť:

```

1 #include <iostream>
2 using namespace std;
3
4 int a[100];
5 int n;
6
7 void vypis(int k) {
8   if (k == n) {
9     for (int i = 0; i < n; i++) cout << a[i] << " ";
10    cout << endl;
11  } else {
12    vypis(k + 1);
13    a[k] = 1 - a[k];
14    vypis(k + 1);
15  }
16 }
17
18 int main() {
19   cin >> n;
20   for (int i = 0; i < n; i++) a[i] = 0;
21   vypis(0);
22 }
```

<sup>3</sup>takéto usporiadanie sa volá Grayov kód

## Rekurzia

---

**Úloha 39.** Napíš program, ktorý prečíta čísla  $n, m, k$  a vypíše všetky postupnosti dĺžky  $n$  zložené z čísel  $0, 1, 2$ , ktoré obsahujú  $m$  jednotiek,  $k$  dvojok a ostatné sú nuly. Napr. pre vstup  $4 \ 2 \ 1$  vypíše 12 možností:

0	1	1	2
0	1	2	1
0	2	1	1
1	0	1	2

1	0	2	1
1	1	0	2
1	1	2	0
1	2	0	1

1	2	1	0
2	0	1	1
2	1	0	1
2	1	1	0

**Úloha 40.** Napíš program, ktorý prečíta číslo  $n$  a vypíše všetky možné usporiadania čísel  $1, 2, \dots, n$ . Napr. pre  $n = 3$  vypíše (v nejakom poradí):

1	2	3
1	3	2
2	1	3
2	3	1
3	2	1
3	1	2

**Úloha 41.** Na vstupe sú čísla  $n, m$  a dvojrozmerné pole nul a jednotiek s  $n$  riadkami a  $m$  stĺpcami. Napokon sú tam dve čísla  $r, s$ . Napíš program, ktorý vyplní jednotkami uzavretú oblasť nul ohraničenú jednotkami, ktorá obsahuje poličko na riadku  $r$  a stĺpco  $s$ . Konkrétnie treba urobiť toto: prefarbiť poličko na riadku  $r$  a stĺpco  $s$  na 1, pozrieť sa na štyroch susedov (hora, dole, vľavo a vpravo) a pre každého z nich, ktorý má hodnotu 0, urobiť to isté (t.j. prefarbiť ho na 1, pozrieť sa na štyroch susedov, ...)

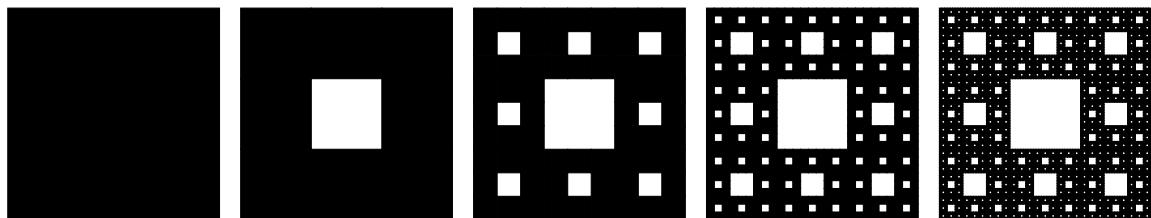
Vstup:

8	11										
0	0	1	1	1	1	0	0	0	0	0	0
0	1	0	0	0	0	1	0	0	1	0	
0	1	0	0	0	0	1	0	1	0	1	
0	1	0	0	0	0	1	0	1	0	1	
0	0	1	0	0	0	1	0	1	0	1	
0	0	1	0	0	0	1	0	0	1	0	
0	0	1	0	1	0	1	1	1	0	1	
0	0	1	0	0	0	0	0	0	1	1	
0	0	0	1	1	1	1	1	1	1	0	

Výstup:

0	0	1	1	1	1	1	0	0	0	0	0
0	1	1	1	1	1	1	1	0	0	1	0
0	1	1	1	1	1	1	1	0	1	1	1
0	0	1	1	1	1	1	1	0	1	1	1
0	0	1	1	1	1	1	1	0	0	1	1
0	0	1	1	1	1	1	1	1	0	1	1
0	0	1	1	1	1	1	1	1	1	1	1
0	0	0	1	1	1	1	1	1	1	1	0

**Úloha 42.** Sierpiňského koberec je vzor, ktorý je popísaný rekurzívne. Koberec úrovne nula je čierny štvorec. Koberec úrovne  $n$  dostaneme tak, že zoberieme štvorec, rozdelíme ho na mriežku  $3 \times 3$ , stredný štvorec vyrežeme, a osem zvyšných štvorcov nahradíme kópiami koberca úrovne  $n - 1$ . Prvých päť kobercov vyzerá takto:



Napiš program, ktorý zo vstuпу prečíta  $n$  a vytvorí obrázok rozmerov  $3^n \times 3^n$  (t.j. úroveň 0 má stranu dĺžky 1, úroveň 1 stranu 3, a úroveň  $n$  stranu  $3^n = \overbrace{3 \cdot 3 \cdots 3}^n$ ), v ktorom bude koberec úrovne  $n$ .

Ked' máme niečo naprogramovať, väčšinou je viac možností, ako to urobiť. A nie všetky sú rovnako dobré. Fibonacciho čísla z úlohy 11 môžem naprogramovať dvoma spôsobmi takto:

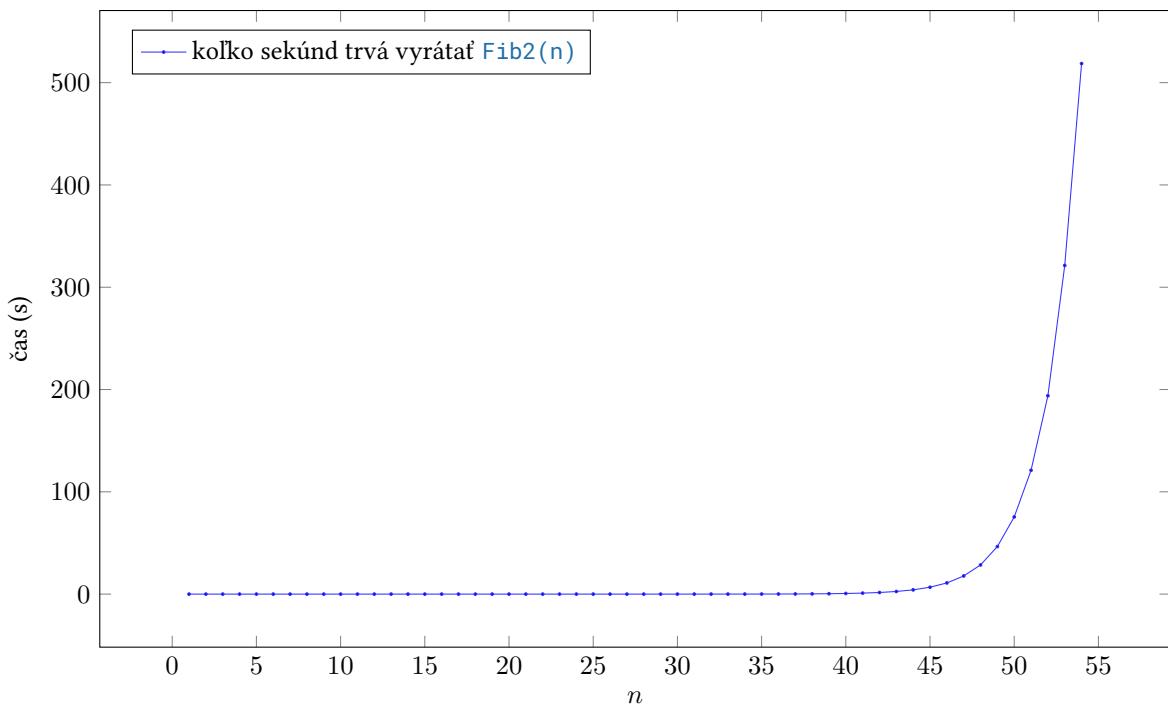
```

1 long int Fib1(int n) {
2     if (n < 3) return n - 1;
3     long int a = 0, b = 1, c;
4     int i;
5     for (i = 2; i < n; i++) {
6         c = a + b;
7         a = b;
8         b = c;
9     }
10    return c;
11 }
```

```

1 long int Fib2(int n) {
2     if (n < 3) return n - 1;
3     return Fib2(n - 1) + Fib2(n - 2);
4 }
```

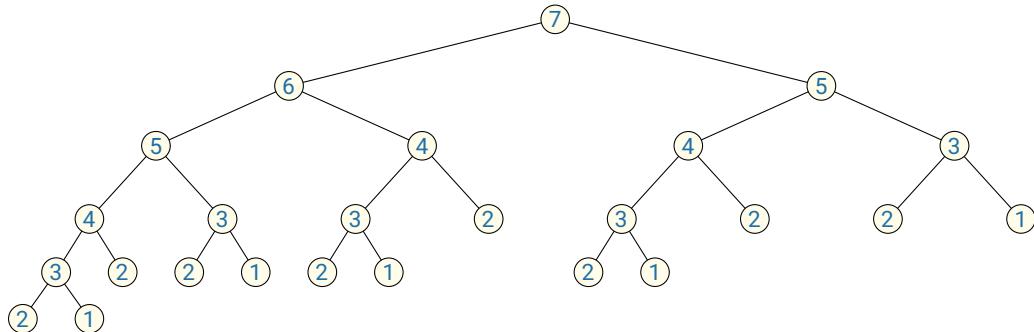
Rekurzívna verzia vpravo je kratšia a lepsie pochopiteľná, lebo z nej priamo vidíme postup *n-té číslo vyrátať tak, že sčítaš dve predchádzajúce*. Má to ale háčik. Začal som si merať čas, kolko trvá, kým sa vyráta  $n$ -té číslo. Funkcia Fib1 všetky čísla po 90 (cca toľko sa zmestí do `long int`) vyráta pod jednu milisekundu. Pri funkcií Fib2 som sa dostať po 54-té číslo. Tam to trvalo cca 518 sekúnd a potom ma to prestalo baviť. Tu vidiš graf, ako závisel čas behu od velkosti  $n$ . Čísla [Fib2\(90\)](#) by som sa nedočkal ani do konca vesmíru.



Problém tu bol v tom, že rekurzívna verzia ráta veľa vecí zbytočne: keď chce vyrátať `Fib2(10)`, najprv ráta `Fib2(9)`. Pri rátaní `Fib2(9)` vyráta `Fib2(8)=13` a `Fib2(7)=8`, takže vráti hodnotu `Fib2(9)=21`. Pokračuje sa v rátaní `Fib2(10)`: `Fib2(9)` už je vyrátané, ide sa rátať `Fib2(8)`. To sa už sice raz rátalo, ale výsledok zanikol spolu so svetom predchádzajúceho rekurzívneho volania, takže sa bude rátať znova. Takto sa nabaľuje veľa zbytočnej práce a už aj pre relatívne malé  $n$  to začne byť celkom dosť problém. Napr. na vypočítanie `Fib2(7)` sa v rekurzii postupne vyrobí všetkých 25 svetov z nasledujúceho obrázka, ale na `Fib2(50)` by sa už vyrobilo

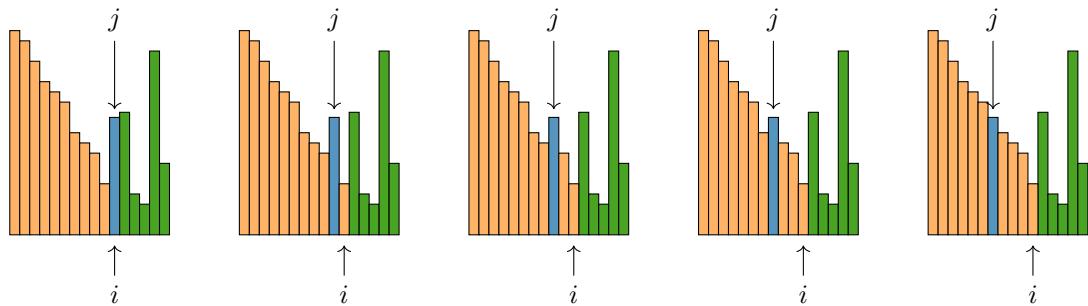
## Zložitosť

25172538049 svetov.



Ked' máme dva programy, ako môžeme povedať, ktorý z nich je rýchlejší? Keby som ti povedal, že vyrátať 37. Fibonacciho číslo mi tvalo 140 ms, nič ti to nepovie, lebo to závisí od toho, aký mám rýchly počítač. Hlavný problém s funkciou [Fib2](#) neboli v konkrétnych časoch, ale v tom, že sa príliš rýchlo zväčšovali. Chceme teda vymyslieť spôsob, ako povedať, ako rýchlo rastie čas programu v závislosti od veľkosti vstupu. Ukážeme si to na príklade. Povedzme, že už máme načítané pole  $a$ , v ktorom je  $n$  čísel, každé z nich z rozsahu  $0, \dots, 999$ . Chcem ho preusporiadať tak, aby bolo v poradí od najväčšieho čísla po najmenšie. Podobný problém si už riešil v úlohách [18](#) a [34](#), teraz skúsme trochu iný prístup, vlastne dva.

Prvý program je tzv. *Insertion Sort*. Idea je takáto: budeme sa snažiť prehadzovať prvky v poli tak, aby nakoniec polo utriedené. Algoritmus bude pracovať v kolách. Na začiatku  $i$ -teho kola chceme, aby platilo, že prvých  $i$  pozícii je utriedených. Na začiatku prvého kola to platí – prvý jeden prvk je utriedený vždy. Ak sa nám podarí, aby to platilo aj po  $n$  kolách, budeme mať utriedených prvých  $n$  pozícii, t.j. celé pole. Predpokladajme teraz, že sme na začiatku  $i$ -teho kola. Skontrolujeme, či je prvak  $a[i]$  je menší, ako  $a[i-1]$ . Ak áno, môžeme toto kolo skončiť, lebo prvých  $i + 1$  prvkov je utriedených. Ak nie, vymeníme prvky  $a[i]$  a  $a[i-1]$  a skontrolujeme, či je  $a[i-1]$  menší ako  $a[i-2]$ . Taktô pokračujeme ďalej, kým nedostaneme prvok  $a[i]$  na správne miesto:



Celá funkcia bude vyzerať takto<sup>1</sup>:

<pre> 1 void vymen(int i, int j) { 2     int x; 3     x = a[i]; 4     a[i] = a[j]; 5     a[j] = x; 6 }</pre>	<pre> 1 void InsertSort(int n) { 2     int i, j; 3     for (i = 1; i &lt; n; i++) 4         for (j = i; j &gt; 0 &amp;&amp; a[j - 1] &lt; a[j]; j--) 5             vymen(j - 1, j); 6 }</pre>
--	---

Druhý program, tzv. *CountSort* vychádza z toho, že vieme, že vstupné čísla sú z malého rozsahu (od 0 do 999). Vyrobíme si pomocné pole  $\text{pocty}[1000]$ , v ktorom si budeme pamätať počty jednotlivých prvkov ( $\text{pocty}[i]==j$  znamená, že sa v poli vyskytuje  $j$ -krát hodnota  $i$ ). Takže raz prejdeme vstupným poľom a

<sup>1</sup>premysli si, čo by sa stalo, keby som vo funkcií `vymen` napísal iba  $a[i]=a[j]$ ;  $a[j]=a[i]$ ;

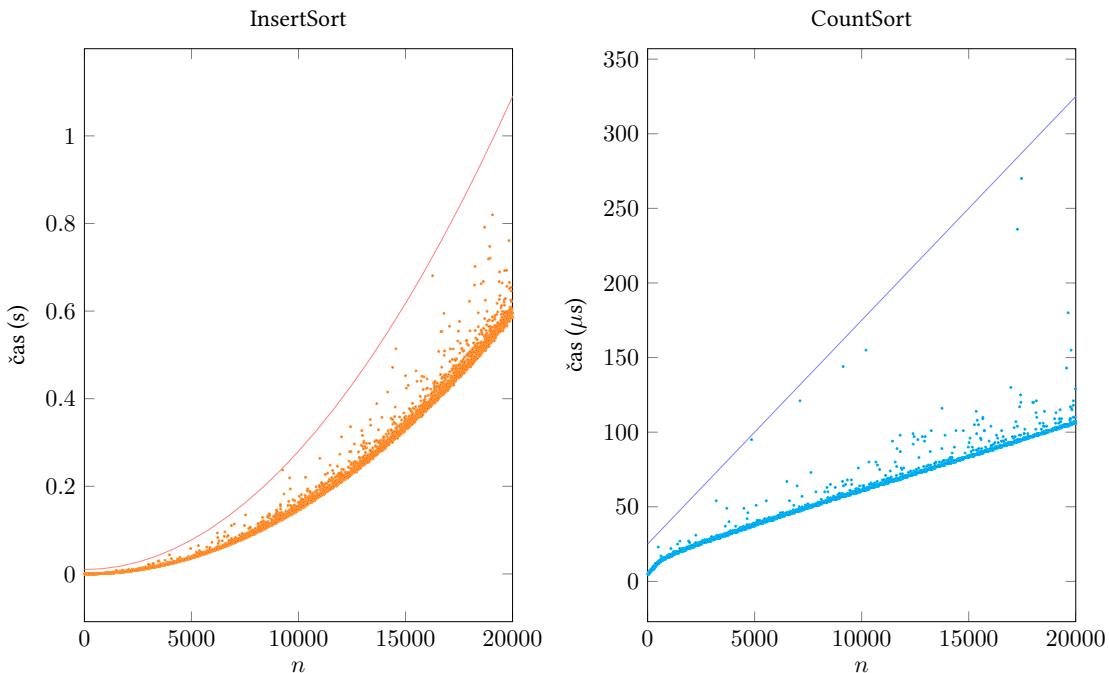
zistíme počty a potom len zapíšeme správny počet hodnôt takto:

```

1 void CountSort(int n) {
2     int pocty[1000];
3     int i, c;
4
5     for (i = 0; i < 1000; i++) pocty[i] = 0;
6     for (i = 0; i < n; i++) pocty[a[i]]++;
7     i = 0;
8     for (c = 999; c>=0 ; c--) {
9         while (pocty[c] > 0) {
10            a[i] = c;
11            i++;
12            pocty[c]--;
13        }
14    }
}

```

Teraz som spravil to, že som oba programy veľakrát spustil na rôznych poliach a meral im čas. Výsledky sú v nasledovných grafoch<sup>2</sup>



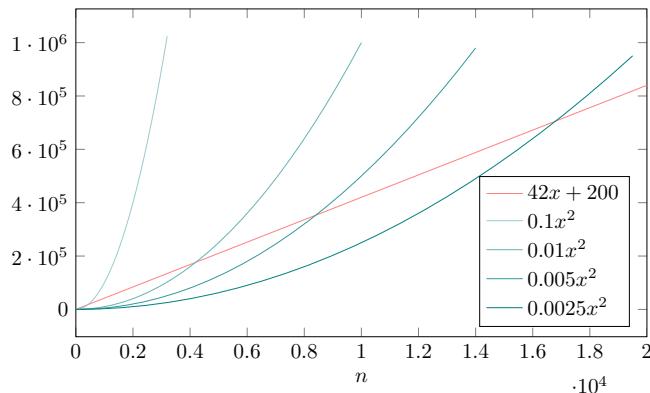
Z grafov vidno, že obom programom väčšie vstupy trvajú dlhšie, ale aj na rovnako dlhých vstupoch môžu bežať rôzne dlho (podľa toho, aké konkrétné čísla v tom poli sú). Ničmenej všetky body na ľavom obrázku sa dajú schovať pod parabolou<sup>3</sup> a všetky body vpravo pod priamku. Preto hovoríme, že InsertSort má kvadratickú zložitosť, kým CountSort má lineárnu.

Je program s lineárной zložitosťou vždy lepší, ako ten s kvadratickou? Predpokladajme, že máme lineárny program  $\mathcal{A}$ , ktorého čas na vstupe veľkosti  $n$  je najviac  $42n + 200$ . Majme hocjaký kvadratický program  $\mathcal{B}$ , ktorého čas na vstupe veľkosti  $n$  je  $an^2 + b$  pre nejaké čísla  $a, b$ . Pre malé hodnoty  $n$  môže byť program  $\mathcal{B}$  rýchlejší. Ale existuje  $n$ , kedy  $24n + 200 = an^2 + b$ , a od tohto  $n$  ďalej už bude rýchlejší program  $\mathcal{A}$ .

<sup>2</sup>Vyzierajú sice podobne, ale všimni si, že obrázok vľavo je v sekundách a vpravo v mikrosekundách. CountSort bol na dlhých poliach výrazne rýchlejší.

<sup>3</sup> graf funkcie  $y = ax^2 + b$  pre nejaké čísla  $a, b$

## Zložitosť



Inými slovami, pre každý kvadratický program platí, že pre dosť veľké vstupy je  $\mathcal{A}$  rýchlejší. Podobne vieme hovoriť o kubických programoch, programoch štvrtého stupňa atď. A ako by dopadol program [Fib2](#)? Jeho zložitosť je až  $2^n$ : rastie rýchlejšie, ako akýkoľvek polynóm  $x^c$ . Takýmto programom hovoríme *exponenciálne*.

Typický príklad programu s lineárnom zložitosťou je jeden cyklus, ktorý raz prejde cez vstupné hodnoty a niečo na nich ráta (napr. hľadá maximum): keď bude vstup dvojnásobne dlhý, program pobeží dvakrát dlhšie – to je lineárna závislosť. Dva vnorené cykly takto:

```

1 for (i = 0; i < n; i++)
2   for (j = 0; j < n; j++)
3     if (i != j) {
4       x = a[j] - a[i];
5       if (x < 0) x = -x;
6       if (x < m) m = x;
7   }

```

sú zas typický príklad kvadratického programu: keď má vstup dĺžku  $n$ , vnútro cyklu sa vykoná  $n^2$ -krát (mimochodom: skús sa zamyslieť, čo ten program robí). Vnútro cyklu trvá vždy rovnako dlho (je tam len niekoľko priradení a testov), nech je to  $s$  milisekúnd. Čas celého programu preto bude  $n^2 \cdot s$  milisekúnd, čo je kvadratická závislosť. Treba byť ale opatrný, nie všetky vnorené cykly znamenajú kvadratickú zložitosť. Pozrime sa na takýto príklad: v poli  $a$  je uložených  $n$  rôznych čísel. Naším cieľom je nájsť najdlhší úsek, v ktorom sú čísla utriedené od najmenšieho. Napr. v poli 12 4 5 8 6 5 10 9 11 13 14 3 sú tri také úseky označené farebne a najdlhší z nich má dĺžku 4.

Priamočiary spôsob riešenia je pre každú pozíciu v poli vyskúsať, aký dlhý rastúci úsek sa z nej začína. Budeme mať teda jednu premennú  $i$ , ktorou prejdeme v cykle cez celé pole a zakaždým v druhom cykle premennou  $j$  zistíme dĺžku rastúceho úseku. Napríklad pre  $i = 1$ , bude najprv  $j = 2$  a pretože  $a[2] = 5 > 4 = a[1]$ ,  $j$  sa nastaví na 3 a cyklus sa zopakuje. Rovnako  $a[3] = 8 > 5 = a[2]$ , preto sa cyklus opäť zopakuje a  $j$  bude 4. Teraz  $a[4] = 6 < 8 = a[3]$ , preto cyklus skončí s hodnotou  $j = 4$ . Dĺžka rastúceho úseku so začiatkom v  $i$  je preto  $j - i$ .

$i = 1$	$j = 4$
↓	↓
12 4 5 8 6 5 10 9 11 13 14 3	

Program by vyzeral napr. takto:

```

1 int usek1() {
2   int i = 0, j, m = 0;
3   while (i < n - 1) {
4     j = i + 1;
5     while (j < n && a[j] > a[j - 1])
6       j++;
7     if (j - i > m) m = j - i;
8     i++;
9   }
10  return m;
11 }

```

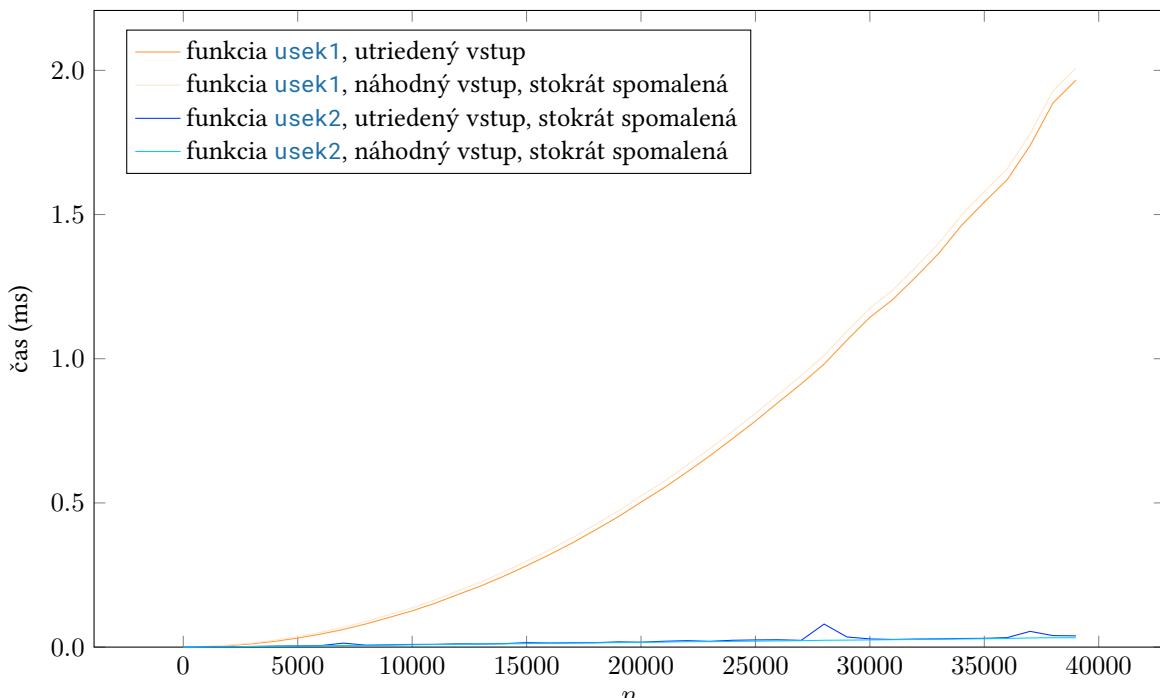
V programe máme dva vnorené cykly. Kolkokrát sa vykoná inštrukcia `j++` vo vnútornom cykle? Predpokladajme, že na vstupe sú čísla usporiadane vzostupne  $1 \ 2 \ 3 \ \dots \ n$ . Na začiatku je  $i=0$ , takže vnútorný cyklus začína od 0, a keďže je celé pole rastúce, zopakuje sa  $n - 1$  krát. Potom sa nastaví  $i=1$  a vnútorný cyklus sa zopakuje  $n - 2$  krát atď. Celkovo sa teda vnútorný cyklus zopakuje  $(n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$  krát. Ako takýto súčet môžeme vyrátať? Stačí si preusporiadať čísla: prvé s posledným dá dokopy  $(n-1) + 1 = n$ . Druhé s predposledným dá  $(n-2) + 2 = n$  atď. Dokopy takto získame  $(n-1)/2$  dvojíc po  $n$ , teda výsledok<sup>4</sup> je  $n(n-1)/2$ , čo je kvadratická funkcia.

Vedel by si zložitosť tohto programu vylepšíť? Stačí si uvedomiť, že veľa robity sa robí zbytočne. Napríklad pre  $i = 1$  skončí vnútorný cyklus s hodnotou  $j = 4$ . Čo sa stane pre  $i = 2$ ? Vnútorný cyklus opäť príde po  $j = 4$  a potom zastane, lebo nájde menšie číslo  $a[4] = 6 < 8 = a[3]$ . Samozrejme, rastúci úsek od  $i = 2$  je kratší. Nás ale zaujíma najdlhší rastúci úsek, preto nepotrebuje skúsať  $i = 2$  ani  $i = 3$ , ale najbližšie  $i$ , ktoré potrebujeme skúsiť je  $i = j$ . Upravme program takto:

```

1 int usek2() {
2     int i = 0, j, m = 0;
3     while (i < n - 1) {
4         j = i + 1;
5         while (j < n && a[j] > a[j - 1]) j++;
6         if (j - i > m) m = j - i;
7         i = j;
8     }
9     return m;
10 }
```

Zdanlivo je to malá zmena, stále máme dva vnorené cykly, ale vidno, že každá premenná v poli `a` sa testuje na  $a[j] > a[j - 1]$  iba raz za celý program (ked sa raz v nejakom cykle otestuje, ďalšia iterácia vonkajšieho cyklu ju už preskočí). Funkcia `usek2` má teda lineárnu zložitosť. Naozaj sa ten rozdiel prejaví? Skúšal som si merať časy na rôznych vstupoch. Pre funkciu `usek1` je najhoršie, ak je vstup celý rastúco utriedený, lebo vtedy vnútorný cyklus vždy prejde až do konca. Tento prípad bol tak pomaly, že všetky ostatné som musel stokrát spomalíť, aby na obrázku bolo vôbec niečo vidno. Okrem toho som skúšal aj náhodné postupnosti pre rôzne dĺžky (pre každú dĺžku som zobraľ priemer 100 náhodných vstupov). Výsledky vyzerajú takto:



<sup>4</sup>Toto, čo sme povedali, platí pre nepárne  $n$ , aby  $(n-1)/2$  bolo celé číslo. Ako je to pre párne  $n$ ? Dostaneme  $(n-2)/2$  dvojíc po  $n$  a ešte v strede ostane jedno číslo  $n/2$ . Dokopy to teda bude  $n(n-2)/2 + n/2 = (n^2 - 2n)/2 + n/2 = (n^2 - 2n + n)/2 = (n^2 - n)/2 = n(n-1)/2$ , takže výsledok sedí aj v tomto prípade.

## Zložitosť

Na to, aby sme určili zložitosť (t.j. aby sme vedeli povedať, či sa čas behu programu v závislosti od veľkosti vstupu vždy zmestí pod priamku, parabolu, atď) treba vedieť, čo je pre program najhorší vstup a celé to môže byť aj celkom ľažké. Ale je treba si zapamätať, že tá istá úloha sa dá naprogramovať rôznymi spôsobmi a veľmi záleží na tom, ako to urobiš. Pri súťažiach je pre úlohy väčšinou stanovený aj časový limit: ak tvoj program neskončí dosť rýchlo, testovač ho prerusí a neuzná (dostaneš odpoveď TLE = *time limit exceeded*). V tom prípade máš dve možnosti: skúsiť nejaké finty, ako veci trochu zrýchliť, alebo sa zamyslieť nad úplne iným spôsobom, ako úlohu naprogramovať.

Tu je zopár úloh, ktoré sa dajú naprogramovať rôzne rýchlo. Skús nájsť najrýchlejší spôsob.

**Úloha 43.** Na vstupe je číslo  $n$ , potom pole  $n$  čísel a potom  $n$  otázok, pričom každá otázka pozostáva z dvoch čísel  $i, j$ , pričom platí  $0 \leq i \leq j < n$ . Úlohou je napísanie programu, ktorý pre každú otázku  $i, j$  vypočíta súčet  $a[i] + a[i+1] + \dots + a[j]$ . Napr. pre pole  $5 \ 7 \ -1 \ 8 \ 3 \ -2$  a otázku  $1 \ 3$  je odpoveď  $7 - 1 + 8 = 14$ .

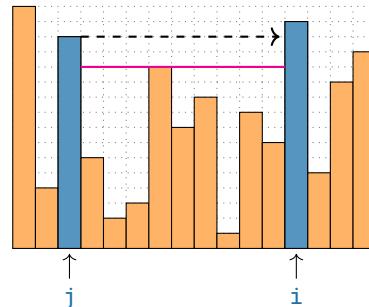
**Úloha 44.** Na vstupe je číslo  $n$ , utriedené pole  $n$  čísel a číslo  $k$ . Úlohou je napísanie programu, ktorý nájdzie počet dvojíc čísel, ktorých rozdiel je  $k$ . Napr. pre pole  $3 \ 4 \ 7 \ 7 \ 8 \ 11 \ 12 \ 16$  a  $k = 5$  je odpoveď  $3$ , lebo v poli sú dvojice  $(3, 8)$ ,  $(7, 12)$ ,  $(11, 16)$ .

Na záver ešte jedna ukážka, ktorú použijeme aj v ďalšej kapitole.

**Úloha 45.**  $n$  klaunov s výškami  $1, 2, \dots, n$  sa postavilo do radu v nejakom poradí. Každý z nich hodil smerom doprava šľahačkovú tortu, ktorá dopadla na najbližšieho vyššieho klauna (ak tam taký neboli, torta preletela preč). Napíš program, ktorý načíta číslo  $n$  a poradie klaunov a vypočíta, kolko najviac zásahov nejaký klaun dostal. Napr. pre  $n = 5$  a poradie  $3 \ 2 \ 5 \ 4 \ 1$  je odpoveď  $2$ , lebo na klaunovi  $5$  pristane torta od dvojký aj trojky.

Priamočiary spôsob riešenia je napísanie funkcie, ktorá pre dvojicu klaunov  $j$  a  $i$  povie, či klaun  $j$  trafi klauna  $i$ . To je jednoduché: stačí zistiť, či je medzi nimi niekto vyšší:

```
1 bool trafi(int j, int i) {
2     int k;
3     if (a[i] < a[j]) return false;
4     for (k = j + 1; k < i; k++)
5         if (a[k] > a[j]) return false;
6     return true;
7 }
```

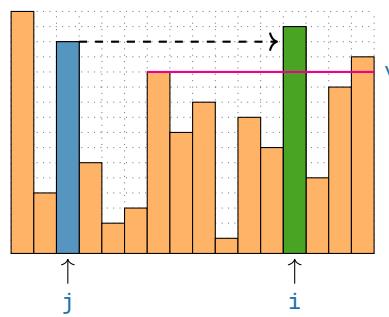


S pomocou tejto funkcie už ľahko otestujeme všetky dvojice klaunov:

```
1 int klauni1() {
2     int i, j, max = 0;
3     for (i = 1; i < n; i++) { // skúšame každého klauna
4         int hits = 0; // kolko zásahov dostal
5         for (j = 0; j < i; j++) // všetci klauni naľavo od i
6             if (trafi(j, i)) hits++;
7         if (hits > max) max = hits;
8     }
9     return max;
10 }
```

Akú má tento program zložitosť? Funkcia `trafi(j, i)` urobí  $i - j$  operácií. Na otestovanie klauna  $i$  voláme `trafi(j, i)` pre všetky menšie  $j$ , preto ten test spotrebuje  $i + (i - 1) + (i - 2) + \dots + 1$  operácií. Tači ktorý súčet si už rátal, je to  $(i + 1)i/2$ . Napokon toto celé robíme pre všetkých klaunov, teda dokopy máme  $(1 + 1)1/2 + (2 + 1)2/2 + (3 + 1)3/2 + \dots + n(n - 1)/2$  operácií. Presne zrátať tento súčet je trochu ľahšie<sup>5</sup>, ale ľahko môžeš odhadnúť, že  $(i + 1)i/2 > i^2/2$ , a preto v súčte je aspoň  $(n - 1)/2$  členov, z ktorých každý je väčší ako  $(n/2)^2/2$ . Dokopy je teda zložitosť väčšia ako  $\frac{n-1}{2} \frac{(n/2)^2}{2} = \frac{1}{4}(n-1)n^2/4 > \frac{1}{16}n^3$ . Takáto zložitosť sa volá *kubická* (čas sa vždy zmestí pod graf funkcie  $an^3$  pre nejaké číslo  $a$ ). Vzťah medzi kubickou a kvadratickou zložitosťou je rovnaký ako medzi kvadratickou a lineárnu: hocjaký kvadratický program začne byť pre dosť veľké vstupy lepší ako daný kubický.

Podľme to trochu vylepšiť. Kde sa robí zbytočná robota? Vo funkcii `trafi` zakaždým hľadáme najvyššieho klauna medzi  $i$  a  $j$ . Túto prácu si vieme ušetriť, ak si ho budeme pamätať priebežne a iba aktualizovať. Vo vylepsenej verzii budeme opäť prechádzat v cykle cez všetkých klaunov (na to bude premenná `i`) a pre každého zistíme, kolko zásahov dostane. To budeme robiť v druhom cykle (s premennou `j`). Budeme postupne prechádzat klaunov od  $i - 1$  smerom doľava, až kým neprídeme na začiatok, alebo nestretнемe väčšieho klauna (spoza neho už klauna `i` nikto netrafi). Každý klaun  $j$ , ktorého takto stretнемe, trafi klauna `i`, ak je vyšší, ako najvyšší klaun, ktorý je medzi nimi. Jeho výšku si budeme pamätať v premennej `v` a vždy, keď stretнемe klauna  $j$ , ktorý je vyšší ako `v`, si ju upravíme.



```

1 int klauni2() {
2     int i, j, max = 0;
3     for (i = 1; i < n; i++) { // skúšame každého klauna
4         int hits = 0, // kolko zásahov dostal
5             v = 0; // najvyšší klaun medzi nimi
6         for (j = i - 1; j >= 0 && a[j] < a[i]; j--) // skúšame vľavo
7             if (a[j] > v) { // trafi ?
8                 hits++; // zarátame zásah
9                 v = a[j]; // upravíme výšku
10            }
11            if (hits > max) max = hits; // pamätáme si najzapatlanejšieho
12        }
13    return max;
14 }
```

Akú má zložitosť? Podobne, ako v predchádzajúcom príklade, najhorší prípad je, keď sú klauni utriedení: vtedy vnútorný cyklus pôjde zakaždým až po začiatok, a teda celkový počet opakovania bude  $1 + 2 + 3 + \dots + (n - 2) + (n - 1)$ , čo je kvadraticky veľa.

<sup>5</sup>výsledok je  $\frac{1}{6}n(n+1)(n+2)$

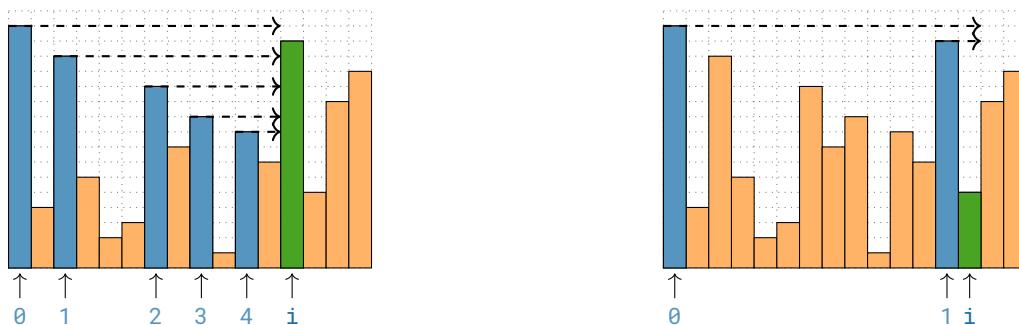
## 13 Dátové štruktúry: zásobník

Pokračujme v úlohe o klaunoch. Verzia [klauni2](#) má kvadratickú zložitosť. Dá sa to ešte zlepšiť? Stále sa ešte robí dosť zbytočnej roboty: ak je medzi klaunom  $i$  a  $j$  veľa malých klaunov, budeme ich testovať znova pri  $i+1$ ,  $i+2$  atď, aj keď je jasné, že už nikoho netrafia. Chceli by sme si preto pamätať iba takých klaunov, ktorí ešte majú šancu niekoho trafiť. Na to ale potrebujeme pomocné premenné, kde si ich uložíme, lebo vo vstupnom poli môžu byť na rôznych miestach.

Dátová štruktúra je spôsob rozmýšľania o programe, ktorý pomáha rozdeliť program na menšie nezávislé časti. Podobnú úlohu majú funkcie: v prvej verzii sme mali funkciu `trafi(j, i)`, ktorá testovala, či klaun  $j$  trafi  $i$ . V hlavnom programe sme ju mohli používať a nestarať sa o to, ako je naprogramovaná vnútri, stačila nám jej *specifikácia*, t.j. charakteristika toho, čo robí. S dátovými štruktúrami je to podobné. Povieme si, aké operácie potrebujeme s premennými vedieť robiť a zvlášť rozmyšľame o tom, ako naprogramovať tie operácie a zvlášť o tom, ako s ich pomocou vyriešiť našu úlohu.

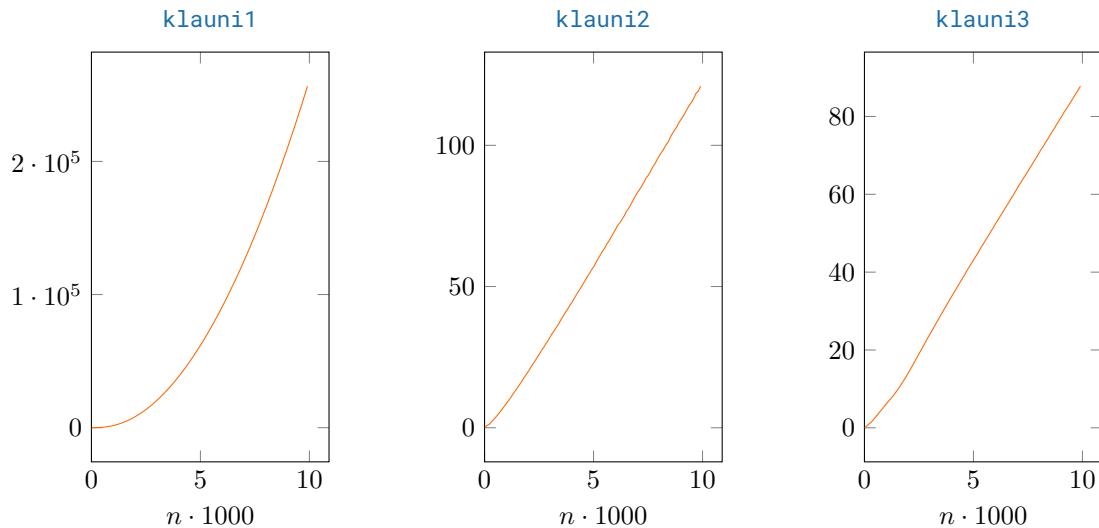
V našom prípade použijeme dátovú štruktúru *zásobník* (stack). Zásobník je kopa vecí poukladaných na sebe. Dá sa položiť vec na vrch, zobrať vec z vrchu a pozrieť sa, či je zásobník prázdný. Ak dopredu vieme, aký najväčší zásobník budeme potrebovať, ľahko ho vyrobíme v poli. Budeme mať pole `int stack[n]` a premennú `int m`, ktorá hovorí, kolko prvkov je momentálne v zásobníku. Keďže prvky číslujeme od nuly, `stack[m-1]` je prvok na vrchu, zobrať prvok z vrchu znamená urobiť `m-=1` a pridať `x` na vrch znamená `stack[m]=x; m++`.

S pomocou zásobníka vieme vylepšiť našu klaunskú úlohu. Keď testujeme klauna  $i$ , budeme mať v zásobníku uložených všetkých kandidátov: t.j. klaunov, ktorých torty ešte letia vzduchom (modré prvky v ľavom obrázku sú v zásobníku pri spracovaní klauna  $i$ ). Pri spracovaní klauna  $i$  nepôjde doľava v pôvodnom poli, ale iba v zásobníku (o klanoch mimo zásobníka vieme, že ich torty aj tak  $i$ -čka netrafia). Pre všetkých menších ako  $i$ , ktorých stremene, zarátame zásah do  $i$  a zároveň ich zo zásobníka vyhodíme, lebo za  $i$ -čkom už nikoho netrafia. Na obrázku vpravo je situácia po prechode na ďalšieho klauna.

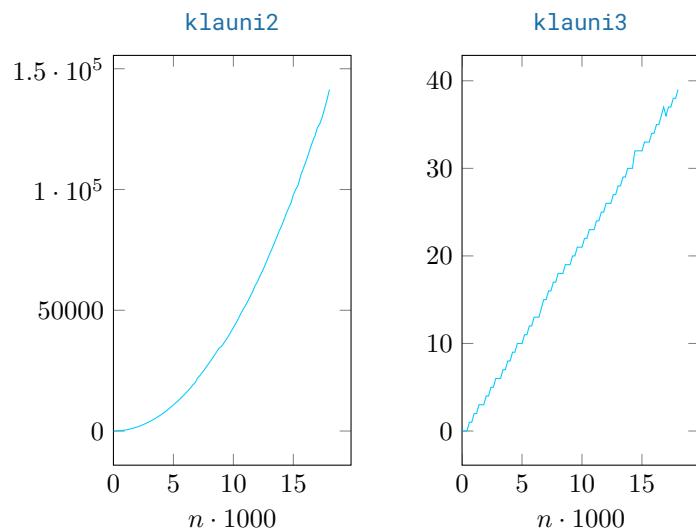


```
1 int klauni3() {
2     int i, max = 0;
3     int s[n], m = 0;           // zásobník a jeho veľkosť
4     for (i = 0; i < n; i++) {  // skúšame každého klauna
5         int h = 0;             // kolko zásahov dostal
6         while (m > 0 && s[m - 1] < a[i]) { // prechádzame zásobníkom až kým neprídeme
7                                         // na začiatok alebo na vyššieho klauna
8             h++; // zarátame zásah
9             m--; // vyhodíme zo zásobníka
10        }
11        s[m] = a[i]; // pridáme i-čka do zásobníka
12        m++;
13        if (h > max) max = h; // aktualizujeme zásahy
14    }
15    return max;
16 }
```

Akú má tento program zložitosť? Zdalo by sa, že sme si moc nepomohli, lebo stále máme dva vnorené cykly, ale v skutočnosti každý prvok raz do zásobníka pridáme a raz vyhodíme. Preto celková zložitosť bude lineárna. Ako vychádzajú skutočné časy? Tentokrát som zobrajal som pre každé  $n$  500 náhodných vstupov a vyrátal priemerný čas (v mikrosekundách).



Vidno, že verzia `klauni1` už na vstupoch dĺžky 8000 (čo nie je nijak príliš veľa) beží viac ako 1000-krát pomalšie oproti ostatným dvom. Medzi `klauni2` a `klauni3` taký veľký rozdiel nie je, pretože na náhodných vstupoch sa aj vnútorný cyklus v `klauni2` zastaví pomerne skoro. Ked si ale pozrieme utriedený vstup, situácia sa zmení:



**Úloha 46.** Na vstupe je pole  $n$  čísel. Napíš program, ktorý pre každé číslo vypíše pozíciu najbližšieho menšieho čísla vľavo, alebo  $-1$  ak také neexistuje. Napr. pre vstup  $5 \ 2 \ 3 \ 4 \ 3$  treba vypísať  $-1 \ -1 \ 1 \ 2 \ 1$ .

**Úloha 47.** Na vstupe je výraz zložený z rôznych druhov zátvoriek: ' $($ ', ' $)$ ', ' $[$ ', ' $]$ ', ' $\{$ ', ' $\}$ '. Vstup je ukončený znakom  $\$$ . Napíš program, ktorý zistí, či sú zátvorky vyvážené a či každá otvorená zátvorka je uzavretá správnou zatváracou zátvorkou. Napr. výraz " $([]())\$$ " aj " $(\$)$ " je správny, ale výraz " $({})\$$ " ani " $(\$)$ " nie je.

**Úloha 48.** Napíš program, ktorý prečíta prirodzené číslo a vypíše jeho zápis v dvojkovej sústave.

**Úloha 49.** Naprogramuj úlohu 37 bez rekurzie s použitím zásobníka.

**Úloha 50.** Na vstupe je v prvom riadku číslo  $n$  a v druhom riadku reťazec  $n$  znakov zložený z malých písmen a zátvoriek. Zátvorky sú správne uzátvorkované. Napíš program, ktorý vyrobí pole  $p$  v ktorom pre každú pozíciu vstupu, na ktorej je písmeno, bude  $-1$  a pre pozície, kde je zátvorka, bude pozícia zodpovedajúcej zátvorky. Napr. pre  $n = 25$  a reťazec  $(o(atk)((nece)e(sarp)i)p)$  má v poli  $p$  byť  $24 \ -1 \ 6 \ -1 \ -1 \ -1 \ 2 \ 22 \ 13 \ -1 \ -1 \ -1 \ -1 \ 8 \ -1 \ 20 \ -1 \ -1 \ -1 \ -1 \ 15 \ -1 \ 7 \ -1 \ 0$

**Úloha 51.** Vstup je rovnaký ako v predchádzajúcej úlohe s tým, že prvy a posledný znak sú zátvorky ' $($ ', ' $)$ '. Vstup predstavuje zašifrovaný text, ktorý treba rozlúštit takto: zober nejakú najvnútornejšiu dvojicu zátvoriek (t.j. otváraciu a zatváraciu zátvorku, medzi ktorými sú iba znaky), zátvorky zahod a znaky medzi nimi otoč. Toto opakuj, kým sa neminú zátvorky. Napíš program, ktorý dešifruje text na vstupe. Ideálne by bolo, aby mal lineárnu zložitosť. Napr. pre vstup z predchádzajúcej úlohy je dešifrovaný text [peceneprasiatko](#).

## Memoizácia a dynamické programovanie 14

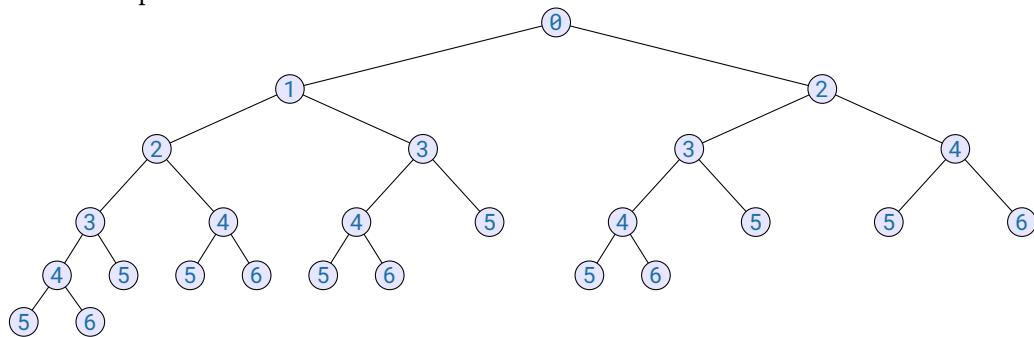
Na začiatku kapitoly 12 sme mali príklad počítania Fibonacciho čísel, na ktorom sme ukazovali, že niekedy je rekurzívna formulácia veľmi prirodzená, ale trvá dlho. Pozrime sa teraz na podobný príklad.

Na vstupe je číslo  $n$  a potom  $n$  nezáporných čísel. Naším cieľom je vybrať niekolko z nich tak, aby mali čo najväčší súčet. Je tam ale navyše podmienka, že nesmieme vybrať dve po sebe idúce čísla. Napr. pre vstup  $6 \ 20 \ 1 \ 2 \ 5$  je výsledok  $25$ , čo získame, ak vyberieme  $20$  a potom  $5$ . Viac vybrať nemôžeme, lebo ak by sme nezobrali  $20$ , tak celkový súčet všetkých ostávajúcich čísel je  $14$ . Ale ak zoberieme  $20$ , tak nemôžeme zobrať  $6$  ani  $1$ . Z dvojice  $2$  a  $5$  môžeme vybrať len jedno číslo, takže  $25$  je najviac, ako sa dá.

Túto úlohu môžeme ľahko vyriešiť rekurziou, ktorá bude skúšať všetky možnosti. Ako môže vyzeráť výber, ktorý spĺňa naše pravidlá? Budť prvé číslo  $a[0]$  nevyberieme, a potom najlepšie, čo môžeme urobiť, je vybrať z čísel  $a[1], a[2], \dots, a[n-1]$  čo najviac podľa pravidiel. Alebo prvé číslo  $a[0]$  vyberieme, potom ale nesmieme vybrať  $a[1]$  a opäť najlepšie, čo môžeme urobiť, je dovyberať podľa pravidiel z čísel  $a[2], \dots, a[n-1]$ . V oboch prípadoch vieme urobiť nejakú akciu (vybrať, či nevybrať prvé číslo) a potom musíme vyriešiť tú istú úlohu (vyberať čísla podľa pravidiel), len na menšom vstupe. Toto sa prirodzene zapíše rekurziou. Naša funkcia **najdi** bude hľadať najlepší výber počnúc číslom na pozícii  $i$ .

```
1 #include <iostream>
2 using namespace std;
3
4 int a[10000];
5 int n;
6
7 int najdi(int i) {
8     if (i >= n) return 0;
9     int x = a[i] + najdi(i + 2), y = najdi(i + 1);
10    if (x > y)
11        return x;
12    else
13        return y;
14 }
15
16 int main() {
17     cin >> n;
18     for (int i = 0; i < n; i++) cin >> a[i];
19     cout << najdi(0) << endl;
20 }
```

Na to, aby sme vyrátali hodnotu **najdi(i)**, dvakrát sa zavolá rekurzívna funkcia: raz pre **najdi(i+1)** a raz pre **najdi(i+2)**. Keď si začneme kresliť, aké svety funkcií sa budú vytvárať nre vstup s  $n=5$ , dostaneme obrázok, ktorý sme už videli pri Fibonacciho číslach:

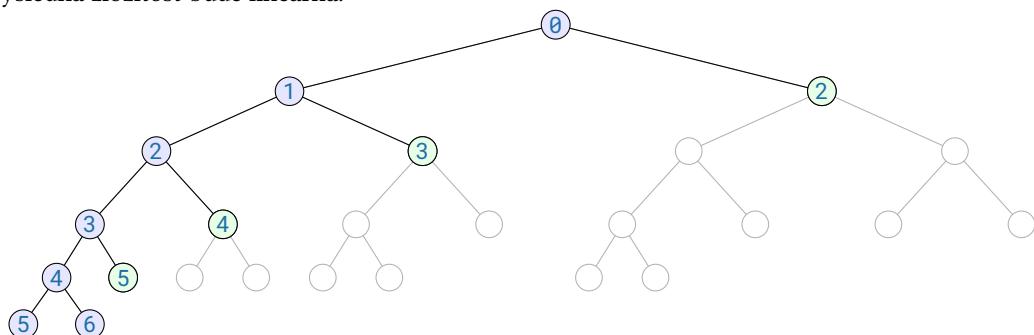


## Memoizácia a dynamické programovanie

Treba si všimnúť dve veci: po prvé, rekurzívne volania funkcie `najdi` sice narobia veľa rôznych svetov, ale parameter `i` je vždy z rozsahu 0 až  $n - 1$ , takže tých veľa volaní vždy opakuje iba málo rôznych parametrov. Navyše, zavolanie funkcie nijak neovplyvní svet naokolo<sup>1</sup>: nemení globálne premenné, nečíta vstup, ani nevypisuje, takže ak dvakrát zavolám `najdi` s rovnakým parametrom `i` (a medzitým nezmením pole `a`), dostanem rovnaké výsledky. To znamená, že ak už raz vyrátam hodnotu pre nejaké `i`, môžem si ju zapamätať a pri nasledujúcim volaní s rovnakým parametrom ju môžem použiť:

```
1 #include <iostream>
2 using namespace std;
3
4 int a[10000], m[10000];
5 int n;
6
7 int najdi(int i) {
8     if (i >= n) return 0;
9     if (m[i] >= 0) return m[i];
10    int x = a[i] + najdi(i + 2), y = najdi(i + 1);
11    if (x > y) {
12        m[i] = x;
13        return x;
14    } else {
15        m[i] = y;
16        return y;
17    }
18}
19
20 int main() {
21     cin >> n;
22     for (int i = 0; i < n; i++) {
23         cin >> a[i];
24         m[i] = -1;
25     }
26     cout << najdi(0) << endl;
27 }
```

Teraz sa pre každé `i` vyráta hodnota `najdi(i)` iba raz, pri ďalšom použití sa iba zoberie zapamätaná hodnota. Preto výsledná zložitosť bude lineárna.



Ked si porovnám časy, pôvodná verzia už pre vstup dĺžky 50 trvá vyše minúty, vylepšená verzia aj pre vstup dĺžky 1000 zbehne za zhruba milisekundu. Technika, ktorú sme práve použili, sa volá *memoizácia*<sup>2</sup>. Pri nej máme rekurzívnu funkciu (dôležité je, aby nemala vedľajšie efekty), ktorá má sice veľa volaní, ale málo rôznych parametrov a vylepšíme ju tak, že si zapamätávame už raz vyrátané výsledky, aby sme ich nemuseli rátať znova.

<sup>1</sup>hovoríme, že `najdi` nemá vedľajšie efekty

<sup>2</sup>Nevypadlo mi "r", nie je to *memorizácia*, je to *memoizácia*.

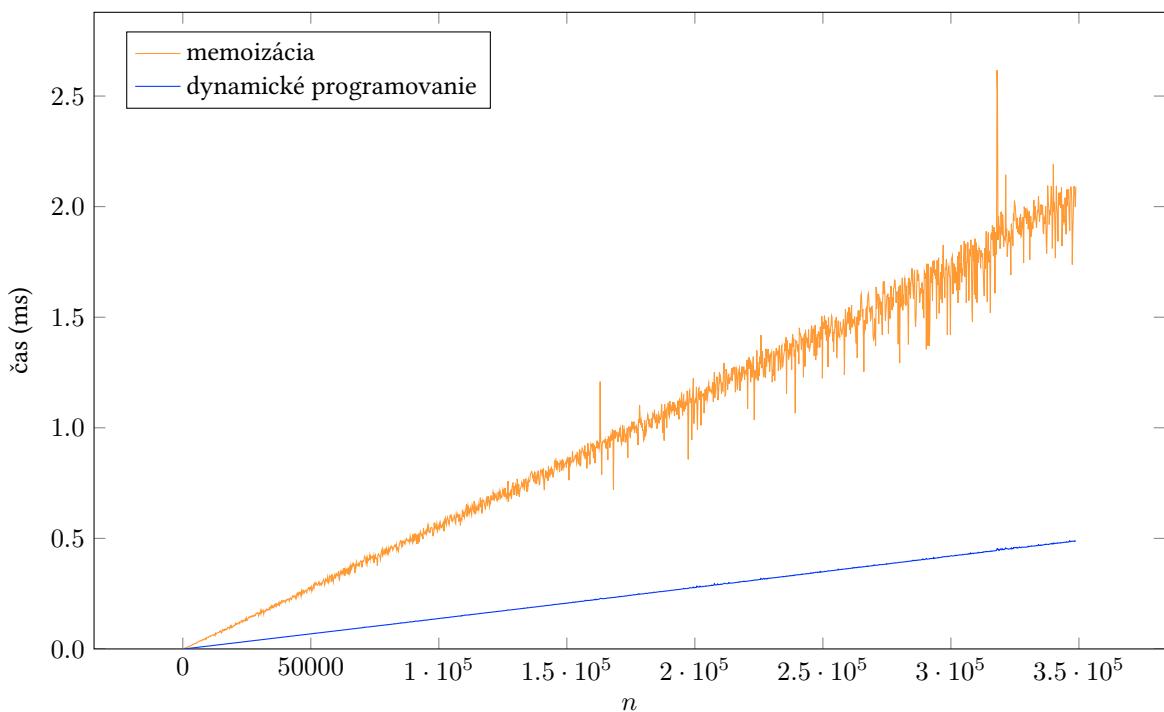
Skúsmo pokračovať v tom, aké veľké vstupy dokáže memoizovaná verzia vyriešiť. Aj pre pomerne veľké vstupy memoizovaná funkcia beží rýchlo, ale pri istej (nie príliš veľkej) veľkosti vstupu program zrazu zhavaruje. Čo sa stalo? Ako si videl v časti 11, pri volaní každom rekurzívnej funkcie sa vyrábí nový svet a ostatné svety čakajú. Pre každý svet musí byť v pamäti vyhradená časť s jeho premennými. Z dôvodov, ktoré tu teraz nebudem rozoberať, sa svety ukladajú do zásobníka, ktorý má fixnú veľkosť. Keď je svetov privela, zásobníku dojde pamäť a program zhavaruje.

Našťastie, v našom príklade sa pretečeniu zásobníka vieme ľahko vyhnúť. Všimni si, že v memoizovanom programe používame pole  $m$  na ukladanie už raz vypočítaných výsledkov; na konci výpočtu pole  $m$  obsahuje všetky priebežné výsledky, t.j. v  $b[i]$  je hodnota  $\text{najdi}(i)$ . V prístupe, ktorý sa volá *dynamické programovanie* sa budem snažiť vyrobiť si rovnaké pole  $m$ , ale bez rekurzívnych volaní. Hodnota  $m[i]$  sa nastaví ako maximum z  $a[i] + \text{najdi}(i+2)$  a  $\text{najdi}(i+1)$ . Ak budem postupovať od konca poľa, bude platíť, že keď počítam  $m[i]$ , tak hodnoty  $m[i+2]$  aj  $m[i+1]$  už mám vypočítané. Preto (všimni si, že som si na koniec poľa pridal zarážku) môžem napísť

```

1 int najdi() {
2     m[n] = 0;
3     m[n - 1] = a[n - 1];
4     for (int i = n - 2; i >= 0; i--) {
5         int x = a[i] + m[i + 2], y = m[i + 1];
6         if (x > y) m[i] = x;
7         else m[i] = y;
8     }
9     return m[0];
10 }
```

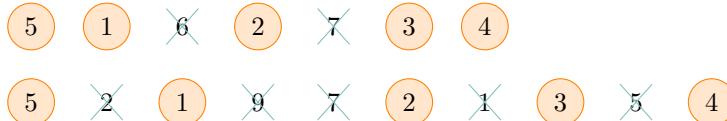
Dynamické programovanie je prístup, v ktorom vypĺňam celú memoizačnú tabuľku bez toho, aby som robil rekurzívne volania. Na to potrebujem násť správne poradie, aby som si bol istý, že hodnoty, ktoré potrebujem na výpočet, už mám vyrobené. Okrem toho, že nepretečie zásobník, získam aj na rýchlosť. Obidve verzie majú lineárnu zložitosť (t.j. časy behu na všetkých vstupoch sa zmestia pod priamku), ale keďže dynamické programovanie nemusí vytvárať a rušiť svety na zásobníku, ušetrí dosť veľa času.



## Memoizácia a dynamické programovanie

**Úloha 52.** Uprav riešenie s dynamickým programovaním tak, aby okrem výsledného súčtu vypísalo aj vybraté čísla.

Skúsme urobiť ďalší príklad. Na vstupe máme dve postupnosti čísel  $a_0, a_1, \dots, a_{n-1}$  a  $b_0, b_1, \dots, b_{m-1}$ . Našim cieľom je čo najmenej z nich vyhodiť tak, aby obidve postupnosti ostali rovnaké. Napr. v nasledovnom vstupe musíme vyhodiť 7 čísel, tri z prvej postupnosti a štyri z druhej.



Prvá vec, ktorú si môžeš všimnúť je, že ak obidve postupnosti začínajú rovnakým číslom, toto číslo sa neopláti ani z jednej postupnosti vyhodiť<sup>3</sup>. Ak teda  $a_0 = b_0$ , stačí nám vyriešiť úlohu pre postupnosti  $a_1, \dots, a_{n-1}$  a  $b_1, \dots, b_{n-1}$ . Čo ak  $a_0 \neq b_0$ ? Tu sa to nedá tak ľahko povedať: sú situácie, keď treba vyhodiť  $a_0$ , inokedy treba vyhodiť  $b_0$  a niekedy aj obidve<sup>4</sup>. Najjednoduchší spôsob, ako zaručiť, že nájdeme správne riešenie, je vyskúsať všetky možnosti, to znamená, že najprv skúsime vyriešiť úlohu pre dvojicu postupností  $a_1, \dots, a_{n-1}$  a  $b_0, \dots, b_{n-1}$ , potom pre  $a_0, \dots, a_{n-1}$  a  $b_1, \dots, b_{n-1}$  a napokon pre  $a_1, \dots, b_{n-1}$  a  $b_1, \dots, b_{n-1}$  a vyberieme tú najlepšiu možnosť. Pretože vždy sa opakuje tá istá úloha, napišeme si rekurzívnu funkciu `diff` s parametrami `i` a `j` tak, že `diff(i, j)` nájde riešenie úlohy pre vstup  $a_i, a_{i+1}, \dots, a_{n-1}$  a  $b_j, b_{j+1}, \dots, b_{n-1}$ . Mohlo by to vyzerať napríklad takto:

```
1 #include <iostream>
2 using namespace std;
3
4 int n[2], a[2][10000];
5
6 int diff(int i, int j) {
7     if (i >= n[0]) return (n[1] - j); // ak je prvá postupnosť prázdna,
8                                     // z druhej treba vyhodiť všetko
9     if (j >= n[1]) return (n[0] - i);
10    if (a[0][i] == a[1][j]) return diff(i + 1, j + 1);
11    int x = 1 + diff(i + 1, j), y = 1 + diff(i, j + 1);
12    if (y < x) x = y;
13    y = 2 + diff(i + 1, j + 1);
14    if (y < x) x = y;
15    return x;
16 }
17
18 int main() {
19     cin >> n[0] >> n[1];
20     for (int i = 0; i < 2; i++)
21         for (int j = 0; j < n[i]; j++) cin >> a[i][j];
22     cout << diff(0, 0) << endl;
23 }
```

Kedže funkcia `diff` nemá vedľajšie efekty, môžeme použiť memoizáciu a už raz vypočítané výsledky si zapamätať: budeme mať dvojrozmerné pole `m` tak, že `m[i][j]` bude mať zapamätaný výsledok volania `diff(i, j)`. Memoizovanú verziu vieme prerobiť na dynamické programovanie tak, že budeme pole `m` vytvárať zaradom celé. Zase si musíme dať pozor na to, aby sme pole prechádzali v takom poradí, že keď počítame `m[i][j]`,

<sup>3</sup>Skús si to premyslieť, nie je to až také zrejmé, ako sa na prvý pohľad zdá. Môže byť totiž viacerých optimálnych riešení a v niektorých z nich sa to prvé číslo vyhodiť môže. Napr. pre 5 4 5 5 a 5 6 5 treba vyhodiť všetko tak, aby v prvej aj druhej postupnosti ostalo 5 5. Takže z druhej postupnosti treba ponechať obidve päťky a z prvej si môžem vybrať, ktoré dve päťky si nechám. Ale ak obidve postupnosti začínajú rovnako, tak určite existuje aj také optimálne riešenie, ktoré prvé číslo v obidvoch ponechá.

<sup>4</sup>skús nájsť príklady pre všetky tri situácie

všetky potrebné hodnoty, t.j.  $m[i][j+1]$ ,  $m[i+1][j]$  a  $m[i+1][j+1]$  už máme vypočítané.

```

1  for (int i = n[0]; i >= 0; i--)
2    for (int j = n[1]; j >= 0; j--)
3      if (i == n[0])
4        m[i][j] = n[1] - j;
5      else if (j == n[1])
6        m[i][j] = n[0] - i;
7      else if (a[0][i] == a[1][j])
8        m[i][j] = m[i + 1][j + 1];
9      else {
10        int x = 1 + m[i + 1][j], y = 1 + m[i][j + 1];
11        if (y < x) x = y;
12        y = 2 + m[i + 1][j + 1];
13        if (y < x) x = y;
14        m[i][j] = x;
15      }
16
17 cout << m[0][0] << endl;

```

Do tretice skúsme takýto príklad. Budeme mať známu hru minesweeper, ale vo veľmi zjednodušenej podobe: na jednorozmernom poli. Budeme mať  $n$  políčok, pričom na niektorých môžu byť bomby. Na políčkach, ktoré susedia s bombou je napísaný počet bômb, s ktorými susedia. Takže herný plán môže vyzerať napr. takto

*	1		1	*	2	*	*	1
---	---	--	---	---	---	---	---	---

Na vstupe budeme mať danú rozohratú pozíciu, v ktorej niektoré políčka sú zakryté. Vstup je daný ako reťazec znakov, kde '\*' znamená bombu, napr.

?	?	?	1	*	2	?	?	1
---	---	---	---	---	---	---	---	---

je **???1\*2??1**. Našou úlohou je zistiť, koľkými spôsobmi je možné doplniť zakryté políčka tak, aby sme mali korektný hrací plán. V našom príklade na predposlednom políčku musí byť bomba (lebo na poslednom je napísaná jednotka), a na siedmom políčku tiež (lebo šieste políčko susedí s dvoma bombami). Na tretom políčku bomba nesmie byť, preto sú štyri možné hracie plány:

			1	*	2	*	*	1
*	1		1	*	2	*	*	1
1	*	1	1	*	2	*	*	1
*	*	1	1	*	2	*	*	1

Priamočiary spôsob, ako to naprogramovať, je rekurzívne skúšať všetky možnosti. Ak je na nejakom políčku  $i$  znak '?', postupne skúsime dosadiť znaky '\*', '0', '1', '2' a rekurzívne vyskúšame všetky možnosti pre políčka  $i+1$  a ďalšie. Vždy, keď prídem na koniec, skontrolujeme, či máme prípustný hrací plán, a ak áno zarátame ho ako jednu z možností:

```

1 #include <iostream>
2 using namespace std;
3
4 char s[10000];
5 char znaky[] = "*012";
6 int n;
7

```

## Memoizácia a dynamické programovanie

---

```
8 int pocet; // tu si pamäťame celkový počet možností
9
10 void skus(int i) {
11     if (i == n) { // ak sme prišli na koniec vstupu, zistíme, či máme správny hrací plán
12         for (int j = 0; j < n; j++) {
13             if (s[j] != '*') { // pre každé poličko, ktoré nie je bomba
14                 int val = s[j] - '0'; // hodnota 0,1,2
15                 int sus = 0; // počet bômb, s ktorými susedí
16                 if (j > 0 && s[j - 1] == '*') sus++;
17                 if (j < n - 1 && s[j + 1] == '*') sus++;
18                 if (val != sus) return; // ak je to nesprávne, nebola to dobrá možnosť,
19                                         // skončíme celú funkciu
20             }
21             pocet++; // ak sme prišli až sem, všetky polička sú korektné, máme novú možnosť
22         return;
23     }
24
25     // v opačnom prípade, kým nie sme na konci vstupu
26     if (s[i] == '?') { // ak tam bol otáznik, skúsime dosadiť všetky možnosti
27         for (int j = 0; j < 4; j++) {
28             s[i] = znaky[j];
29             skus(i + 1); // zakaždým sa rekúrzívne zavoláme
30         }
31         s[i] = '?'; // nakoniec upraceme - rozmysli si, prečo je toto dôležité !
32     } else
33         skus(i + 1); // ak na poličku neboli otáznik, len sa zavoláme ďalej
34     } // koniec funkcie skus
35
36 int main() {
37     cin >> s;
38     n = 0;
39     while (s[n] != 0) n++;
40     cout << n << endl;
41
42     pocet = 0;
43     skus(0);
44     cout << pocet << endl;
45 }
```

Pre malé vstupy to funguje, ale už napr. vstup, v ktorom je 20 otáznikov, trvá pridlho. Kontrolná otázka: môžeme na túto funkciu použiť memoizáciu?

Samozrejme, že nie, pretože modifikuje globálne premenné. Prvý problém je, že výsledok nevracia pri volaní, ale modifikuje globálnu premennú `pocet`. To sa dá napraviť jednoducho: `skus` bude vracať počet možností, ako sa dá doplniť zvyšok od  $i$ -tej pozície. V podmienke `if (i == n)` sa vráti 0 alebo 1 podľa toho, či je pozícia korektná. Pre opačnú podmienku si zrátam dokopy výsledky všetkých rekúrzívnych volaní a tento súčet vrátim. Druhý problém je, že počas volania sa modifikuje pole `s`. Preto výsledok volania `skus(i)` nemôžem memoizovať, lebo závisí od toho, aké je práve pole `s`. Na memoizáciu potrebujem spraviť takú rekúrzívnu funkciu, ktorá by si všetky zmeny posielala v parametroch. Keby som ale ako parameter posielal celé pole `s`, memoizačná tabuľka by bola privellká. Naďastie si stačí uvedomiť, že na to, aby som skontroloval, či je hrací plán v poriadku, nepotrebujem čakať do konca poľa, ale viem robiť kontrolu priebežne. Na to, aby som skontroloval poličko  $i$  mi stačí vedieť hodnoty poličok  $i - 1$  a  $i + 1$ . Lenže keď volám `skus(i)`, tak hodnotu polička  $i + 1$  nepoznám (možno je tam '?'). Vyriešim to napr. takto: budem si ako parameter posielat hodnoty predchádzajúcich dvoch poličok. Volanie `skus(i, a, b)` bude teda znamenať *Zrátaj, kolkými možnosťami môžeme doplniť hrací plán od pozície  $i$  do konca, ak na poličkach  $i - 2$  a  $i - 1$  sú znaky  $a$  a  $b$ .* Teraz pri volaní `skus(i, a, b)` viem skontrolovať, či je poličko  $i - 1$  v poriadku. Celé to môže vyzeráť napr. takto:

```

1 // Máme tri za sebou idúce znaky a,b,c. Je znak b v poriadku?
2 // parameter i je iba na to, aby som nekontroloval zarážku pred vstupom
3 int kontrola(int i, char a, char b, char c) {
4     if (i == 0 || b == '*') return 1; // zarážka alebo bomba je vždy v poriadku
5     int val = b - '0';           // hodnota podľa čísla
6     int sus = 0;                // počet susedov
7     if (a == '*') sus++;
8     if (c == '*') sus++;
9     if (val != sus) return 0;
10    return 1;
11 }
12
13 char znaky[] = "*012";
14
15 int skus(int i, char a, char b) {
16     if (i == n) return kontrola(i, a, b, '0');
17     if (s[i] == '?') {
18         int sum = 0;
19         for (int j = 0; j < 4; j++)
20             sum += kontrola(i, a, b, znaky[j]) * skus(i + 1, b, znaky[j]);
21         return sum;
22     } else
23         return kontrola(i, a, b, s[i]) * skus(i + 1, b, s[i]);
24 }
```

Táto funkcia už nemodifikuje žiadne globálne premenné, takže ju ľahko memoizujem: pridám si pole  $m[10000][4][4]$ , ktoré si inicializujem zarážkou (napr. -1). Vždy, keď zavolám `skus(i,a,b)` najprv skontrolujem, či  $m[i][a][b] > -1$ . Ak áno, iba vrátim uloženú hodnotu. Ak nie, urobím všetko ako doteraz, iba pred skončením aktualizujem hodnotu  $m[i][a][b]$ , aby som ju mal uloženú pre ďalšie volania. Posledný krok je dynamické programovanie: budem postupne prechádzať poľom  $m$  (odzadu) a vyplňať všetky možnosti.

```

1 for (int i = n; i >= 0; i--)
2     for (int a = 0; a < 4; a++)
3         for (int b = 0; b < 4; b++) {
4             if (i == n) {
5                 if (znaky[b] == s[n - 1] || s[n - 1] == '?')
6                     m[i][a][b] = kontrola(i, znaky[a], znaky[b], '0');
7             } else if (s[i] == '?') {
8                 int sum = 0;
9                 for (int j = 0; j < 4; j++)
10                     sum += kontrola(i, znaky[a], znaky[b], znaky[j]) * m[i + 1][b][j];
11                 m[i][a][b] = sum;
12             } else {
13                 int j;
14                 for (j = 0; j < 4; j++)
15                     if (s[i] == znaky[j]) break;
16                 m[i][a][b] = kontrola(i, znaky[a], znaky[b], s[i]) * m[i + 1][b][j];
17             }
18 }
```

**Úloha 53.** Na vstupe je dané  $n$  a potom pole  $n$  celých čísel. Napiš program, ktorý vypíše najdlhšiu (nie nutne súvislú) rastúcu postupnosť, ktorá sa vyskytuje vo vstupnom poli. Napr. pre pole 10 6 9 7 1 8 2 4 3 4 treba vypísat 1 2 3 4.

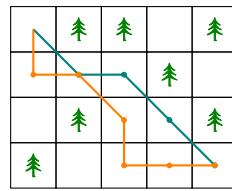
**Úloha 54.** Na vstupe je reťazec znakov. Napiš program, ktorý vypíše najdlhší (nie nutne súvislý) palindróm (reťazec, ktorý je rovnaký spredu aj zozadu), ktorý sa v ňom vyskytuje.

Napr. pre vstup z jedzlesnoplodyvsipilivokneglej je najdlhší palindróm jelenovipivonelej.

**Úloha 55.** Na vstupe je číslo  $n$  a potom  $n$  čísel. Každé číslo je buď  $-1$  alebo celé číslo z rozsahu  $1, \dots, 200$ . Našim cieľom je nahrať všetky  $-1$  číslami z rozsahu  $1, \dots, 200$  tak, aby vo výslednej postupnosti neboli lokálne maximá, t.j. aby každé číslo susedilo aspoň s jedným číslom, ktoré nie menšie. Napíš program, ktorý zistí, kolkými spôsobmi sa to dá urobiť. Napr. pre vstup  $-1 -1$  je 200 možností, lebo obidve čísla musia byť rovnaké. Pre vstup  $100 -1 -1$  je tiež 200 možností: druhé číslo musí byť 100 alebo viac, lebo inak by prvé číslo bolo lokálne maximum. Ak je druhé číslo 100, tretie číslo môže byť 100 alebo čokoľvek menšie. Ak je druhé číslo väčšie ako 100, tretie číslo musí byť rovnaké, lebo inak by druhé číslo bolo lokálne maximum.

**Úloha 56.** Máme danú mapu lesa rozmerov  $m \times n$ , pričom každé poličko je buď voľné, alebo je na ňom strom. Začíname v ľavom hornom rohu a v každom kroku sa môžeme pohnúť o jedno poličko doprava, dole, alebo šikmo. Napíš program, ktorý zistí, kolkými možnosťami to vieme urobiť. Napríklad pre vstup vľavo existuje 21 rôznych ciest, dve z nich sú naznačené na obrázku vpravo.

```
4 5
.##.#
...#.
.#..#
#....
```



**Úloha 57.** Máme batoh s objemom 4200 litrov. Na vstupe je  $n$  vecí, každá z nich má objem a cenu (obidve sú celé čísla, objem každej je nanajvýš 4200). Chceme vybrať niektoré veci tak, aby sa nám zmestili do batohu (t.j. súčet ich objemov bol nanajvýš 4200) a mali čo najväčšiu cenu. Napíš program, ktorý zistí, akú najväčšiu cenu môžeme dosiahnuť. Napríklad pre veci s cenami  $100 100 201$  a objemami  $2000 2000 3000$  najlepšie, čo môžeme urobiť, je zobrať tretiu vec a získať  $201$ .

## Vlastné typy: struct 15

Základné typy, o ktorých sme si hovorili v kapitole 7 majú spoločné to, že zaberajú nejakú fixnú časť pamäte. Okrem základných typov je možnosť vyrobiť si aj vlastné typy. Na to slúži klúčové slovo **struct** (záznam). Vlastný typ, ktorý si vyrobíš v programe, bude pozostávať z niekoľkých častí, z ktorých každá má meno a svoj typ. Napríklad pre prácu s farbami sa často používa RGBA model: farba je pamätaná ako štyri čísla z rozsahu 0...255, ktoré udávajú množstvo červenej, zelenej a modrej zložky. Posledné číslo je tzv. alfa zložka a udáva priesvitnosť: 0 je úplne priesvitná a 255 úplne nepriesvitná.

```
1 struct Farba {  
2     unsigned char r, g, b, a;  
3 };
```

Týmto si vyrobil typ **Farba**. Premenné tohto typu budú v pamäti zaberať 4 byty, pričom každý z nich sa bude správať ako celé číslo bez znamienka (t.j. budú mať hodnoty 0...255). Premennú tohto typu vyrobíš rovnako ako akúkoľvek inú, napr. **Farba f**; vyrobí premennú **f** typu **Farba**. K jej jednotlivým zložkám sa dá pristupovať pomocou operátora **.** (bodka): **f.r** je premenná typu **unsigned char**, ktorá je zložka **r** z premennej **f**. Celú premennú môžeš nastaviť zápisom v kučeravých zátvorkách, napr. **f = {255, 255, 0, 255}**; urobí žltú farbu. Jednu premennú môžeš skopírovať do druhej pomocou priradenia. Priradenie skopíruje celú pamäť, t.j. ak máš premenné **Farba f1, f2**; tak **f1 = f2**; je to isté ako **f1.r = f2.r; f1.g = f2.g; f1.b = f2.b; f1.a = f2.a**; Kedže to ale nie sú čísla, nemôžeš použiť iné operácie, napr. sčítanie, **f1 + f2** je chyba. Program

```
1 struct Farba {  
2     unsigned char r, g, b, a;  
3 };  
4  
5 int main() {  
6     Farba f, h;  
7     f = {100, 100, 255, 255};  
8     h = f;  
9     h.r = 0;  
10    h.b = 80;  
11    cout << (int)h.r << " " << (int)h.g << " " << (int)h.b << endl;  
12 }
```

vypíše **0 100 255**. Všimni si zápis **(int)h.r**. Ten sa volá pretypovanie (*konverzia*). Výraz **(typ)(vyraz)** hovorí *Zober hodnotu výrazu (vyraz) a urob z nej typ (typ)*. V našom prípade je **(vyraz) h.r**, čo je hodnota typu **unsigned char**. Bez konverzie by sa k nej príkaz **cout <<** správal ako k znaku: vypísal by znak s príslušnou ASCII hodnotou. Konverzia spôsobí, že hodnota výrazu sa sice nezmiení, ale **cout <<** vypíše príslušné číslo. Podobne príkaz

```
int i = -1; cout << (unsigned int)i << " " << i << endl;
```

vypíše **4294967295 -1** (v pamäti na adrese premennej **i** je stále uložená tá istá hodnota, 32 jednotiek v dvojkovom zápise).

Jednotlivé zložky typu **struct** môžu byť akékoľvek typy alebo polia, takže môžeš mať napr.

```
1 struct Gradient{  
2     Farba x[2]; // ak je tu pole, musí mať fixnú veľkosť  
3     bool linear;  
4 };
```

## Vlastné typy: `struct`

---

a písat<sup>1</sup>Gradient g = {{f, {100, 100, 100, 255}}, false}; alebo g.x[1].r = g.x[0].b; a pod. V súbore `obrazok.h`<sup>2</sup> je aj jednoduchá podpora na prácu s farebnými obrázkami, takže napr. program

```
1 #include "obrazok.h"
2 #include <iostream>
3 using namespace std;
4
5 struct Farba {
6     unsigned char r, g, b, a;
7 };
8
9 Farba pal[2] = {{0, 0, 255, 255}, {255, 255, 0, 255}};
10
11 int main() {
12     int d = 100;
13     Farba a[8 * d][8 * d];
14     int i, j;
15     for (i = 0; i < 8 * d; i++)
16         for (j = 0; j < 8 * d; j++)
17             a[i][j] = pal[((i / d) + (j / d)) % 2];
18     zapis_rgba_png(8 * d, 8 * d, a, "vytup.png");
19 }
```

vyrobí súbor `vytup.png`, v ktorom bude žlto-modrá šachovnica.

**Úloha 58.** Pozri si stránku [galéria štátnych vlajok<sup>3</sup>](#) a napíš program, ktorý pre zadaný názov štátu vyrobí obrázok s jeho vlajkou. Pre ktoré štáty to vieš urobiť?

---

<sup>1</sup>S takýmto zápisom v kučeravých zátvorkách sa stretнемe ešte veľakrát, hovorí sa mu  *inicializačný zoznam (initializer list)*. Do (takmer) hociakej premennej, ktorá má viac zložiek (napr. pole, `struct`) môžeš priradiť konštantu, ak jednotlivé položky (v poradí, v akom sú definované) vymenuješ v kučeravých zátvorkách.

<sup>2</sup><https://github.com/pocestny/programovanie/raw/master/materialy/obrazok.h>

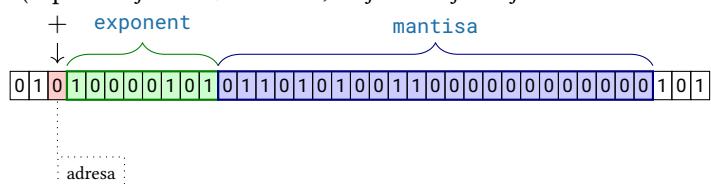
<sup>3</sup>[https://sk.wikipedia.org/wiki/Gal%C3%A9ria\\_%C5%A1t%C3%A1tnych\\_vlajok](https://sk.wikipedia.org/wiki/Gal%C3%A9ria_%C5%A1t%C3%A1tnych_vlajok)

## (Ne)Reálne čísla 16

Naše rozprávanie o základných typoch treba doplniť o ďalšie dva dôležité typy: `float` a `double`. Slúžia na prásu s reálnymi (desatinnými) číslami. V kapitole 7 sme hovorili, že typy `int`, `long`, `unsigned int`, `char` a pod. ukladajú čísla v pamäti v dvojkovej sústave vo fixnom počte bitov. Pri práci s desatinnými číslami je o jeden problém naviac, a to, že idú donekonečna nielen smerom k veľkým číslam, ale sú aj nekonečne husté. Kedže, podobne ako celočíselné typy, aj typy `float` a `double` majú v pamäti rezervovaný fixný počet bitov (32, resp. 64), môže sa stať, že dve veľmi blízke čísla sa zlejú do jedného. Aby sa šetrilo pamäťou, používa sa tzv. vedecká notácia s *mantisou* a *exponentom*. Možno si niekedy videl taký zápis v desiatkovej sústave, napr.  $1.3 \cdot 10^5$ , čo je  $1.3 \cdot 100000 = 130000$ . Podobne  $1.3 \cdot 10^{-5} = 1.3 \cdot 0.00001 = 0.000013$ . Je to veľmi príjemné, keď treba pracovať s veľmi veľkými alebo veľmi malými číslami. V počítači je to podobné, iba sa používa dvojková sústava. V dvojkovej sústave tiež môžeme používať "desatinnú" čiarku, napr. číslo 1011010.10011 v dvojkovej sústave je

64	32	16	8	4	2	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$
:	:	:	:	:	:	:	:	:	:	:	:
1	0	1	1	0	1	0	1	0	0	1	1

$64 + 16 + 8 + 2 + 0.5 + 0.0625 + 0.03125 = 90.59375$  v desiatkovej. Pre zapamätanie čísel sa tieto najprv normalizujú tak, aby "desatinná" čiarka bola hned za prvou jednotkou. Naše číslo 90.59375 by sme preto v dvojkovej sústave zapísali  $1.01101010011 \cdot 2^6$ . Časť `01101010011` sa volá mantisa a `6` je exponent. Typ `float` má vyhradený jeden bit na znamienko, 8 bitov na exponent a 23 bitov na mantisu. K hodnote exponentu sa priráta 127, aby sme mohli mať aj záporné exponenty a nemuseli ukladať zvlášť znamienko. Naše číslo by preto v pamäti vyzeralo takto (exponent je  $127 + 6 = 133$ , čo je v dvojkovej sústave `10000101`):



Drobné technické detaily som zamlčal, ale pre nás to teraz stačí. To, čo je dôležité vedieť je, že v premennej typu `float` sice môžeme mať zapamätané aj veľmi veľké čísla, ale potom nemáme veľkú presnosť. Skús si napríklad takýto program:

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     float a, b;
6     cin >> a;
7     b = a + 1;
8     cout << fixed << a << endl;
9     cout << b << endl;
10    if (a == b) cout << "rovnake" << endl;
11    else cout << "rozne" << endl;
12 }
```

(`fixed` je, podobne ako `endl`, špeciálna hodnota, ktorá spôsobí, že `cout <<` nebude používať vedeckú notáciu). Keď program spustíš a na vstupe napíšeš `12.3`, vypíše sa, presne ako očakávaš,

```

12.300000
13.300000
rozne
```

## (Ne)Reálne čísla

---

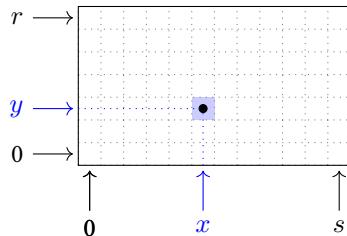
Ale keď na vstupe zadáš **16777216** (čo je  $2^{24}$ ), vypíše sa

```
16777216.000000
16777216.000000
rovnake
```

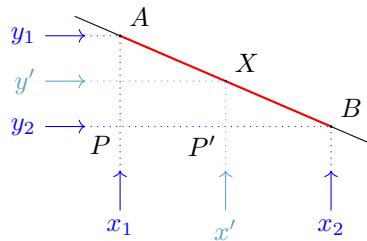
Čo sa stalo? Vstupné (celé) číslo je v dvojkovej sústave jednotka a za ňou 24 nul, zapamätá sa teda ako  $1.0\dots0 \cdot 2^{24}$ , čiže má exponent 151 ( $127 + 24$ ) a mantisu 0. Potom k nemu prirátaš 1 a dostaneš číslo, ktoré má jednotku, potom 23 nul a potom zase jednotku. Opäť by teda exponent bol 151, ale mantisa nemá dosť miest – prvých 23 miest mantisy sú muly a jednotka na 24tom mieste sa stratí. Na podobné efekty si treba dávať pozor, môžu okrem iného spôsobiť, že výsledok niekoľkých operácií môže stratiť presnosť a test na rovnosť dvoch **float** čísel je vždy ošemetný. Pri type **double**, ktorý má 64 bitov (exponent 11 bitov a mantisa 52 bitov) sú efekty straty presnosti menej viditeľné, ale tiež ich treba mať na pamäti. Posledná vec: konverzia (**int**)**a**, kde **a** je typu **float** alebo **double** vráti odrezanú desatinnú časť (t.j. (**int**)**12.3** je **12** a (**int**)**-12.3** je **-12**). Ak sa výsledné číslo nezmestí do **int**, hodnota môže pretiečť, napr. ak máš **float a = 4294967297.0**; tak **cout << (int)a;** vypíše **0**.

## Projekt: offline grafický editor 17

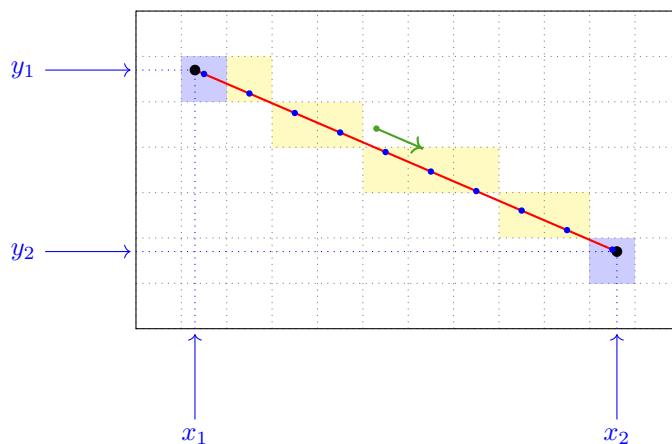
Majme dvojrozmerné pole `obr` s  $r$  riadkami a  $s$  stĺpcami. Bod v rovine so súradnicami  $x, y$  (kde  $x, y$  sú reálne čísla) sa zobrazí na pixel `obr[r-(int)y][(int)x]`:



Majme teraz nasledovnú úlohu. Máme dané dva body  $A = [x_1, y_1]$ ,  $B = [x_2, y_2]$  a chceme vykresliť čiaru z bodu  $A$  do bodu  $B$ . Ak čiara ide zvislo alebo vodorovne, situácia je jednoduchá, stačí nám jeden cyklus. V opačnom prípade máme viac roboty. Zober si priamku, ktorá prechádza bodmi  $A, B$ . Keď sa z bodu  $A$  presunieš do bodu  $B$ , prejdeš vo vodorovnom smere  $x_2 - x_1$  a vo zvislom smere  $y_2 - y_1$ . Ak prejdeš do polovice cesty, dostaneš sa do bodu  $X$ :



Pretože  $\triangle APB$  a  $\triangle X P' B$  sú podobné, tak pri ceste do bodu  $X$  prejdeš polovičnú vzdialosť vo vodorovnom smere, aj polovičnú vzdialosť v zvislom smere, t.j. v smere osi  $x$  sa posunieš o  $0.5(x_2 - x_1)$  a v smere osi  $y$  o  $0.5(y_2 - y_1)$ . Toto platí všeobecne: ak sa na osi  $x$  posunieš o  $c(x_2 - x_1)$ , na osi  $y$  sa posunieš o  $c(y_2 - y_1)$ . Ak si za  $c$  zoberiem  $\frac{1}{x_2 - x_1}$ , tak keď som v nejakom bode na priamke a v smere osi  $x$  sa pohnem o  $c(x_2 - x_1) = 1$ , tak v smere osi  $y$  sa pohnem o  $c(y_2 - y_1) = \frac{y_2 - y_1}{x_2 - x_1}$ . Číslo  $s = \frac{y_2 - y_1}{x_2 - x_1}$  budem volať *sklon* priamky: ak sa v smere  $x$  pohnem na priamke o 1, v smere  $y$  sa pohnem o  $s$ . Predpokladajme, že  $x_1 < x_2$  a  $-1 \leq s \leq 1$ , t.j. ak sa pohnem o 1 v smere osi  $x$ , v smere osi  $y$  sa pohnem o menej ako 1.



## Projekt: offline grafický editor

---

Na nakreslenie čiary stačí vyrátať  $s$ , pohnúť sa na priamke tak, aby som bol v smere  $x$ -ovej osi v strede počiatočného pixelu (prvá modrá bodka, súradnice<sup>1</sup>  $\lfloor x_1 \rfloor + 0.5$  a  $y_1 + s(\lfloor x_1 \rfloor + 0.5 - x_1)$ ) a potom postupne sa vždy pohnúť o 1 v smere osi  $x$  a o  $s$  v smere osi  $y$  a zafarbiť príslušný pixel (na obrázku žlté).

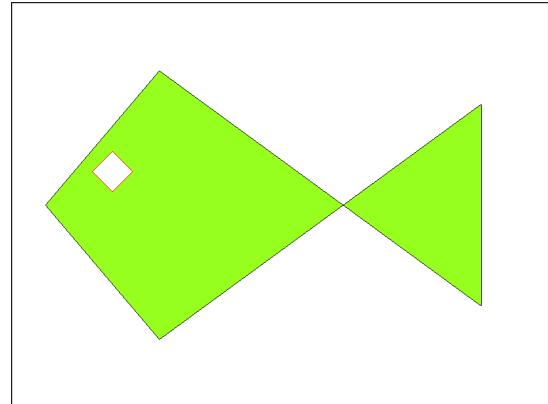
Ak by naopak  $s < -1$  alebo  $s > 1$ , to znamená, že ak sa pohnem o 1 v smere osi  $x$ , v smere osi  $y$  sa môžem pohnúť o veľa. Preto keby som použil tento istý program, môže sa mi stať, že niektoré pixely preskočím. Preto to treba urobiť naopak: posunúť sa vždy o 1 v smere osi  $y$  a dorátať si pozíciu v smere osi  $x$ . Skús si to premyslieť a naprogramovať.

**Úloha 59.** Napiš program, ktorý prečíta postupnosť príkazov (na každom riadku jeden) a podľa nich vyrobí obrázok. Príkazy môžu byť

- **i sirka vyska (init)** Toto je vždy prvý príkaz (a je v zozname iba raz). Nastaví rozmery obrázka.
- **c r g b (color)** Nastaví farbu na  $\{r, g, b, 255\}$ .
- **m x y (move)** Presunie sa do bodu  $[x, y]$  (bez kresenia).
- **l x y (line)** Nakreslí čiaru (aktuálou farbou) z aktuálneho bodu do bodu  $[x, y]$ .
- **f x y r g b (fill)** Vyplní aktuálnou farbou oblasť, ktorej okraje sú ohraničené pixelmi bud' aktuálnej farby alebo farby  $\{r, g, b, 255\}$  (spomeň si na úlohu 41)
- **s meno (save)** Uloží obrázok a skončí.

Napríklad tento vstup vyrobí obrázok vpravo

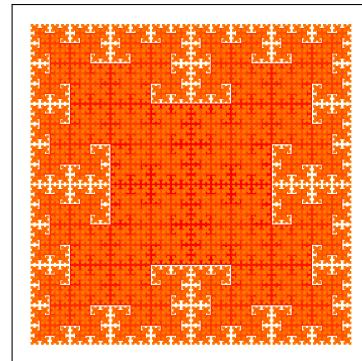
```
i 800 600
m 50 300
l 220 100
l 700 450
l 700 150
l 220 500
l 50 300
c 150 255 30
f 220 220 0 0 0
f 650 300 0 0 0
c 255 0 0
m 120 350
l 150 380
l 180 350
l 150 320
l 120 350
c 255 255 255
f 130 340 255 0 0
s edit.png
```



<sup>1</sup>v matematike sa zvykne písat  $\lfloor x \rfloor$  na označenie dolnej celej časti, t.j. najbližšieho menšieho celého čísla. Treba dať pozor na to, že  $\lfloor x \rfloor = (\text{int})x$  iba pre kladné čísla, napr.  $(\text{int})3.14 == 3$ . Pre záporné čísla to ale neplatí:  $\lfloor -3.14 \rfloor = -4$ , ale  $(\text{int})(-3.14) == -3$ .

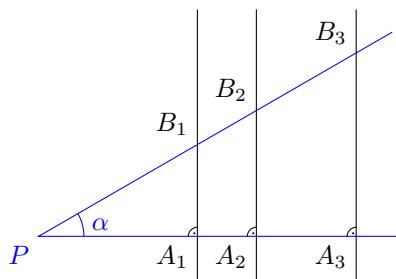
**Úloha 60.** Uvažujme nasledovnú variáciu na tému Sierpiňského koberca: máme parametre `int x`, `int d`, `double i` a `double f`. Chceme mať vzorku na obrázku rozmerov  $x \times x$ . Vzorka stupňa  $d$  je štvorec so stranou dĺžky  $i$  umiestnený v strede obrázka. Navyše, ak  $d > 1$  tak v každom z rohov štvorca umiestníme vzorku stupňa  $d - 1$  zmenšenú o faktor  $f$  (a môže mať upravenú farbu). Napríklad pre hodnoty `1000 6 0.45 0.5` by vzorka mohla vyzerat ako na obrázku vpravo.

Napiš program, ktorý načíta zo vstupu parametre `x d i f` a vypíše po- stupnosť príkazov, na základe ktorej program z predchádzajúcej úlohy vykreslí vzorku.



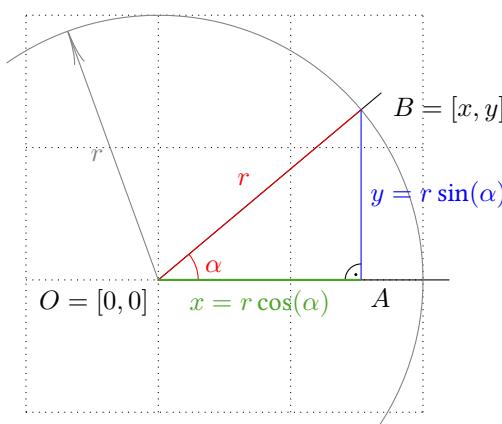
### Odbočka: goniometrické funkcie sin a cos

Niekedy je dobré vedieť kresliť čiaru nie do konkrétneho bodu, ale pod istým uhlom. Povedzme, že máme uhol  $\alpha$  okolo bodu  $P$ . Urobíme si hocikolko kolmíc takto:



Všetky trojuholníky  $\triangle A_1PB_1, \triangle A_2PB_2, \dots$  sú podobné: majú všetky uhly rovnako veľké. Preto sa zachovávajú aj pomery strán: napr.<sup>2</sup>  $\|A_3P\| = 2\|A_1P\|$ , preto aj  $\|B_3P\| = 2\|B_1P\|$ . To znamená, že  $\frac{\|A_3P\|}{\|B_3P\|} = \frac{2\|A_1P\|}{2\|B_1P\|} = \frac{\|A_1P\|}{\|B_1P\|}$ . Inými slovami, pomer  $\frac{\|A_iP\|}{\|B_iP\|}$  je stále rovnaký, nezáleží, ktorú kolmicu  $A_iB_i$  si zoberieme. Tento pomer závisí iba od uhlia  $\alpha$ , a označuje sa  $\cos(\alpha)$  (*kosínus*). Podobne pomer  $\frac{\|B_iA_i\|}{\|B_iP\|}$  závisí iba od  $\alpha$  a označuje sa  $\sin(\alpha)$  (*sínus*).

Na čo je dobré poznať hodnoty  $\sin(\cdot)$  a  $\cos(\cdot)$ ? Povedzme, že chceš zistiť, aké súradnice má bod  $B$ , ktorý je od  $[0, 0]$  vo vzdialosti  $r$  pod uhlom  $\alpha$ :



Keď si z  $B$  spravíš kolmicu, dostaneš pravouhlý  $\triangle AOB$ , pričom  $\|OA\| = x$ ,  $\|AB\| = y$  sú hľadané súradnice bodu  $B$ . Z predchádzajúceho vieš, že  $\cos(\alpha) = \frac{\|OA\|}{\|OB\|} = \frac{x}{r}$ , preto  $x = r \cos(\alpha)$ . Rovnako dostaneš  $y = r \sin(\alpha)$ .

<sup>2</sup>znakom  $\|\cdot\|$  označujem dĺžku

## Projekt: offline grafický editor

Funkcie `sin` a `cos` sú prístupné v knižnici, ktorá sa volá `cmath`. Ich vstup ale nie je v stupňoch, ale v radiánoch. Ak obsah kruhu<sup>3</sup> s polomerom 1 označíme  $\pi$ , tak jeho obvod je  $2\pi$ . Radiány merajú uhol dĺžkou príslušného oblúka na jednotkovej kružnici: celý kruh, čiže  $360^\circ$  je  $2\pi$  radiánov, pravý uhol je  $\pi/4$  radiánov atď. Vo všeobecnosti  $d$  stupňov je  $d \frac{2\pi}{360}$  radiánov. V knižnici `cmath` je definovaná konštantá `M_PI`, ktorá dosť presne reprezentuje číslo  $\pi$ . Nasledovný program:

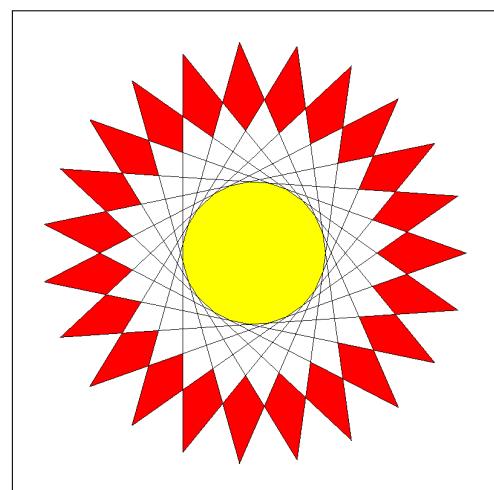
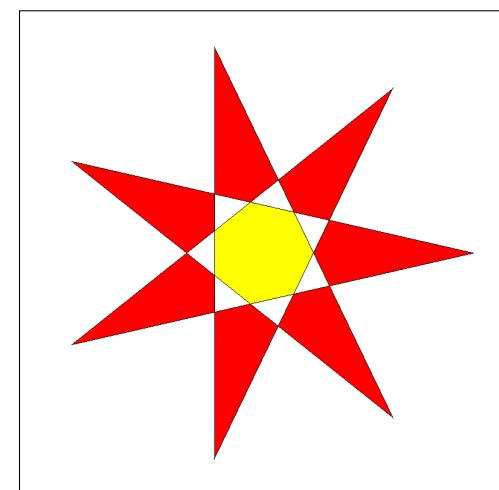
```

1 #include <cmath>
2 #include <iostream>
3
4 using namespace std;
5
6 int main() {
7     cout << 3*cos(M_PI / 2.0) << " " << 3*sin(M_PI / 2.0) << endl;
8 }
```

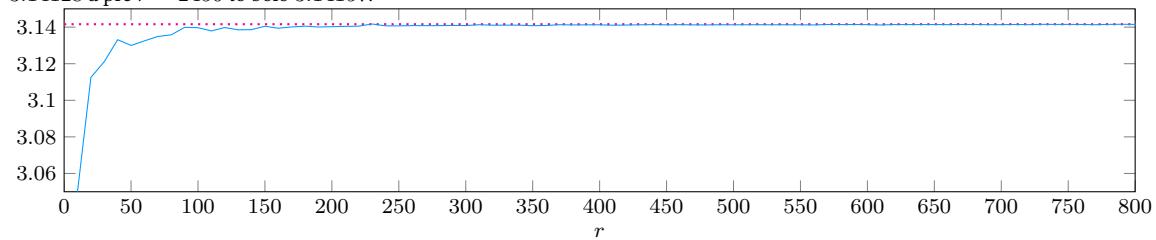
má vypísat súradnice bodu, ktorý je vo vzdialosti 3 pod uhlom  $\pi/2$  (t.j. kolmo hore), teda by sa malo vypísat  $0 \ 3$ . V skutočnosti sa vypíše čosi ako `1.83697e-16 3`. Zápis `1.83697e-16` znamená  $1.83697 \cdot 10^{-16}$ , t.j. číslo  $0.000000000000000183697$ . To je skoro nula, takže je to skoro správne. Nie je to presne nula, lebo sa pri výpočte `cos` prejavili problémy s presnosťou, ktoré si videl v predchádzajúcej časti.

**Úloha 61.** Napiš program, ktorý na vstupe dostane číslo  $n$  a vypíše postupnosť príkazov, na základe ktorých program z úlohy 59 vykreslí vyplnený zelený pravidelný  $n$ -uholník.

**Úloha 62.** Napiš program, ktorý na vstupe dostane číslo  $n$  a  $s$  a vytvorí príkazy pre program z úlohy 59, ktorá nakreslia nasledovný útvar: zober si vrcholy pravidelného  $n$ -uholníka a spájaj ich čiarou s krokom  $s$ , t.j. spoj vrchol číslo 0, vrchol číslo  $s$ , vrchol číslo  $2s$  a tak ďalej. Vnútro má byť vyfarbené žlté a vonkajšie cípy červeno. Napr. pre  $n = 7$ ,  $s = 3$  a pre  $n = 23$ ,  $s = 9$  sú tieto dva vzory:



<sup>3</sup>Urob si takýto pokus: zober si riešenie úlohy 28 a zráťaj v premennej `S` počet čiernych pixelov. Na konci programu vypíš pomer čiernych pixelov k obsahu štvorca so stranou  $r$ , t.j. `(double)S / (double)(r * r)` Pre  $r = 10$  bol môj výsledok 3.05, pre  $r = 500$  to už bolo 3.14128 a pre  $r = 2450$  to bolo 3.14157.



Výsledok sa blíži k číslu  $3.1415926535 \dots$ , čo je hodnota  $\pi$ .

**Úloha 63.** Korytnačia grafika pozostáva z postupnosti príkazov, každý na samostatnom riadku. Príkazy sú takéto:

- **i** *sirka vyska* (init) Toto je vždy prvý príkaz (a je v zozname iba raz). Nastaví rozmery obrázka.
- **c r g b** (color) Nastaví farbu na {r, g, b, 255}.
- **u** (pen up) Zdvihne pero – nasledujúce posuny nekreslia.
- **d** (pen down) Položí pero – nasledujúce posuny kreslia.
- **f x** (forward) Posunie sa dopredu o x.
- **l a** (left) Otočí sa doľava o a stupňov.
- **r a** (right) Otočí sa doprava o a stupňov.
- **s meno** (save) Uloží obrázok a skončí.

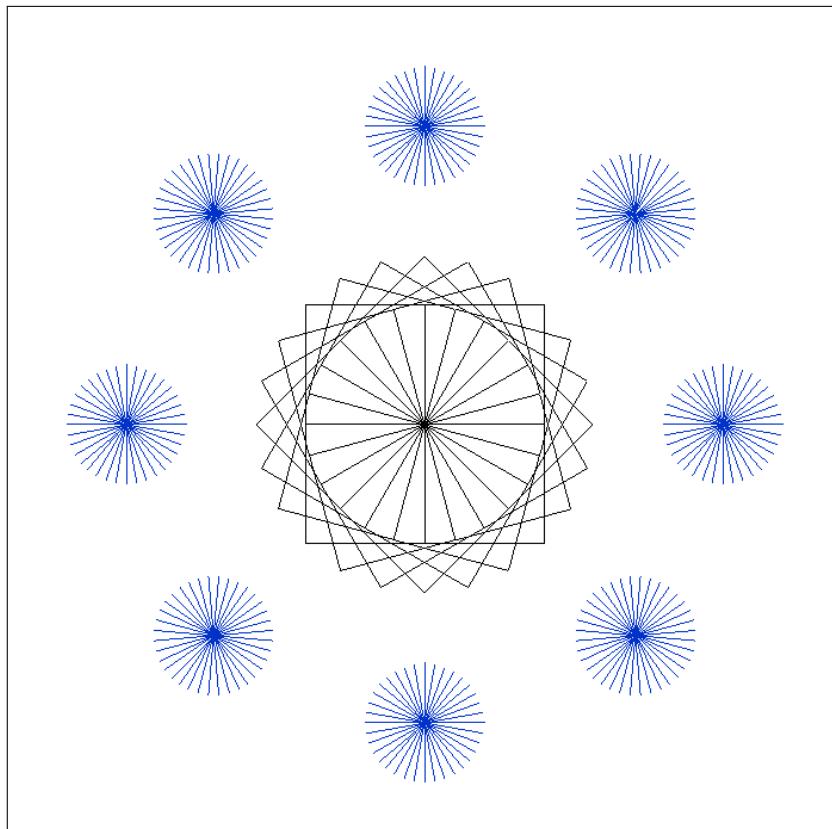
Napíš program, ktorý načíta postupnosť príkazov korytnačej grafiky a vypíše postupnosť príkazov pre program z úlohy 59.

Na záver tejto časti skús rozšíriť korytnačiu grafiku o príkaz opakovania:

- **R cnt** (repeat) Blok príkazov po nasledujúci E sa zopakuje cnt krát.
- **E** (end) Koniec bloku opakovania.

Chceme, aby sa príkazy opakovania mohli vnárať, takže napr. program vľavo urobí obrázok vpravo.

```
i 700 700
R 24
R 4
f 100
r 90
E
l 15
E
c 0 50 200
R 8
u
f 250
d
R 36
f 50
l 180
f 50
l 10
E
u
l 180
f 250
l 225
E
s logo.png
```



Na rozdiel od jednoduchšieho zadania bez cyklov, toto nemôžme riešiť tak, že príkazy čítame zo vstupu a priamo vypisujeme príkazy pre kreslič, ale budeme si potrebovať všetky príkazy zapamätať. Spravme si typ

## Projekt: offline grafický editor

**Prikaz**, v ktorom si budeme pamätať typ príkazu aj všetky potrebné parametre (príkazy **i** a **s** ukladať nepotrebujeme, **i** je vždy na začiatku a **s** je na konci: keď načítame zo vstupu **s**, tak vygenerujeme všetky zapamätané príkazy a skončíme). Okrem parametrov si v príkaze **R** budeme potrebovať pamätať terajší počet opakovaní pri vykonávaní a v príkaze **E** si budeme potrebovať zapamätať pozíciu, kde je príslušný príkaz **R**. Celý typ by mohol vyzeráť napr. takto:

```
1 struct Prikaz {
2     char t;          // typ príkazu
3     int r, g, b;    // pre príkaz 'c'
4     double x;        // generický parameter pre príkazy 'l' 'r' 'f'
5     int i, cnt;      // pre príkaz 'R'
6     int addr;        // pre príkaz 'E'
7 }
```

Pri načítavaní vstupu si ukladáme príkazy do pamäte, pričom na príkazy **R** a **E** použijeme zásobník ako v úlohe 50 na to, aby sme pre každý príkaz **E** našli zodpovedajúci príkaz **R**. Začiatok programu z príkladu by v poli vyzeral takto:

0	1	2	3	4	5	6	7	8	9
t 'R'	t 'R'	t 'P'	t 'r'	t 'E'	t 'P'	t 'E'	t 'c'	t 'R'	t 'u'
...	...	...	...	...	...	...	...	...	...
cnt 24	cnt 4	cnt	cnt	cnt	cnt	cnt	cnt 8	cnt	cnt
i 0	i 0	i	i	i	i	i	i 0	i	i
addr	addr	addr	addr	addr 1	addr	addr 0	addr	addr	addr

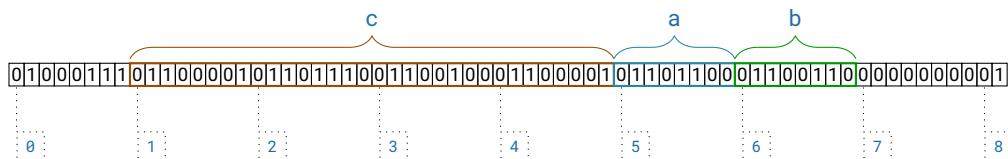
Ked' načítame celý program, začneme ho vyhodnocovať.. Budeme mať jednu globálnu premennú **pc** (program counter), v ktorej si budeme pamätať pozíciu práve vykonávaného príkazu. Bežné príkazy vždy spracujeme (tak, že vypíšeme na výstup príslušný príkaz pre kreslič) a zvýšime **pc**. V príkaze **R** len zvýšime hodnotu **i**. V príkaze **E** sa pozrieme na hodnotu **i** v príslušnom **addr**: ak sa rovná **cnt**, vynulujeme ju a zvýšime **pc**, ak nie, skočíme na príkaz **R**. Ak sa pole príkazov volá **prog** a v premennej **n** máme počet príkazov, vyhodnocovanie by vyzeralo zhruba takto:

```
1 void execute() {
2     int pc = 0;
3     while (pc < n) {
4         if (prog[pc].t == 'R') {
5             prog[pc].i++;
6             pc++;
7         } else if (prog[pc].t == 'E') {
8             int jmp = prog[pc].addr;
9             if (prog[jmp].i == prog[jmp].cnt) {
10                 prog[jmp].i = 0;
11                 pc++;
12             } else {
13                 pc = jmp;
14             }
15         } else if ....
16
17         // tu sa spracujú ďalšie príkazy
18
19     }
20 }
```

**Úloha 64.** Naprogramuj korytnačiu grafiku s príkazmi cyklu.

## Adresy a smerníky (pointre) 18

Technicky je celá pamäť počítača jedno veľké pole nul a jednotiek. V skutočnosti má adresu iba každý ôsmy bit (celá pamäť teda vyzerá ako pole `unsigned char`). Keby na adrese 1 bola premenná `int c`; na adrese 5 premenná `char a` a na adrese 6 premenná `char b`; začiatok pamäte by mohol vyzerať takto:



To, že nejaká premenná je na nejakej adrese, nie je v pamäti nijak špeciálne označené, je na programe, aby pristupoval na správne miesta v pamäti. V pamäti je vždy okrem tvojho programu veľa ďalších vecí, takže keď sa program spustí, operačný systém nájde v pamäti voľné miesto a tvoj program bude používať adresy odtiaľ. Presná adresa premennej ti preto sama osebe veľa nepovie, napriek tomu je užitočné ju vedieť. Až tak užitočné, že na to je špeciálny operátor, `&`. Ak máš premennú `x`, tak `&x` je jej adresa. Skús si spustiť program:

```
1 #include <iostream>
2 using namespace std;
3
4 int c;
5 char a,b;
6
7 int main() {
8     cout << "adresa c:" << (unsigned long)&c << endl;
9     cout << "adresa a:" << (unsigned long)&a << endl;
10    cout << "adresa b:" << (unsigned long)&b << endl;
11 }
```

Vždy, keď ho spustíš, vypíšu sa iné čísla (tvoj program vždy dostane pridelenú inú časť pamäte), ale môže to vyzerať napr. takto:

```
adresa c: 94082916049300
adresa a: 94082916049304
adresa b: 94082916049305
```

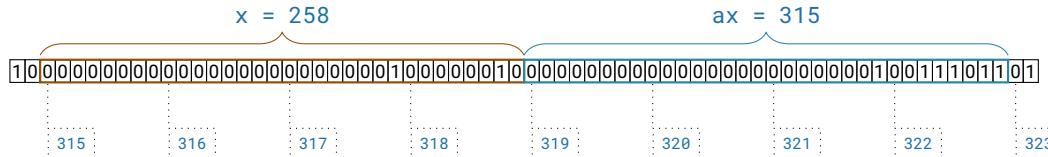
V tomto prípade sa premenná `c` uložila na adresu 94082916049300 a zaberá 4 byty (32 bitov). Za ňou, na adrese 94082916049304 nasleduje premenná `a`, ktorá zaberá jeden byte, takže na nasledujúcej adrese 94082916049305 je premenná `b`.

Napriek tomu, že adresa je vždy číslo, operátor `&` vracia v skutočnosti rôzne typy. Pre každý typ `T` (základný alebo vlastný napr. `int`, `float`, Farba,...) existuje ďalší typ `adresa premennej typu T` (hovorí sa aj *pointer na premennú typu T*). Tento nový typ sa označuje hviezdičkou za menom typu. Takže premenná typu `int*` obsahuje adresu premennej, ktorá je typu `int`:

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int x = 258;      // premenná
5     int *ax = &x;      // ok, ax je adresa x
6     float *err = &x;   // <- !! error: cannot convert 'int*' to 'float*'
7     float* fx = (float*)(&x); // ok, pretypovať sa dá
8 }
```

## Adresy a smerníky (pointre)

V poslednom riadku som použil pretypovanie; `fx` je adresa miesta, kde by mala byť premenná typu `float`, aj keď v skutočnosti je tam uložená premenná typu `int` (konkrétnie `x`). Čo je však záber? premenná `ax` je krabička, v ktorej je uložené číslo adresy<sup>1</sup>, na ktorej je uložená krabička s menom `x`:



Opačný operátor `k &` je `*`: ak `ax` je premenná typu `int*`, tak `*ax` je premenná, ktorá je uložená na adresu, ktorá je uložená v `ax` (hovoríme, že premenná `ax` *ukazuje na premennú `*ax`*). Teraz je vidno, prečo je dobré mať zvlášť typ adresy pre každý typ: keď sa vyhodnocuje výraz `*ax`, procesor sa pozrie do premennej `ax`, nájde tam číslo (napr. 315), pozrie sa na adresu 315 a predpokladá, že tam bude premenná typu `int`. Pozrie sa preto na nasledujúce 4 byty a vráti príslušné číslo. V predchádzajúcom príklade by `cout << *ax << endl`; vypísal 258. `*ax` je naozaj premenná, takže sa dá do nej aj priradoval. Príkazy `x = 47`; a `(*ax) = 47`; urobia to isté. Takže program

```
1 #include <iostream>
2 using namespace std;
3 int main() {
4     int x = 258;
5     int* ax = &x; // ax ukazuje na x
6     (*ax) = 47; // *ax je x
7     cout << x << endl;
8 }
```

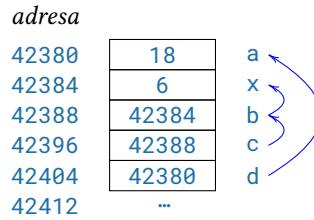
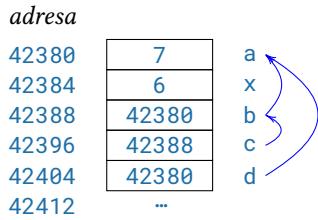
vypíše 47.

Treba rozlišovať medzi hviezdičkou pri vyrábaní premennej (napr. `int *a`), ktorá znamená *pointer na typ int* a hviezdičkou pred menom premennej (typu pointer) vo výraze, ktorá znamená *hodnota na adresu*. Skús prečítať nasledovný program:

```
1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int a = 7, x = 6; // a,x sú premenné typu int
6     int *b = &a; // b ukazuje na a
7     int **c = &b; // c ukazuje na b
8     int *d = *c; // *c je premenná b, preto tento príkaz
9                     // je rovnaký ako int* d = b;
10    **c = 18;
11    b = &x; // teraz b ukazuje na x
12    **c = 42;
13    cout << a << " " << x << endl;
14 }
```

Po prvých štyroch riadkoch by to v pamäti mohlo vyzerať ako na obrázku vľavo. V príkaze `**c=18`; je `*c` premenná `b` a `*b` je premenná `a`, preto sa do premennej `a` zapíše 18. Nasledujúci príkaz `b=&x` spôsobí, že v `b` bude uložená adresa `x` ako na obrázku vpravo. Preto v príkaze `**c=42` je `*c` premenná `b` a `*b` je premenná `x`, takže program vypíše 18 42.

<sup>1</sup>V nasledujúcom obrázku sa zámerne trochu klamem, v premennej typu `int` sa väčšinou jej 4 byty ukladajú v opačnom poradí (tzv. *little endian*), ale pre naše účely to stačí takto.



S pomocou pointrov môžeš mať funkciu, ktorá ako keby menila svoje parametre. Porovnaj tieto dva programy:

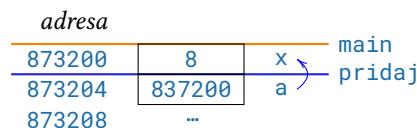
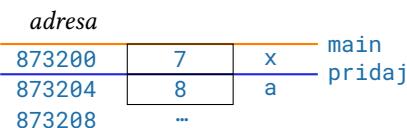
```

1 #include <iostream>
2 using namespace std;
3
4 void pridaj(int a) {
5     a = a + 1;
6 }
7
8 int main() {
9     int x = 7;
10    pridaj(x);
11    cout << x << endl;
12 }
```

```

1 #include <iostream>
2 using namespace std;
3
4 void pridaj(int *a) {
5     (*a) = (*a) + 1;
6 }
7
8 int main() {
9     int x = 7;
10    pridaj(&x);
11    cout << x << endl;
12 }
```

V ľavom programe sa vytvorí svet funkcie `main` a v ňom premenná `x`. Pri volaní funkcie `pridaj` sa vytvorí nový svet s premennou `a` a nastaví sa jej hodnota na 7. Potom sa vykoná funkcia `pridaj`, takže v premennej `a` bude 8. Svet funkcie `pridaj` potom zanikne a v `main` sa vypíše hodnota `x`, t.j. 7. V pravom programe sa vytvorí svet funkcie `pridaj`, ale premenná `a` sa nastaví na adresu premennej `x` (premená `x` je vo svete funkcie `main`, ale svety a premenné sú vec programu a komplilátora; pri prístupe do pamäte vo výslednej binárke nehrajú rolu). Preto vo svete `pridaj` výraz `*a` označuje premennú `x` zo sveta funkcie `main` a funkcia `pridaj` zvýší hodnotu `x` o 1. Keď svet `pridaj` zanikne, zanikne premenná `a`, ale hodnota `x` ostane zmenená.



S pointrami sa dajú robiť aj aritmetické operácie, ale trochu inak ako s normálnymi číslami. Ak k premennej typu pointer pripočítas 1, zväčší sa o veľkosť typu, na ktorý ukazuje (v prípade `int` je to 4). Veľkosť typu sa dá zistiť pomocou príkazu `sizeof`. V poli sú prvky uložené za sebou, takže napr. program

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int a = {12, 35}; // a[0] a a[1] nasledujú v pamäti za sebou
6     int *x = &(a[0]);
7     cout << *x << " " << *(x + 1) << endl;
8     cout << (unsigned long)x << " " << sizeof(int) << " " <<
9         (unsigned long)(x + 1) << endl;
10 }
```

mi vypísal

```
12 35
140727139715200 + 4 = 140727139715204
```

## Adresy a smerníky (pointre)

---

Pri práci s pointrami si treba dávať pozor na to, aby vždy ukazovali do pamäte, ktorá patrí tvojmu programu. Môžeš napríklad napísat `int *a = (int*)856`; a nič zlé sa nestane. Ale ak by si potom chcel urobiť `*a = 3`; program skončí s hláškou **Segmentation fault**: pamäť na adresu 856 ti nepatrí. Je dobrý zvyk do pointrov, ktoré zatial neukazujú na nič rozumné, priradiť 0 (adresa 0 ti celkom isto nepatrí) a pred použitím urobiť test. Môžeš napísat `int *a = 0`; alebo `int *a = NULL`; alebo `int *a = nullptr`; všetky tri spravia v konečnom dôsledku to isté.

Samozrejme, pointer môže ukazovať aj na vlastné typy a dokonca aj na polia. To môže byť trochu zamotané. Napr.

```
1 int a, b, c;
2 int *p[3] = {&a, &b, &c};
3
4 *(p[0]) = 5;
5 *(p[1]) = 25;
6 *(p[2]) = 225;
7
8 cout << a << " " << b << " " << c << endl;
```

vyrobí trojprvkové pole `p`, ktorého prvky budú pointre na premenné typu `int` a naplní ho pointrami na premenné `a`, `b`, `c`. Preto potom `p[0]` je pointer na premennú `a` a teda `*(p[0]) = 5`; priradí 5 do premennej `a` a program vypíše `5 25 225`. Zápis `int *p[3]` čítaš najprv od názvu premennej doprava a potom doľava zvyšok: *pole dĺžky 3, ktorého prvky sú pointre na int*. Ak napišeš `int (*p)[3]`, znemená to *pointer na pole dĺžky 3, ktorého prvky sú int*:

```
1 int a[3] = {1, 2, 3};
2 int(*p)[3] = &a;
3
4 cout << (*p)[1] << endl;
```

Ako by si zapísal pole dĺžky 2, ktorého prvky sú polia dĺžky 3? `int a[2][3]` Pripomína ti to niečo? Presne tak, je to dvojrozmerné pole. Dvojrozmerné pole je vlastne pole riadkov, ktorého prvky sú polia stĺpcov. Preto môžeš mať

```
1 int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
2 int(*p)[3] = &(a[0]);
3
4 cout << (*(p+1))[1] << endl;
```

V tomto prípade `a` je pole dĺžky 2, ktorého prvky sú trojprvkové polia, preto `a[0]` je trojprvkové pole. `p` je pointer na trojprvkové pole, preto môžeš napísat `p=&(a[0])`. Potom `p+1` ukazuje do pamäte o 12 bytov ďalej (`sizeof(int[3])` je 12). Prvky v poli sú uložené za sebou, preto ak `p` ukazuje na `a[0]`, tak `p+1` ukazuje na `a[1]`. Napokon `*(p+1)` je to isté, čo `a[1]`, preto `(*(p+1))[1]` vypíše 5.

Hovoril som ti, že pole nie je premenná, ale veľa premenných, a preto sa správa zvláštne (napr. si nesmel posielat pole ako parameter do funkcie, nesmel si priraďovať a pod.). V skutočnosti je pole konštantný<sup>1</sup> pointer. Keď napišeš `int a[10]`; vytvára sa v pamäti 10 premenných typu `int` a `a` je `const` pointer na prvú z nich. Označenie indexu `[ ]` je v skutočnosti iba skratka za výraz s hviezdičkou: `a[i]` znamená `*(a+i)` pre hocjaký pointer `a`. Rôzne spôsoby prístupu k prvkom poľa sú v nasledujúcom programe:

```

1 #include <iostream>
2 using namespace std;
3
4 int main() {
5     int a[10];
6     int *b, *c;
7     int i;
8     a = b; // <- !! chyba, a je const
9     c = a + 1; // c ukazuje na a[1]
10
11    b = a; // b ukazuje na a[0]
12    for (i = 0; i < 10; i++) {
13        *b = i + 1; // *b zapisuje do pola a
14        b = b + 1; // presuň b na ďalší prvok pola
15    }
16
17    cout << a[0] << endl;
18    for (i = 0; i < 9; i++)
19        cout << c[i] << endl; // c ukazuje na a[1], preto
20                           // c[i] je a[i+1]
21
22    *(a + 5) = 14;           // *(a+5) je a[5]
23    for (b = a; b < a + 10; b++) // k b sa dá pripočítavať číslo aj takto
24        cout << *b << " ";
25    cout << endl;
26 }
```

Teraz by malo byť jasné, ako sa dá posielat pole ako parameter do funkcie: stačí, aby bol typu pointer. Treba si ale vždy poslať aj veľkosť poľa ako samostatný parameter:

```

1 void napln_pole(int n, int *a) {
2     int i;
3     for (i = 0; i < n; i++)
4         a[i] = i * i;
5 }
6
7 int main() {
8     int a[10];
9     napln_pole(10, a);
10    int i;
11    for (i = 0; i < 10; i++)
12        cout << a[i] << " ";
13    cout << endl;
14 }
```

<sup>1</sup>S označením `const` sme sa už stretli v tvare `const int n = 100;`: ak ho napišeš pred definíciu premennej, znamená to, že premenná sa už potom nesmie meniť. Podobne môže byť `const int *a = &x;`

## Polia revealed

---

Tento program vypíše **0 1 4 9 16 25 36 49 64 81**. Len pripomieniem: ak by si v tomto programe zmenil riadok `int a[10];` na `int a[3];`, program by fungoval rovnako, akurát, že vo funkcií `napln_pole` by sa zapisovalo aj do pamäte, ktorá nebola rezervovaná poľom `a`. Ak sú tam nejaké premenné tvojho programu (napr. `i`), prepíšu sa. Ak tá pamäť tvojmu programu nepatrí, pride násť priateľ `segfault`.

Dvojrozmerné pole je pole, ktorého prvkami sú polia. Preto ak máš `int a[2][3]`, tak `a` je (konštantný) pointer na trojprvkové pole typu `int`. Preto `*(a+1)` je to isté, čo `a[1]`, čiže trojprvkové pole typu `int` (t.j. môžeš napísať `int *q = *(a+1);` a potom `q` bude ukazovať na prvý prvok poľa `a[1]`, čiže na druhý riadok poľa `a`). Z toho celého vyplýva, že dvojrozmerné pole je v pamäti vlastne jednorozmerné pole uložené rovnako, ako sme to mali v kapitole 8. Na to, aby si k nemu tak mohol pristupovať, treba ho pretypovať, takže môžeš napísať `int *p = (int *)a;`. To je zároveň aj najjednoduchší spôsob, ako posielat dvojrozmerné pole ako parameter do funkcie.

```
1 int main() {
2     int a[2][3];
3     int i, j, k = 1;
4
5     for (i = 0; i < 2; i++)      // naplníme pole po riadkoch
6         for (j = 0; j < 3; j++) {
7             a[i][j] = k;
8             k++;
9         }
10
11    int *q = *(a + 1);          // q ukazuje na druhý riadok
12    for (int i = 0; i < 3; i++)
13        cout << q[i] << endl;
14
15    int *p = (int *)a;           // pohľad na a ako na 1D pole
16    for (i = 0; i < 6; i++)
17        cout << p[i] << "\u20ac";
18    cout << endl;
19 }
```

Malá odbočka: mechanizmus posielania parametrov pointrami sa používa aj v iných programovacích jazykoch, ale častokrát "pod kapotou". Napríklad Python posielá základné typy ako parametre (tzv. *volanie hodnotou*), ale na zoznamy a iné zložité typy posielá pointer (tzv. *volanie referenciou*). Pythonovský program

```
1 def zelena(x):
2     x = 42
3
4 def modra(x):
5     x[0] = 42
6
7 a = 1
8 zelena(a)
9 print(a)
10
11 b = [1, 2, 3]
12 modra(b)
13 print(b)
```

vypíše

```
1
[42, 2, 3]
```

Do funkcie `zelena` sa poslalo číslo, preto sa menila hodnota lokálnej premennej vo svete funkcie `zelena` (ktorá spolu s jej svetom zanikla), do funkcie `modra` sa poslal pointer na zoznam `b`. Tento spôsob volania dáva zmysel, lebo väčšinou je to presne to, čo chceš, ale pre začínajúcich programátorov to niekedy môže byť mätúce. Koniec malej odbočky.

Vyriešili sme, ako posielat polia ako parametre, ale stále nám ostáva problém s globálnymi poľami. Pamäťaš sa, že keď sme pri kreslení obrázkov chceli mať globálne pole, museli sme ho vyrobiť tak, aby už v čase komplilácie bola známa jeho veľkosť. Riešili sme to tak, že sme pole spravili dostatočne veľké a používali z neho iba tak veľký kus, ako sme potrebovali. Toto evidentne nie je najlepší prístup. Lepšie riešenie je tzv. *dynamická alokácia*: príkaz `new int[100]`; vyhradí v odľahlom kúte pamäte miesto na 100 premenných typu `int` a vráti pointer naň. Toto miesto je vyhradené pre tvoj program, ale vrátený pointer je jediný spôsob, ako sa k tej pamäti dostať: ak si ho neuložíš do premennej (alebo ju neskôr zabudneš), pamäť stále ostane vyhradená pre tvoj program, ale ten sa k nej už nikdy nebude môcť dostať. Ak už alokovanú pamäť nepotrebuješ, treba ju vrátiť systému príkazom `delete[]`. Treba byť opatrny, lebo to, či je pamäť vyhradená, alebo nie, nijak nevidno. Dobrý zvyk je hneď po volaní `delete[]` priradiť do odalokovanej premennej `nullptr`.

```

1 #include <iostream>
2 using namespace std;
3
4 int *a = (int *)12345;
5
6 void rob_nieco() {
7     int i;
8     for (i = 0; i < 100; i++) a[i] = 42;
9 }
10
11 void vypis(int x) { cout << x << ":\u21d3" << (unsigned long)a << endl; }
12
13 int main() {
14     vypis(1);
15     a = new int[100];
16     vypis(2);      // a ukazuje na pridelenú pamäť
17     rob_nieco();   // v poriadku, píšem do svojej pamäte
18     delete[] a;    // pridelená pamäť sa označila ako volná
19     vypis(3);      // hodnota a sa nezmenila, stále ukazuje na to isté miesto,
20                 // ale tá pamäť mi už nepatrí
21     rob_nieco();   // <- !! toto by bol problém - systém medzitým mohol
22                 // tú pamäť dať niekomu inému
23 }
```

Tento program vypíše napr.

```

1: 12345
2: 94751025169088
3: 94751025169088
```

Pri priradovaní si treba dávať pozor, hlavne ak je pointer na dynamicky alokovanú pamäť súčasť zloženého typu, napr.

```

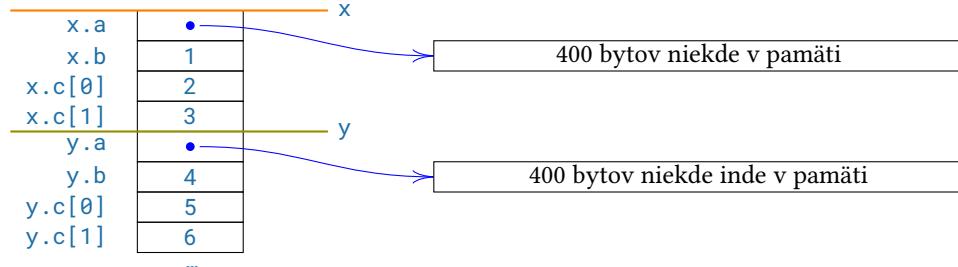
1 struct Kvak {
2     int *a, b, c[2];
3 };
```

Ak napíšeš

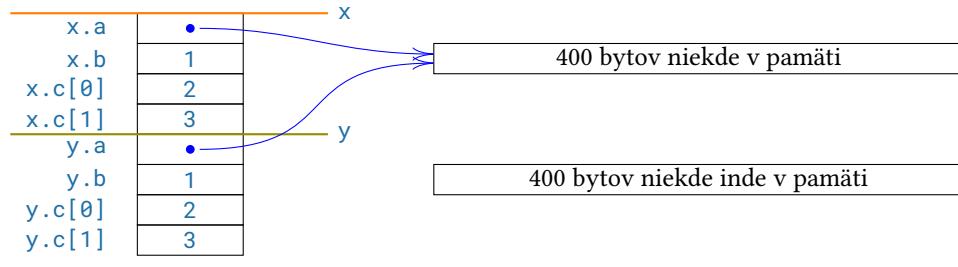
## Polia revealed

```
1 Kvak x, y;
2 x.a = new int[100];
3 y.a = new int[100];
```

a ešte ponastavuješ ďalšie hodnoty, môže to v pamäti vyzeráť takto:



keď potom napíšeš `y=x`, stane sa toto:



Navždy si stratil 400 bytov pamäte a navyše ak napíšeš `delete[] x.a; delete[] y.a;` program zhavaruje s chybou

```
free(): double free detected in tcache 2
Aborted
```

lebo si dvakrát odalokoval tú istú pamäť (aj keď si sa k nej dostał z iných pointrov).

**Úloha 65.** Napiš funkciu, ktorá ako parametre dostane číslo  $n$  a pointer na pole  $n$  čísel a toto pole v pamäti otočí. Napr. ak pred volaním bolo  $n = 5$  a v poli boli čísla 1 2 3 4 5 po volaní tam bude 5 4 3 2 1.

**Úloha 66.** Napiš funkciu, ktorá ako parametre dostane číslo  $n$  a pointer na pole  $n$  čísel a toto pole skopíruje (t.j. vráti pointer na iné pole, v ktorom budú tie isté prvky).

**Úloha 67.** Napiš funkciu, ktorá ako parametre dostane číslo  $n$ , pointer na pole  $n$  čísel a číslo  $k$  a pole v pamäti cyklicky posunie o  $k$  miest. Napr. ak  $n = 5$ ,  $k = 2$  a pole bolo 1 2 3 4 5, tak po volaní funkcie pole bude 3 4 5 1 2.

**Úloha 68.** Napiš funkciu, ktorá dostane parameter  $n$  a vráti pointer na dynamicky alokované pole, v ktorom bude prvých  $n$  prvočísel<sup>2</sup>.

**Úloha 69.** Napiš funkciu, ktorá dostane ako parametre dve utriedené polia veľkostí  $n$  a  $m$  (t.j. dostane  $n$ ,  $m$  a dva pointre) a výrobí utriedené pole dĺžky  $n + m$ , ktoré obsahuje prvky z oboch vstupných polí.

<sup>2</sup> Prvočíslo je číslo  $p$ , ktoré nemá iných deliteľov ako 1 a  $p$ . Inými slovami  $p \neq 1$  pre všetky  $i < p$ . Ak to chceš jemne vylepšíť, premysli si, že ak má číslo  $a$  nejakého deliteľa, tak má aj deliteľa veľkosť nanajvýš  $\sqrt{a}$ . Ak na začiatku napišes `#include<cmath>`, môžeš použiť funkciu `sqrt(x)` na výpočet  $\sqrt{x}$  (vracia typ `double`).

**Úloha 70.** Napíš funkciu, ktorá dostane ako parameter  $n$  a dynamicky alokované pole dĺžky  $n$  a utriedi ho pomocou algoritmu Merge Sort. Algoritmus Merge Sort je rekurzívny algoritmus, ktorý funguje takto: pole dĺžky 1 je utriedené, netreba nič robiť. Ak má pole dĺžku aspoň 2, rozdel' ho na dve polovice (t.j. vyrob si nové polia správnej dĺžky, prekopíruj do nich prvky z pôvodného pola) a rekurzívne obe utried' . Potom použi algoritmus z úlohy 69 na to, aby si vyrobil výsledné pole (nezabudni vrátiť pamäť z pomocných polí).

**Úloha 71.** Majme typ `struct Bod{float x,y;};` Napíš funkciu, ktorá ako parameter dostane pole bodov a utriedi ho<sup>3</sup> podľa  $x$ -ovej súradnice.

**Úloha 72.** Na vstupe je číslo  $n$ , a pole  $n$  čísel  $t[0], \dots, t[n-1]$ , ktoré predstavujú časované bomby:  $i$ -ta bomba vybuchne po  $t[i]$  sekundách. Pyrotechnik vie každú sekundu zneškodniť jednu bombu. Napíš program, ktorý zistí, či sa mu podarí zneškodniť všetky bomby. Napr. pre vstup  $9\ 3\ 2\ 5\ 2$  sa mu to podarí: v prvej sekunde zneškodní poslednú bombu, preto v druhej sekunde budú časovače na bombách ukazovať  $8\ 2\ 1\ 4\ \times$ . Pyrotechnik akurát včas zneškodní bombu číslo 2, takže bude stav  $7\ 1\ \times\ 3\ \times$ , opäť v poslednej sekunde zneškodní bombu č.1:  $6\ \times\ \times\ 2\ \times$  a posledné dve bomby zneškodní s prehľadom. Pre vstup  $3\ 4\ 3\ 7\ 1\ 3$  sa mu to ale nepodarí.

---

<sup>3</sup>napr. pomocou algoritmu *Insertion Sort* z kapitoly 12

## 20 Zásobník ako vlastný typ

V kapitole 13 sme používali zásobník tak, že sme mali pole fixnej veľkosti a jednu premennú, ktorá udávala aktuálny počet prvkov v zásobníku. Keby sme chceli mať viac zásobníkov, mali by sme viac polí a viac premenných. Aby sme v tom udržali poriadok, je dobré spraviť si vlastný typ, kde budú obe veci pokope, napr.

```
1 #include <iostream>
2 using namespace std;
3
4 const int N = 100000;
5
6 struct Stack {
7     int a[N], n;
8 };
9
10 int main() {
11     Stack s1, s2;
12
13     s1.n = 0;
14     s2.n = 0;
15
16     // pridaj prvok
17     s1.a[n]=1;
18     s1.n++;
19 }
```

Takáto definícia má dva problémy: vždy, keď vyrobíme zásobník, vyhradí sa v pamäti veľké pole, aj keby sme dopredu vedeli, že budeme potrebovať menej. Druhý problém je, že na pridanie prvku treba dva príkazy, čo pri častom používaní môže byť neprehľadné. Prvý problém vyriešime dynamickou alokáciou: zásobník si bude pamätať, aké veľké pole má alokované a na začiatku ho inicializujeme. Inicializáciu aj pridanie prvku zariadime funkciami. Tieto musia ako parameter dostávať pointer na zásobník, aby mohli meniť jeho premenné<sup>1</sup>.

```
1 #include <iostream>
2 using namespace std;
3
4 struct Stack {
5     int *a, n, max;
6 };
7
8 void resize(Stack *s, int max) {
9     (*s).a = new int[max];
10    (*s).max = max;
11    (*s).n = 0;
12 }
13
14 bool push(Stack *s, int x) {
15     if ((*s).n == (*s).max) return false;
16     (*s).a[(*s).n] = x;
17     (*s).n++;
18     return true;
19 }
```

<sup>1</sup>Keby dostali ako parameter namiesto `Stack *s` iba `Stack s`, pri volaní funkcie sa zásobník (t.j. premenné `a`, `n`, `max`) skopíruje do lokálneho sveta funkcie, tam sa zmení a po skončení volania spolu so svetom zanikne. Dynamicky alokované pamäť by ostala, ale pointer na ňu by sa stratil. To nie je to, čo chceme.

```

21 void discard(Stack *s) {
22     delete[] (*s).a;
23     (*s).a = nullptr;
24 }
25
26 int main() {
27     Stack s1, s2;
28
29     resize(&s1, 10);
30     resize(&s2, 1000);
31     push(&s1, 4);
32     discard(&s1);
33     discard(&s2);
34 }
35 }
```

Funkcia `push` nám teraz kontroluje, aby zásobník nepretiekol: ak je plný, prvok nevloží a vráti `false`. Volaanie `(*p).x`, kde `p` je pointer na typ, ktorý má zložku `x` je natol'kočasté, že má skratku `p->x`. Teda namiesto `(*s).a` môžeš písť `s->a`. Týmto sme dosiahli, že všetky operácie so zásobníkom sú schované v jednom type a príslušných funkciách. Keby sme sa rozhodli naprogramovať zásobník nejak inak<sup>2</sup> nemusíme meniť zvyšok programu.

Všetky tri funkcie `resize`, `push`, `discard` sú podobné v tom, že prvý parameter je pointer na premennú typu `Stack` a funkcia nejakým spôsobom túto premennú mení. Pretože logicky patria k typu `Stack`, je možné ich definovať priamo v type. Typ, ktorý má definované funkcie, sa volá *rieda*, jemu priradené funkcie sa volajú *metódy*. Premennej typu rieda sa zvykne hovoriť *objekt* a jeho zložkám (jednotlivým premenným vovnútri typu) sa občas hovorí *atribúty*. Ale to všetko sú len mená, na ktorých priliš nezáleží<sup>3</sup>. Dôležité je, že funkcie `resize`, `push` a `discard` môžeš definovať priamo v type `Stack`. Takéto funkcie dostávajú prvý "neviditeľný" parameter `Stack *this`, ktorý obsahuje pointer na objekt (premennú), na ktorom pracujú. Jednotlivé atribúty (premenné) sa potom dajú písť bez zmienky o premennej (t.j. môžeš písť `zzz` namiesto `this->zzz`, aj keď to druhé je rovnako správne). Pretože metóda má neviditeľný parameter `this`, dá sa zavolať iba v spojení s nejakým objektom (premennou). Na to sa používa rovnaký zápis pomocou bodky, ako keď sa pristupuje k atribútom (zložkám) objektu (premennej). Takže môžeme mať niečo takéto:

```

1 #include <iostream>
2 using namespace std;
3
4 struct Stack {
5     int *a, n, max;
6
7     void resize(int _max) {
8         max = _max;
9         a = new int[max];
10        n = 0;
11    }
12
13     bool push(int x) {
14         if (n == max) return false;
15         a[n] = x;
16         n++;
17         return true;
18     }
19 }
```

<sup>2</sup>Zásobník je tak jednoduchá dátová štruktúra, že iný spôsob ľažko vymyslieť, ale stretneme aj oveľa zložitejšie veci, kde to bude dávať dobrý zmysel.

<sup>3</sup>"A rose by any other name would smell as sweet" — W. Shakespeare

## Zásobník ako vlastný typ

---

```
20 void discard() {
21     if (a != nullptr) // aby sme náhodou nevolali delete[] viackrát
22         delete[] a;
23     a = nullptr;
24 }
25 };
26
27 int main() {
28     Stack s1, s2;
29
30     s1.resize(10);
31     s2.resize(1000);
32     s1.push(4);
33     s1.discard();
34     s2.discard();
35 }
```

Príkaz `s1.resize(10);` zavolá metódu `resize` s parametrom 10 na objekte `s1`, t.j. vyrobí sa nový svet pre funkciu `resize`, premenná `_max` sa v ňom nastaví na 10 a neviditeľná premenná `Stack *this` sa nastaví na `&s1`. Pri vykonávaní funkcie premenné `max`, `a`, `n` nie sú v jej svete definované. Ale skôr, ako by sa začali hľadať medzi globálnymi premennými, skúsi sa `this->max`, `this->a`, `this->n`.

Skoro dokonalé, ale ešte tomu zopár vecí chýba. Ak na začiatku zabudneme zavolať `resize` premenné `a`, `n`, `max` budú v nedefinovanom stave, takže potom ďalšie volania budú robiť paseku. Podobne sa nám môže stať, že zabudneme zavolať `discard` a stratí sa nám pamäť. Na to, aby sme tieto problémy vuriešili, slúžia dve špeciálne metódy: *konštruktor* a *deštruktor*. Ak v rámci typu (tryedy) definuješ funkciu, ktorá sa volá rovnako ako typ a nemá návratovú hodnotu<sup>4</sup>, táto metóda sa zavolá vždy, keď sa vyrobí nová premenná (či už osamote alebo v poli) hned potom, ako sa jej vyhradí v pamäti miesto<sup>5</sup>. Konštruktorm vieme zabezpečiť, že náš zásobník nikdy nebude neinicializovaný. Podobne, ak napišeš funkciu, ktorej meno je meno typu s prefixom `~`, táto metóda sa zavolá vždy tesne predtým, ako premenná zanikne (či už preto, že zanikne svet funkcie, v ktorom vznikla, alebo sa volá `delete[]` na dynamicky alokovanú pamäť, v ktorej bola<sup>6</sup>.

```
1 struct Stack {
2     int *a, n, max;
3
4     Stack() { // konštruktor
5         a = nullptr;
6         n = 0;
7         max = 0;
8     }
9
10    ~Stack() { discard(); } // deštruktor
11
12    // zmena veľkosti
13    void resize(int _max) {
14        if (a != nullptr) { // ak tam už niečo bolo, skopírujeme
15            int *b = new int[_max];
16            if (n > _max) n = _max;
17            for (int i = 0; i < n; i++) b[i] = a[i];
18            discard(); // zahodíme, čo tam bolo doteraz
19            a = b;
20        } else { // nemali sme nič naalokované, vyrobíme nové
21            a = new int[_max];
```

<sup>4</sup>Konštruktor a deštruktor je jediná výnimka z pravidla, že funkcia musí mať návratovú hodnotu, keď už nič iné tak aspoň typ `void`

<sup>5</sup>Môžeš skúsiť pridať do konštruktora nejaký výpis a potom v hlavnom programe zavolať napr. `Stack *a = new Stack[5];`.

<sup>6</sup>Skús si opäť pridať výpis do deštruktora a pozrieť sa, ako sa správa.

```

22     n = 0;
23 }
24 max = _max;
25 }
26
27 // pridanie prvku, iba ak máme miesto
28 bool push(int x) {
29     if (n == max) return false;
30     a[n] = x;
31     n++;
32     return true;
33 }
34
35 void discard() {           // odalokujeme pamäť
36     delete[] a;
37     a = nullptr;
38 }
39

```

Teraz je už náš zásobník presne, ako sme ho chceli mať<sup>7</sup>. Vždy, keď budeš potrebovať zásobník, stačí urobiť copy&paste definície typu `Stack` a dá sa pohodlne používať. Lenže robiť zakaždým copy&paste je otrava a navyše keď budeš mať 5–6 podobných tried, tvoj program sa stane neprehľadný. Tento problém elegantne rieši *preprocesor*: prv než sa súbor s programom dá komplilátoru, text súboru prejde jednoduchý prekladač. Ten netuší nič o premenných, typoch, príkazoch atď, len spracováva text. Príkazy pre preprocesor sa spravidla začínajú znakom `#` a jeden z nich je `#include`. Keď v programe napíšeš `#include "velryba"` tak preprocesor nájde súbor `velryba` v aktuálnom adresári<sup>8</sup> a spraví copy&paste jeho obsah na mieste `#include`. Preto ak si celú definíciu typu `Stack` uložíš do súboru `stack.h`, tak kedykoľvek napíšeš `#include "stack.h"`, na tom mieste sa nakopíruje definícia typu `Stack`.

A na záver tejto kapitoly ešte drobnosť. Predstav si, že budeš mať súbor `stack.h` a spravíš inú triedu, `struct Opica{Stack a,b;};`, ktorú si uložíš do súboru `opica.h`. Pretože si nechceš pamätať, že vždy pred použitím `#include "opica.h"` treba napísat `#include "stack.h"`, napíšeš `#include "stack.h"` ako prvý riadok v súbore `opica.h`. Teraz keď napíšeš `#include "opica.h"` tak preprocesor nájde súbor `opica.h` vloží ho do textu a pokračuje v spracovávni už aktualizovaného súboru. V ňom nájde `#include "stack.h"`, tak nájde súbor `stack.h`, vloží ho tam a pokračuje a celé to funguje. Problém ale môže nastať, ak neskôr chceš písat program, v ktorom chceš použiť zásobník aj opicu. Napíšeš

```

1 #include "opica.h"
2 #include "stack.h"
3
4 int main() {
5     // môj program s opicou a zásobníkom
6 }

```

Teraz sa ale stane to, že súbor `stack.h` sa vloží do programu dvakrát (raz v hlavnom súbore, raz pri spracovaní súboru `opica.h`) a komplilátor vyhlási chybu. Tento problém sa spravidla rieši pomocou symbolov, ktoré vie preprocesor spracovávať. Riadok `#define pumpa 118` definuje v preprocesore symbol `pumpa`: kdekoľvek sa v texte programu vyskytne reťazec `pumpa`, preprocesor ho nahradí číslom 118. Preprocesor má aj podmienky, ktoré testujú, či je alebo nie je nejaký symbol definovaný. Zvyčajný spôsob, ako napísat súbor `stack.h` je

<sup>7</sup>v programe som použil novú skratku: ak v cykle `for` chceš použiť premennú, ktorá bude viditeľná iba vo svete zloženého príkazu v cykle, môžeš je vyrobtiť priamo v príkaze `for(int i=0;...`

<sup>8</sup>Ak namiesto úvodzoviek použiješ "zobáčiky", bude sa hľadať nie v aktuálnom adresári, ale v jednom z default, ktoré sú nastavené pri inštalácii. To je presne to, čo sa deje, keď napíšeš `#include <iostream>`

## Zásobník ako vlastný typ

---

```
1 #ifndef _subor_stack_h_uz_je_v_programe_
2 #define _subor_stack_h_uz_je_v_programe_
3
4 struct Stack{
5     // všetko čo treba
6 };
7
8#endif
```

Ked' preprocesor načítava súbor `stack.h` prvýkrát, symbol `_subor_stack_h_uz_je_v_programe_` nie je definovaný, preto sa pokračuje<sup>9</sup>, preto sa definuje a vloží sa definícia `Stack`. Pri každom ďalšom raze už symbol bude definovaný a definícia `Stack` sa preskočí.

---

<sup>9</sup>všimni si, že podmienka je `ifndef` ako `if not defined`

V predchádzajúcej kapitole sme si navrhli typ `Stack`, ktorý spravoval zásobník čísel typu `int`. Čo keby si chcel zásobník typu `float`, alebo `Farba`? Jedna možnosť by bola vyrobiť zakaždým nový typ, napr. `StackInt`, `StackFloat` a pod. Dalo by sa to tak, že skopíruješ `Stack` a na vhodných miestach prepíšeš `int` na nový typ. Ale to nie je dobrý prístup, lebo keď niekedy v budúcnosti budeš chcieť niečo zmeniť, musíš to meniť rovnako na veľa miestach...oštara. V C++ je na to pohodlný mechanizmus, ktorý sa volá *šablóny*(templates). Sú podobné preprocessoru v tom, že sa spracovávajú predtým, ako sa vôbec program začne kompilovať a nahradzujú nejaký symbol iným textom. Ale zároveň ich spracováva už kompilátor, takže vie, kde sa v programe očakáva meno typu, kde premenná a pod. Najjednoduchšie je to vidno na príklade:

```

1 template <typename T>
2 struct Stack {
3     T *a;
4     int n, max;
5
6     Stack() {...}
7     ~Stack() {...}
8     void resize(int _max){...}
9     bool push(T x) {...}
10    void discard() {...}
11 };

```

Týmto hovoríš predpis (šablónu), ako vyrobiť typ zásobník, v ktorom sa budú ukladať prvky hocjakého typu `T` (ľahko doplníš, čo treba do vybodkovaných častí; na vhodných miestach treba nahradíť `int` za `T`). Keď potom v programe napíšeš `Stack<int> s`; kompilátor zoberie šablónu, dosadí za `T` typ `int` čím dostane definíciu typu `Stack<int>` a vyrobí premennú príslušného typu. Ak neskôr napíšeš `Stack<Farba> sf`; kompilátor zoberie šablónu, vyrobí príslušný typ a premennú.

Šablóna sa to volá preto, lebo `Stack` sám osebe nie je typ. Je to len predpis, ktorým hovoríš kompilátoru, ako si má vyrobiť typy `Stack<int>` alebo `Stack<Farba>` a pod., ak ich časom bude potrebovať použiť. Jendotlivé typy `Stack<int>`, `Stack<Farba>` a pod. sú preto nezávislé typy, ktoré sa vyrobia až vtedy, keď sa v programe vyskytnú. A, samozrejme, šablóna môže mať viacero parametrov, napr.

```

1 template <typename S, typename T>
2 struct Dvojica {
3     S s;
4     T* t;
5 };

```

a potom môžeš mať premennú `Dvojica<int, Farba> d`; ktorá bude obsahovať `int d.s` a `Farba* d.t`. Alebo môžeš mať aj premennú `Dvojica<int, Dvojica<int, int>> x`; a potom `x.t` bude pointer na typ `Dvojica<int, int>`, takže napr. `x.t->s` je typu `int`.

Podobne môže byť šablóna na funkcie. Ak napíšeš

```

1 template<typename T>
2 T lama(T z) { return z; }

```

v programe sa nepridá nič, ale hovoríš kompilátoru, ako si vyrobiť (inak celkom zbytočné) funkcie

```

1 int lama<int>(int z){return z;}
2 Farba lama<Farba>(Farba z){return z;}

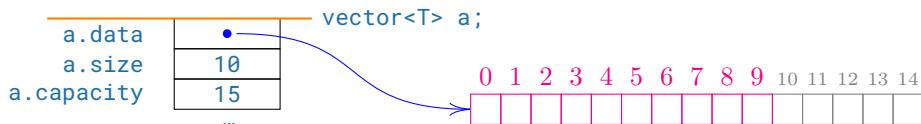
```

STL: typ vector a string

a tak podobne, ak ich niekedy v programe bude potrebovať zavolať. Kompilátor navyše niekedy vie odvodíť parametre šablóny, a vtedy ich netreba písaať. Ak napišeš `lama(42)`, komplilátor pochopí, že tým myslíš funkciu `lama<int>(42)` a vyrobí si ju podľa šablóny. Pravidlá, čo a ako si má domyslieť, sú pomerne zložité, nateraz stačí o tom vedieť, aby ťa to neprekvapilo.

Súčasťou definície jazyka C++ je *Standard Template Library* (STL) – knižnica, ktorá obsahuje šablóny na veľa užitočných dátových štruktúr a algoritmov. Jedným z nich je typ `vector`, ktorý sa veľmi podobá nášmu typu `Stack`: obsahuje v sebe pole prvkov, vie mu rezervovať veľkosť, vie pridať prvok na koniec a pod. Keď ho chceš použiť, treba na začiatku programu pridať `#include <vector>`.

Pre hociajky typ `T`, premenná typu `vector<T>` obsahuje pointer `T*`, ktorý ukazuje na dynamicky alokovanú pamäť. Vektor obsahuje `size` prvkov a má alokovanú pamäť na `capacity` prvkov:



Tu je niekoľko <sup>1</sup>užitočných funkcií:

<code>vector&lt;T&gt; a;</code>	vypočíta vektor <code>a</code>
<code>vector&lt;T&gt; a(n);</code>	vypočíta vektor <code>a</code> a alokuje $n$ prvkov
<code>vector&lt;T&gt; a(n, v);</code>	vypočíta vektor <code>a</code> , alokuje $n$ prvkov a nastaví ich na <code>v</code> (napr. <code>vector&lt;int&gt; a(5, 42);</code> )
<code>a[i]</code>	$i$ -ty prvek <sup>2</sup>
<code>b = a</code>	kopíruje celý vektor <sup>3</sup> , t.j. podobné ako riešenie úlohy 66
<code>a.data()</code>	pointer ( <code>T*</code> ) na dynamicky alokované pole
<code>a.size()</code>	počet prvkov aktuálne uložených v poli
<code>a.capacity()</code>	počet prvkov, na ktoré je rezervovaná pamäť
<code>a.reserve(n)</code>	realokuje pole (podobne ako nás <code>Stack</code> ), aby malo kapacitu $n$
<code>a.shrink_to_fit()</code>	realokuje pole, aby kapacita presne zodpovedala počtu prvkov
<code>a.clear()</code>	vymaže všetky prvky, kapacita sa nezmení
<code>a.resize(n)</code>	zmení počet alokovaných prvkov na $n$ ; ak sa veľkosť zmenší, prvky, ktoré sa do poľa nezmestia, sa vymažú
<code>a.resize(n, v)</code>	to isté, ale ak sa dopĺňajú prvky, ich hodnota je <code>v</code>
<code>a.push_back(x)</code>	pripíše <code>x</code> na koniec; ak by nestačila kapacita, realokuje pole na väčšie
<code>a.pop_back()</code>	odstráni posledný prvek

<sup>1</sup>Na stránkach ako je napr. [www.cppreference.com](http://www.cppreference.com) sa dá nájsť kompletná referencia na všetky funkcie tried z STL.

<sup>2</sup>Podobne ako pri pristupe do pola, je to tzv. **referencia**, takže sa dá priradovať  $a[1]=16$ . Ako také niečo urobíť ti poviem neskôr.

<sup>3</sup>Opäť, neskôr sa dozvieš, ako také niečo urobiť

Veľmi podobný typu `vector` je typ `string` (`#include <string>`): má rovnaké funkcie ako `vector<char>` a aj niečo navyše. Napríklad má operátor `>>` pre načítanie zo vstupu: program `string s; cin >> s;` prečíta zo vstupu znaky (až po prvý whitespace), alokuje pamäť v reťazci `s` a uloží ich tam. Užitočná je aj funkcia `getline(cin, s)`, ktorá do reťazca `s` prečíta riadok zo vstupu (aj s medzerami). Občas sa hodí použiť aj verzia s troma parametrami `getline(cin, s, delim)` ktorá za ukončovač riadku považuje znak `delim` (napr. `getline(cin, s, ',')` uloží do `s` reťazec zo vstupu po prvú čiarku). K reťazcu sa dá pripojiť ďalší reťazec, jeho časť, alebo pole znakov pomocou metódy `append`, napr. `s.append("kuk").append(t, 2, 4)` do `s` najprv<sup>4</sup>pripojí (pole znakov) `"kuk"` a potom úsek 4 znakov z reťazca `t` počnúc od pozície `t[2]`. Podreťazec (substring) sa dá vyrobiť `t=s.substr(2, 4);` Metóda `find` vráti prvý výskyt podreťazca od danej pozície: ak `s="barbakan"`, tak `s.find("ba")` vráti 0 (rovako ako `s.find("ba", 0)`) a `s.find("ba", 1)` vráti 3. Ak sa podreťazec v reťazci nenachádza, vráti sa špeciálna hodnota `string::npos`.

**Úloha 74.** Predpokladajme, že máme premennú typu `string`, v ktorej je uložený zápis čísla (napr. reťazec `"4247"`). Napíš funkciu, ktorá vráti premennú typu `int`, v ktorej bude hodnota tohto čísla.

Takáto funkcia je už v knižnici `string` naprogramovaná a volá sa `stoi`<sup>5</sup>; dá sa napísat `int x = stoi(s);`. V kombinácii s `find` a `substr` sa `stoi` dá použiť na čítanie čísel z reťazca. Keď treba čítať zložitejšie vstupy, hodí sa trieda `stringstream` (`#include <sstream>`). Premenná typu `stringstream` funguje podobne ako `cin` a `cout`, ale namiesto štandardného vstupu a výstupu vie čítať a zapisovať do reťazca. Má špeciálnu metódu `str()`, ktorou sa dá zistíť aj nastaviť reťazec, nad ktorým práve pracuje. Ukážme si to na príklade:

**Úloha 75.** Na vstupe je viacero riadkov, ktoré obsahujú rôzne znaky (okrem `#`). Medzi nimi sú rôzne povkladané aktívne úseky ohraničené `#`, ktoré obsahujú iba čísla a medzery. Vstup je taký, že ak napíšeme čísla z aktívnych úsekov za sebou dostaneme číslice čísla  $\pi$ , t.j. `31415926535.....`. Napíšte program, ktorý vyberie čísla z aktívnych úsekov a v premennej typu `double` uloží číslo  $\pi$ .

Príklad vstupu

```
1 K3d s0m 1s13l # 3 14 # c3z h0ru#1 5#
2
3 str3t0l #      9#s0m t4m
4 p0tv0ru#2##65##
```

Príklad riešenia je tu:

```
1 #include <iostream>
2 #include <sstream>
3 #include <string>
4 using namespace std;
5
6 int main() {
7     string s;
8     stringstream out;
9     double pi;
10
11    out << "0.";
12    while (cin.good()) {
13        getline(cin, s);
14        int i = 0, j;
15        while ((i = s.find("#", i)) != string::npos) {
16            j = s.find("#", i + 1);
17            stringstream in;
```

<sup>4</sup>Všimni si, ako sa dajú volania `append` reťazíť. Je to preto, že metóda `append` vracia *referenciu* na daný `string`; viac sa o tom dozvieš v ďalšej kapitole.

<sup>5</sup>*string-to-integer*, podobná funkcia je `stoi` ktorá vracia `long`.

## STL: typ `vector` a `string`

---

```
18     in.str(s.substr(i + 1, j - i - 1));
19     int n;
20     for (in >> n; !in.fail(); in >> n)
21         out << n;
22     i = j + 1;
23 }
24 }
25 out >> pi;
26 pi = 10 * pi;
27 cout << out.str() << endl;
28 cout << pi << endl;
29 }
```

Urobil som si premennú typu `stringstream` na pozliepanie úsekov zo vstupu. Streamy (`cin`, `cout` aj premenné typu `stringstream`) majú metódy `bool good()` a `bool fail()`, ktoré hovoria, či je stream pripravený čítať a či sa posledné čítanie z neho podarilo. Preto hlavný `while` cyklus na riadku 12 číta zo vstupu riadky, kým tam nejaké sú a volanie `getline(cin, s)` ich ukladá do reťazca `s`. Vo vnútornom cykle na riadku 15 nastavím `i` a `j` na pozície<sup>6</sup> dvoch po sebe idúcich znakov `#` (všimni si `i = j + 1`; na konci cyklu na riadku 22). Potom si vyrobím `stringstream`, ktorému nastavím podreťazec `s` medzi `i` a `j` ako vstupný reťazec volaním `in.str(...)`. V najvnútornejšom cykle na riadku 20 čítam z `in` čísla a zapisujem ich do `out` bez medzier. Po dočítaní vstupu bude reťazec `out.str()` obsahovať `"0.31415..."`, preto z `out` môžem prečítať premennú typu `double`. Všimni si, že `out.str()` aj po načítaní obsahuje celý buffer, znaky z neho sa pri čítaní nestrácajú. Keby som ho chcel vyprázdniť, mohol by som použiť `out.str("")`.

Streamy sú urobené tak<sup>7</sup>, že sa vedia tváriť ako typ `bool`, takže kratšia verzia vyzerá takto:

```
1 #include <iostream>
2 #include <sstream>
3 #include <string>
4 using namespace std;
5 int main() {
6     string s;
7     stringstream out;
8     double pi;
9     out << "0.";
10    while (getline(cin, s)) {
11        int i = 0, j;
12        while ((i = s.find("#", i)) != string::npos) {
13            j = s.find("#", i + 1);
14            stringstream in(s.substr(i + 1, j - i - 1));
15            int n;
16            while (in >> n) out << n;
17            i = j + 1;
18        }
19    }
20    out >> pi;
21    pi = 10 * pi;
22    cout << out.str() << endl;
23    cout << pi << endl;
24 }
```

**Úloha 76.** Na vstupe sú dve čísla. Napíš program, ktorý vypíše ich súčin. Pozor, čísla môžu byť ľubovoľne veľké.

<sup>6</sup> Tu som použil čudný zápis `while ((i=x) == y) ...`. Priradenie je predsa príkaz. Áno, ale dá sa použiť aj ako výraz a vtedy ako výsledok vracia priradenú hodnotu. V predchádzajúcom zápisu to znamená *Do i prirad x a ak priradená hodnota bola y, pokračuj vo vykonávaní cyklu*.

<sup>7</sup> Ako je to presne urobené ti poviem v kapitole 28.

## Konštruktory, referencie, operátory a iný cukor

22

V predchádzajúcej kapitole si videl, že typ `vector` mal niektoré divné vlastnosti (napr. že `a[x]` vráti `x`-tú premennú podobne ako pole). Teraz ti chcem ukázať, že to nie je žiadna mágia. Podľme popriadiu. Povedzme, že chceme urobiť typ na prácu s tabuľkami. Tabuľka má mať  $m$  riadkov a  $n$  stĺpcov a majú v nej byť celé čísla. Aby sme zabránili možnosti, že pole `data` nebude inicializované, spravíme aj konštruktor:

```
1 struct Tabulka {
2     int *data;
3     int m, n;
4
5     Tabulka() {
6         m = 0;
7         n = 0;
8         data = nullptr;
9     }
10
11 };
```

Konštruktor je funkcia takmer ako každá iná, a preto môže mať aj parametre. Konštruktor pre typ `Tabulka` preto môžem zmeniť takto:

```
1 Tabulka(int _m, int _n) {
2     m = _m;
3     n = _n;
4     data = new int[m * n];
5     for (int i = 0; i < m; i++)
6         for (int j = 0; j < n; j++)
7             data[i * n + j] = 0;
8 }
```

A pridáme aj deštruktor:

```
1 ~Tabulka() {
2     if (data!=nullptr) delete[] data;
3 }
```

Teraz sa premenná typu `Tabulka` dá vyrobiť, iba ak dostane dva parametre:

```
1 int main() {
2     Tabulka t(3,3); // v poriadku
3     Tabulka q;      // komplilátor vyhlási chybu
4 }
```

Kým si nemal vlastný konštruktor, komplilátor používal na vytváranie premenných *implicitný* (default) konštruktor. Keď si si ale zadefinoval svoj, default sa už nepoužíva. Stav, keď sa premenná bez potrebných parametrov nedá vyrobiť, je niekedy žiadúci. Len si treba uvedomiť, že teraz nevieš alokovať dynamické pole `Tabulka *x = new Tabulka[10];`, lebo nemáš ako dodať parametre. Môžeš ale dynamicky alokovať jednu premennú, a to pomocou verzie príkazov `new, delete` bez hranatých zátvoriek:

```
1 int main() {
2     Tabulka *t = new Tabulka(3,3);
3     // niečo urob
4     delete t;
5 }
```

## Konštruktory, referencie, operátory a iný cukor

`new T` bez hranatých závoriek urobí veľmi podobnú vec ako `new T[1]`, t.j. alokuje jednu premennú, ale máš možnosť dodať parametre. Len si treba dať pozor na to, že premenné alokované cez `new` treba odalokovať cez `delete` a premenné alokované cez `new[]` treba odalokovať cez `delete[]`.

Okrem alokovania polí môžu parametre v konštruktoroch spôsobať problémy aj pri vnorených typoch. Predstav si, že chceme mať typ, ktorý obsahuje tabuľku a pole dvoch čísel, napr.

```
1 struct Tukan {  
2     int a[2];  
3     Tabulka t;  
4 };
```

Teraz sa zamysli, čo sa stane, keď chceš vyrobiť premennú `Tukan x`: v pamäti sa vyhradí miesto pre jednu premennú typu `int[2]` a jednu premennú typu `Tabulka`. V čase, keď sa zavolá konštruktor `Tukan` (ktorý je zatiaľ iba implicitný), musia sa nastaviť premenné `a` a `t` ich vlastnými konštruktormi. Ale `Tabulka` nemá konštruktor bez parametrov, takže sa vyhlásí chyba. Ešte raz zopakujem: keď sa vytvára premenná nejakého typu, najprv sa vyhradí miesto, potom sa (rekurzívne) zavolajú konštruktory všetkých zložiek a nakoniec sa zavolá hlavný konštruktor<sup>1</sup>. Ako môžeš dodať premennej `t` parametre? Mohlo by ťa napadnúť napísť

```
1 struct Tukan {  
2     int a[2];  
3     Tabulka t(3,3);  
4 };
```

ale tým jednak stratíš možnosť inicializovať tabuľku inými parametrami ako 3, ale hľavne nie je jasné, kedy sa má konštruktor `Tabulka` vlastne volať. Takýto zápis sa ti komplítor ani nepokúsi skompilovať. Riešenie je speciálny zápis konštruktora s dvojbodkou:

```
1 struct Tukan {  
2     int a[2];  
3     Tabulka t;  
4  
5     Tukan() : t(3,3) {}  
6 };
```

Týmto zápisom hovoríš, že `Tukan` má konštruktor bez parametrov, ktorý predtým, ako sa začne vykonávať, zavolá konštruktor `Tabulka` s parametrami 3, 3. Za dvojbodkou môže byť aj viacero konštruktordov oddelených čiarkami, napr.

```
1 Tukan(int x) : t(x,x), a{x,x} {a[1] += a[0];}
```

Ked teraz vyrobiš premennú `Tukan x(3)`; tak najprv sa zavolá konštruktor `Tabulka t(3,3)` a inicializuje sa pole `a` hodnotami 3, 3 a potom sa začne vykonávať hlavný konštruktor, v ktorom sa nastaví `a[1]=6`.

Nielen pri konštruktoroch je niekedy užitočný mechanizmus *default parametrov*. Ak pri definovaní funkcie za posledných<sup>2</sup> niekoľko parametrov dopíšeš =, nastavíš im implicitné hodnoty. Ak sa potom pri volaní príslušný parameter vynechá, doplní sa implicitnou hodnotou. Napr.

<sup>1</sup>toto je jediná postupnosť, ktorá dáva zmysel: v konštruktore typu `Tukan` asi chceš pristupovať k premennej `t`, takže chceš, aby `Tabulka t` už bola inicializovaná

<sup>2</sup>Musia to byť posledné, lebo inak by nebolo jasné, ktorý parameter sa má nahradit. Keby si mal `f(int a=3, int b, int c=1)`, volanie `f(2, 2)` môže znamenať `a=2, b=2 a c=1` (default), ale aj `a=3` (default), `b=2 a c=2`. Kvôli tejto nejednoznačnosti je takéto umiestnenie default parametrov zakázané.

```

1 void chlp(int x = 4, int y = 5) { cout << x << " " << y << endl; }
2
3 int main() {
4     chlp();      // vypíše 4 5
5     chlp(3);    // vypíše 3 5
6     chlp(1,2);  // vypíše 1 3
7 }
```

Preto, keď sa vrátime k našim tabuľkám, keď do konštruktora typu **Tabulka** pridáš default parametre napr. takto: **Tabulka(int \_m = 3, int \_n = 3)**, tak zápis **Tabulka q;** vyrobí tabuľku  $3 \times 3$ .

Podobne **Tabulka \*a = new Tabulka[10];** vyrobí pole 10 tabuľiek, každú rozmerov  $3 \times 3$ .

Ďalej si môžeme napísť funkciu, ktorá ku každému prvku tabuľky priráta číslo. Naša trieda bude vyzerať takto:

```

1 struct Tabulka {
2     int* data;
3     int m, n;
4
5     Tabulka(int _m = 3, int _n = 3) {
6         m = _m;
7         n = _n;
8         data = new int[m * n];
9         for (int i = 0; i < m; i++)
10            for (int j = 0; j < n; j++)
11                data[i * n + j] = 0;
12     }
13
14     void pridaj(int x) {
15         for (int i = 0; i < m; i++)
16             for (int j = 0; j < n; j++)
17                 data[i * n + j] += x;
18     }
19
20     ~Tabulka() {
21         if (data!=nullptr) delete[] data;
22     }
23 };
```

Toto nijak neprekvapí, môžeš napísť napr.

```

1 Tabulka t, r;
2 t.pridaj(5);
```

Teraz chcem napísť funkciu, ktorá priráta nie jedno číslo, ale tabuľku. Možno je trochu prekvapivé, že ju môžem nazvať rovnako: ak majú dve funkcie rôzne parametre, môžu sa volať rovnako, lebo kompilátor aj tak vie, ktorú má použiť. Preto môžem do definície triedy **Tabulka** pridať funkciu **void pridaj(Tabulka x)**. Musím si rozmyslieť, čo robiť, ak majú tabuľky rôznú veľkosť. Rozhodol som sa, že budem prirátať len v rozsahu, ktorý je dosť malý na to, aby sa zmestil do oboch:

```

1 void pridaj(Tabulka x) {
2     int nn = n, mm = m;
3     if (x.n < n) nn = x.n;
4     if (x.m < m) mm = x.m;
5     for (int i = 0; i < mm; i++)
6         for (int j = 0; j < nn; j++)
7             data[i * n + j] += x.data[i * x.n + j];
8 }
```

## Konštruktory, referencie, operátory a iný cukor

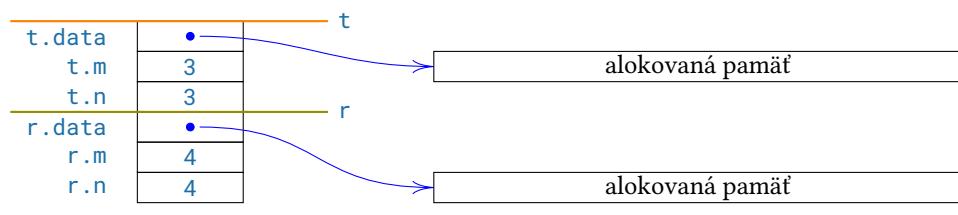
Teraz sa program skompliluje<sup>3</sup> (kompilátor sa nesťaže na dve funkcie s rovnakým menom), ale keď spustím

```
1 int main() {
2     Tabulka t, r(4,4);
3     t.pridaj(5);
4     r.pridaj(t);
5 }
```

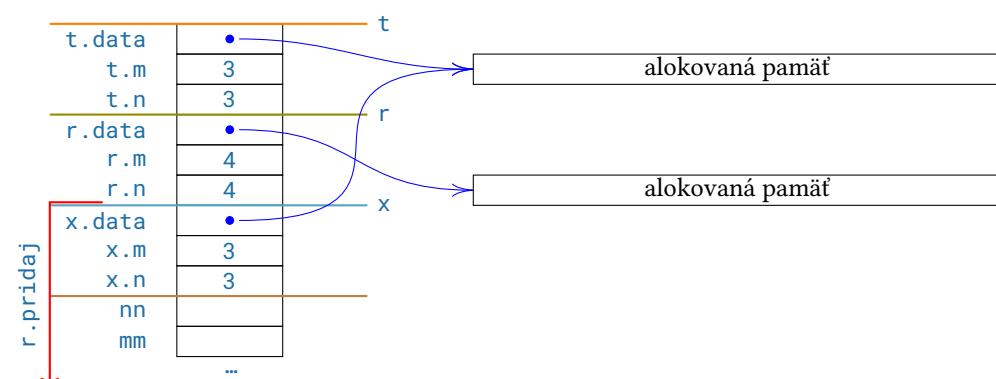
vypíše sa mi

```
free(): double free detected in tcache 2
Aborted
```

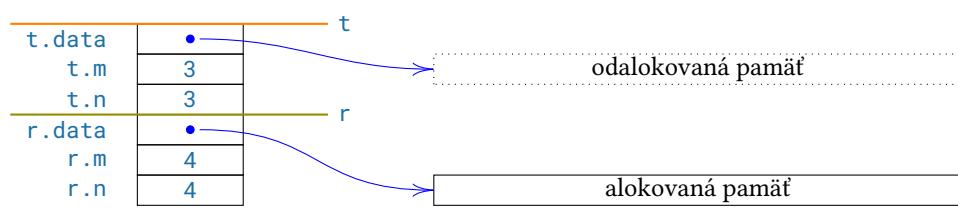
Skús zistíť, kde je problém. Prišiel si na to? Pred volaním `r.pridaj(t);` to vyzeralo takto:



Pri volaní sa vytvoril nový svet pre funkciu `r.pridaj`. V ňom je premenná `x` typu `Tabulka`, do ktorej sa nako-  
píruje premenná `t`.



Ked sa funkcia `r.pridaj` skončí, jej svet zanikne a s ním aj premenné v ňom. Ked ide zaniknúť premenná `x`, zavolá sa jej deštruktor, ktorý odalokuje pamäť.



Pointer `t.data` sa nemenil a ukazuje na to isté miesto v pamäti, ale tá už bola odalokovaná. Ked sa potom končí hlavný program, volá sa deštruktor `t`, ktorý sa pokúša druhýkrát odalokovať tú istú pamäť, a preto program skončí s chybou.

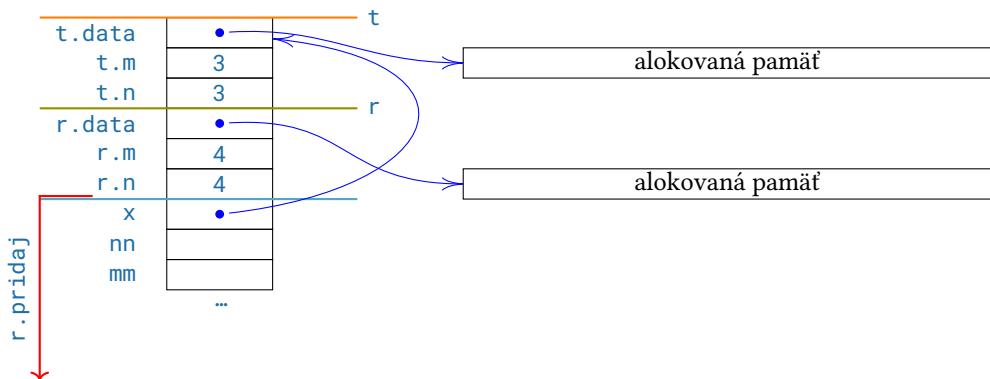
<sup>3</sup>zápis `x += y`; je skratka za `x = x + y`;

Aby sme sa týmto efektom vyhli, je lepšie ako parameter funkcie pridaj neposielať hodnotu `Tabulka`, ale iba pointer `Tabulka *` takto:

```

1 void pridaj(Tabulka *x) {
2     int nn = n, mm = m;
3     if (x->n < n) nn = x->n;
4     if (x->m < m) mm = x->m;
5     for (int i = 0; i < mm; i++)
6         for (int j = 0; j < nn; j++)
7             data[i * n + j] += x->data[i * x->n + j];
8 }
```

Ked' potom zavoláme `r.pridaj(&t)`; do sveta funkcie sa pridá iba pointer na `t`.



Preto ked' zaniká svet `r.pridaj`, zanikne pointer a nie je dôvod volať deštruktor `t`.

Posielať ako parameter do funkcie pointer je veľmi užitočné, ale niekedy, ako uvidíme o chvíľu, to môže byť nepraktické. V C++ je možnosť "zamaskovať" pointer pomocou tzv. *referencie*. Referencia sa v programe tvári ako premenná, ale v skutočnosti je to pointer na jej adresu<sup>4</sup>. Zapisuje podobne ako pointer, ale namiesto hviezdičky sa použije ampersand (&)<sup>5</sup>. Napríklad obidva programy

```

1 void zblnk(int &x) { x++; }
2
3 int main() {
4     int x = 3;
5     zblnk(x);
6     cout << x << endl;
7 }
```

```

1 void brnk(int *x) { (*x)++; }
2
3 int main() {
4     int x = 3;
5     brnk(&x);
6     cout << x << endl;
7 }
```

vypíšu 4, lebo parameter `int &x` hovorí *zober referenciu (t.j. adresu) na premennú x*. Takže v skutočnosti sa do funkcie `zblnk` aj `brnk` posielá pointer na premennú `x`. Pri volaní `brnk(&x)` je jasné, že parameter je pointer, ale pri volaní `zblnk(x)` nevieš rozlíšiť, či je funkcia `zblnk` definovaná s parametrom `int` alebo `int &`. Kedže je to ten druhý prípad, tak `zblnk(x)` zavolať môžeš, ale `zblnk(3)` vyhlásí chybu v tom zmysle, že sa nedá urobiť pointer na trojku.

Referencie sa hodia napríklad keď chceme upraviť, akým spôsobom sa majú tabuľky kopírovať. Podobne ako pri vyrábaní premenných sa volá konštruktor, pri priradení sa volá *operátor priradenia* (=). Default operátor priradenia sa správa tak, že skopíruje premenné, ale môžeme si to upraviť: je to funkcia triedy ako každá iná, len má trochu divný zápis. V našom prípade môžeme triedu `Tabulka` definovať napr. takto:

<sup>4</sup>Už z tohto vidno, že je to tak trochu ako cirkulárka: užitočná vec, ale chceš si pri nej dávať pozor, inak sa neprestaneš čudovať, čo sa to deje.

<sup>5</sup>& a \* môžu byť zo začiatku mätúce. Pri vyrábaní premennej je `int *x` pointer na `int` a `int &x` referencia. Vo výrazoch je `&x` adresa (t.j. pointer) a `*x` je hodnota pointra.

```

1 struct Tabulka {
2     int* data;
3     int m, n;
4
5     Tabulka(int _m = 3, int _n = 3) { alokuj(_m, _n); }
6
7     void alokuj(int _m, int _n) {
8         m = _m;
9         n = _n;
10        data = new int[m * n];
11        for (int i = 0; i < m; i++)
12            for (int j = 0; j < n; j++)
13                data[i * n + j] = 0;
14    }
15
16    void pridaj(int x) {
17        for (int i = 0; i < m; i++)
18            for (int j = 0; j < n; j++)
19                data[i * n + j] += x;
20    }
21
22    void pridaj(Tabulka& x) {
23        int nn = n, mm = m;
24        if (x.n < n) nn = x.n;
25        if (x.m < m) mm = x.m;
26        for (int i = 0; i < mm; i++)
27            for (int j = 0; j < nn; j++)
28                data[i * n + j] += x.data[i * x.n + j];
29    }
30
31    Tabulka& operator=(Tabulka& x) {
32        if (data!=nullptr) delete[] data;
33        alokuj(x.m, x.n);
34        pridaj(x);
35        return *this;
36    }
37
38    ~Tabulka() {
39        if (data!=nullptr) delete[] data;
40    }
41};

```

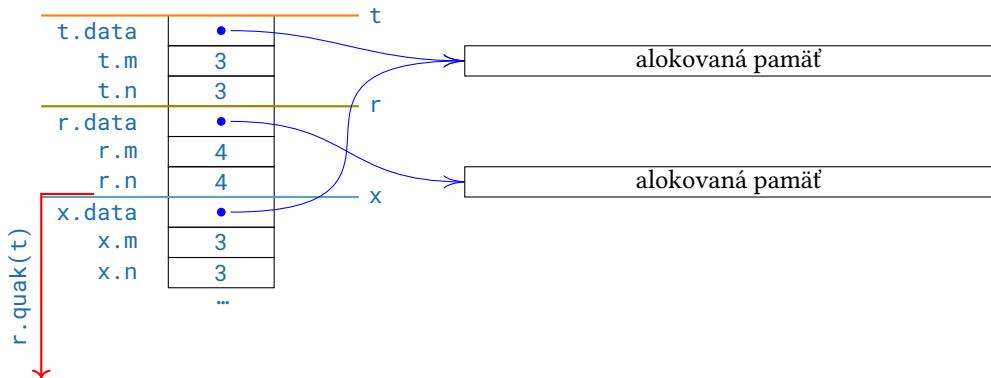
Pridal som funkciu `operator=`, čo je špeciálne meno vyhradené na úpravu operátora priradenia. Ako parameter sa zoberie referencia na tabuľku. Vracia sa tiež referencia, a to referencia na seba (`return *this;`, lebo `this` je pointer na seba a `*this` je jeho hodnota). To ti umožňuje písat skrátene `t = r = q`; V tomto prípade sa najprv zavolá funkcia `r.operator=(q)`, ako parameter sa zoberie referencia na `q`, na základe nej sa upravia hodnoty v `r` a vráti sa referencia na `r`. Tá sa hned zoberie ako parameter pri volaní `t.operator=()`, takže hodnoty `t` sa upravia podľa čerstvo upravených hodnôt `r`.

Podobný ako operátor priradenia je tzv. *copy constructor*, ktorý sa volá vtedy, ak sa má vyrobiť nová premenná z inej. Konkrétnie napr. pri posielaní parametrov do funkcie. V našom príklade máme operátor priradenia, ale ak by si mal napr. v rámci `Tabulka` metódu (bez referencie) `void quak(Tabulka x) { pridaj(x); }`, tak volanie `r.quak(t)` bude mať rovnaký problém ako predtým: pri volaní sa do lokálnej premennej `x` skopírujú hodnoty z `r` a pri skončení fukcie sa zavolá deštruktur, ktorý odalokuje pointer. Keby si okrem konštruktora `Tabulka(int _m, int _n)` pridal ešte konštruktor, ktorý má ako parameter referenciu na už existujúcu premennú typu `Tabulka`:

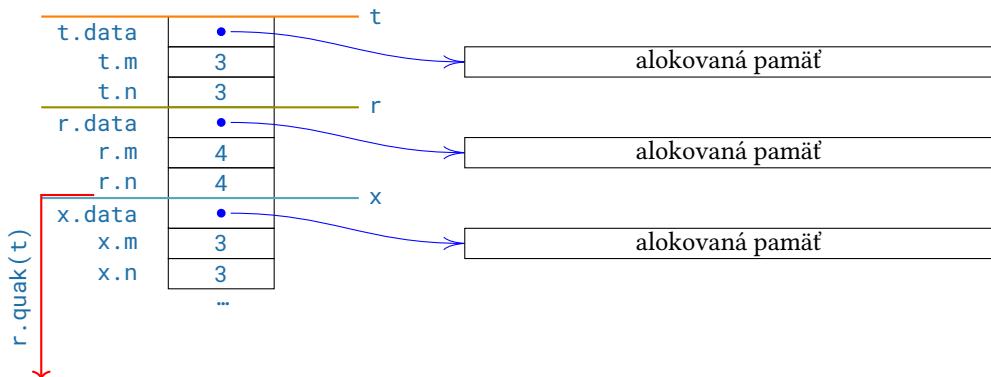
```

1 Tabulka(Tabulka& x) {
2     alokuj(x.m, x.n);
3     pridaj(x);
4 }
```

tak pri volaní `r.quak(t)` sa použije tento konštruktor, ktorý namiesto toho, aby kopíroval pointer, v lokálnej premennej `x` naalokuje pamäť, takže volanie bude namiesto:



vyzerať takto:



Preto deštruktor na konci volania správne odalokuje pamäť v lokálnej premennej `x`. Tento spôsob funguje, ale pri väčších tabulkách je neefektívny: dátá sa najprv kopírujú z `t` do lokálnej `x`, potom z lokálnej `x` do `r`, a nakoniec sa lokálna `x` zahodí. Je lepšie ako parameter do funkcie posielat pointer alebo referenciu.

Operátor priradenia nie je jediný operátor, ktorý sa dá preddefinovať, dajú sa takmer všetky<sup>6</sup>. Ak chceme napríklad vedieť sčítovať tabuľky, môžeme spraviť niečo takéto:

```

1 struct Tabulka {
2     // celá definícia
3 };
4
5 Tabulka operator+(Tabulka &x, Tabulka &y) {
6     Tabulka res(x);
7     res.pridaj(y);
8     return res;
9 }
10
11 int main() {
```

<sup>6</sup>detailedy sú napr. na [en.cppreference.com/w/cpp/language/operators](http://en.cppreference.com/w/cpp/language/operators)

## Konštruktory, referencie, operátory a iný cukor

```
12     Tabulka r, s, t;
13     t = r + s;
14 }
```

Definoval som (mimo triedy **Tabulka**) operátor **+**, ktorý zoberie referencie na dve tabuľky a vytvorí výslednú, ktorá bude ich súčtom. Všimni si, že keby sa vracala hodnota referenciou ako

```
1 Tabulka& operator+(Tabulka &x, Tabulka &y) {
2     Tabulka res(x);
3     res.pridaj(y);
4     return res;
5 }
```

tak by to celé nefungovalo: pri volaní funkcie **operator+** sa v jej svete vytvorí lokálna premenná **res**, nejak sa upraví a vráti sa referencia (teda pointer) na ňu. Lenže pri skončení volania sa celý svet funkcie zruší, zavolá sa deštruktur na **res**, a preto vrátená referencia (teda pointer) je nanič<sup>7</sup>.

Ale keď tento program skúsiš skompilovať, zistíš, že to stále nejde. Prečo? Z rovnakého dôvodu, ako nešlo spustiť **zblnk(3)** o pár odstavcov vyššie: **operator+** vráti *hodnotu* typu **Tabulka**, ale **operator=** má ako parameter *referenciu*, teda pointer na premennú. A ten sa nedá zobrať, ak premenná nie je. Na druhej strane, ona tá premenná v skutočnosti kdesi musí byť: pri využití výrazu je uložená na nejakom bezmennom mieste v pamäti. Ak komplilátoru slúbiš, že ju nebudeš meniť, bude ochotný z nej pointer (referenciu) zobrať. Takže zmeníme definíciu **operator=** na **Tabulka& operator=( const Tabulka& x)**. Teraz ti ale komplilátor bude výčitať, že si nedodrážal slub: v rámci volania **operator=** voláš **pridaj(x)**. Kedže **x** sa tam posielá referenciou, ktorové, či sa tam nezmení. Musíš ho upokojiť a slúbiť, že ani tam sa **x** nebude meniť: **void pridaj(const Tabulka& x)**. Teraz už všetko bude fungovať správne.

Ak si v rámci definície triedy **Tabulka** predefinuješ operátor **()** takto:

```
1 int& operator()(int r, int s) { return data[r * n + s]; }
```

môžeš pristupovať k políčkam tabuľky napr. **t(i, j) = r(i, j);**. To, že k položkám triedy **vector** môžeš pristupovať ako k poľu je preto, že **vector** si predefinoval operátor **[ ]**.

Teraz aj vidíš, že som ťa klamal, keď som hovoril, že **cout <<** je *príkaz* na vypísanie. V skutočnosti **cout** je premenná typu **ostream** definovaná v súbore **iostream**. Trieda **ostream** má predefinovaný operátor **<<** (ktorý pri celých číslach robí bitový posun). Ak by si chcel napr. vypisovať tabuľky, môžeš si definovať<sup>8</sup> (mimo definície triedy **Tabulka**)

<sup>7</sup> Poznámka pre fajnšmekrov. Asi si postrehol, že toto riešenie nie je ideálne: pri volaní **return** sa všetky dátá z **res** skopírujú do výsledku a vzápäť sa volá deštruktur na **res**. Nebolo by lepšie nejak povedať, aby si komplilátor len "premenoval adresy" a v ďalšom používal adresu premennej **res** ako výsledok? Zatiaľ si v programoch videl tzv. **lvalue** (meno premennej, referencia, t.j. niečo, čo môže stať na ľavej strane priradenia) a **rvalue** (konštantá, volanie funkcie, tiež meno premennej, t.j. čokoľvek, čo môže stať na pravej strane priradenia). C++ má navyše aj tzv. **xvalue** (*expiring value*, napr. lokálna premenná pri volaní **return**, čo je **lvalue**, ktorá zároveň komplilátoru hovorí **s mojimi datami si môžeš robiť, čo chceš, aj tak budem za chvíľu volať deštruktur**) a **pvalue** (napr. dočasné premenné, v ktorej je uložený výsledok výrazu pri volaní; ak volám **pridaj(4+7)**, tak pri vytváraní sveta funkcie **pridaj** je v pamäti dočasné premenné **int** s hodnotou 11). Komplilátor vie pri priradení z **xvalue** alebo **pvalue** optimalizovať veci tak, aby sa vyhol kopírovaniu. Na to treba, aby tvoria trieda tzv. **move constructor** a **move assignment** (všetky STL triedy ako **vector**, **string** a pod. ich majú). Takže keby si okrem copy konštruktora **Tabulka(Tabulka& x)** mal aj move konštruktor **Tabulka(Tabulka&& x)** (s dvoma ampersandmi, tzv. **rvalue reference**) a move operátor **Tabulka& operator=(const Tabulka&& x)**, komplilátor by pri volaní **return res;** zavolal move operátor. Ten namiesto kopírovania dát môže len priradiť pointer. Samozrejme, je dôležité, aby sa postaral o to, že následné volanie deštruktora **x** bude v poriadku, napr. **Tabulka& operator=(Tabulka&& x){m=x.m; n=x.n; data=x.data; x.data=nullptr; return \*this;}**. Na explicitné pretypovanie z **lvalue** na **xvalue** slúži funkcia **move**, napr.

```
1 swap(T& a, T& b) {
2     T tmp(move(a)); // z a urob xvalue, použije sa move constructor
3     a = move(b);   // z b urob xvalue, použije sa move assignment
4     b = move(tmp); // z tmp urob xvalue, použije sa move assignment
5 }
```

Výsledkom toho je, že premenné z STL (napr. **vector**) môžeš pokojne vraciať z funkcie hodnotou; komplilátor použije **move** sémantiku a dátu sa nebudú kopírovať.

<sup>8</sup>všimni si, že vraciam referenciu na prvý parameter, takže volanie **cout << x << y** robí to, čo sa očakáva

```

1 ostream& operator<<(ostream& str, Tabulka& o) {
2     for (int i = 0; i < o.m; i++) {
3         for (int j = 0; j < o.n; j++) str << o(i, j) << " ";
4         str << endl;
5     }
6     return str;
7 }
```

tak môžeš zavolať napr `Tabulka t; cout << t;` Nemôžeš ale urobiť `cout << s + t` opäť z rovnakých dôvodov: `operator<<` potrebuje referenciu. Opäť to môžeš vyriešiť tak, že slúbiš, že parameter `o` bude konštantný, t.j. `const Tabulka& o`. Tým ale vyrobíš iný problém: zápis `o(i, j)` je volanie funkcie `o.operator()`, čo je metóda objektu `o`. No a keďže sa nevie, či ho nemení, komplíátor ju odmietne. To, že daná metóda nemení svoj objekt, vieš slúbiť tak, že `const` napíšeš za definíciu metódy. Takže celý výsledný program by mohol vyzerat nejak takto (mám dve verzie `operator()`: jedna vráti hodnotu, a tá je `const`, druhá referenciu a tá nie je `const`; komplíátor si vie vybrať podľa toho, čo práve potrebuje):

```

1 #include <iostream>
2 using namespace std;
3
4 struct Tabulka {
5     int* data;
6     int m, n;
7
8     Tabulka(int _m = 3, int _n = 3) { alokuj(_m, _n); }
9     Tabulka(const Tabulka& x) {alokuj(x.m, x.n); pridaj(x); }
10    ~Tabulka() { delete[] data; }
11    void alokuj(int _m, int _n) { ... }
12    void pridaj(int x) {...}
13    void pridaj(const Tabulka& x) { ... }
14    Tabulka& operator=(const Tabulka& x) { pridaj(x); return *this; }
15    int& operator()(int r, int s) { return data[r * n + s]; }
16    int operator()(int r, int s) const { return data[r * n + s]; }
17    Tabulka& operator+=(int x) { pridaj(x); return *this; }
18 };
19
20 ostream& operator<<(ostream& str, const Tabulka& o) { ... }
21 Tabulka operator+(const Tabulka& x, const Tabulka& y) { ... }
22
23 int main() {
24     Tabulka t; t += 5;
25     Tabulka r = t; r += 2;
26     r(1,1) = 42;
27     cout << t + r;
28 }
```

## 23 Binárne vyhľadávanie a logaritmy

Hrajme takúto hru: myslím si číslo od 1 po  $N$  a tvojou úlohou je ho uhádnuť. Môžeš sa opýtať na nejaké číslo a ja ti odpoviem *viac*, *menej* alebo *uhádol si*. Ako budeš hádať, aby si čo najskôr uhádol? Koľko otázok na to budeš potrebovať?

Povedzme, že  $N = 1000$ . Ak sa na začiatku opýtaš na 5 a máš šťastie, dostaneš odpoveď *menej* a vieš, že to môže byť iba 1, 2, 3, alebo 4. Ale ak šťastie nemáš, príliš si si nepomohol: môže to byť čokoľvek medzi 6 a 1000. Ale možnože nehrám férovo a podvádzam: v skutočnosti si nemyslím číslo, ale čakám, čo sa ma opýtaš. Ak sa spýtaš na 5, odpoviem ti *viac* a budem sa tváriť, že som si mysel číslo medzi 6 a 1000 (potom to už musím dodržať, lebo inak by si zistil, že som podvádzal). Najlepšie, čo môžeš v tejto situácii urobiť, je opýtať sa na číslo v strede. Koľko otázok potrebuješ, aby si touto stratégiou uhádol číslo? Jednoduchšie je pýtať sa opačnú otázku: pre aký najväčší rozsah  $N$  ti stačí  $n$  otázok? Podme postupne. Ak môžeš položiť 1 otázku, tak  $N$  môže byť najviac 3: ak si myslím číslo od 1 po 3, tak sa najprv spýtaš na 2, a keď si neuhádol a dostaš si odpoveď *menej*, vieš, že som si mysel 1, a ak si dostaš odpoveď *viac* tak vieš, že som si mysel 3. Keď si ale myslím číslo od 1 po 4, jedna otázka ti nestaačí: spýtaš sa na 2, je odpoviem *viac* a teraz nevieš, či som si mysel tri alebo štyri.

Čo ak máš dve otázky? Ak si myslím číslo od 1 po 7, vieš ho zistiť: opýtaš sa na 4. Ak ti poviem *menej* máš jednu otázku na to, aby si uhádol číslo od 1 do 3 (tri čísla jednou otázkou, to vieš), ak ti odpoviem *viac* máš jednu otázku na to, aby si uhádol číslo od 5 do 7 (zase tri čísla jednou otázkou). Ak máš tri otázky, môžem si myslieť číslo od 1 po 15: prvou otázkou za opýtaš na 8 a ostanú ti dve otázky na uhádnutie čísla spomedzi siedmich.

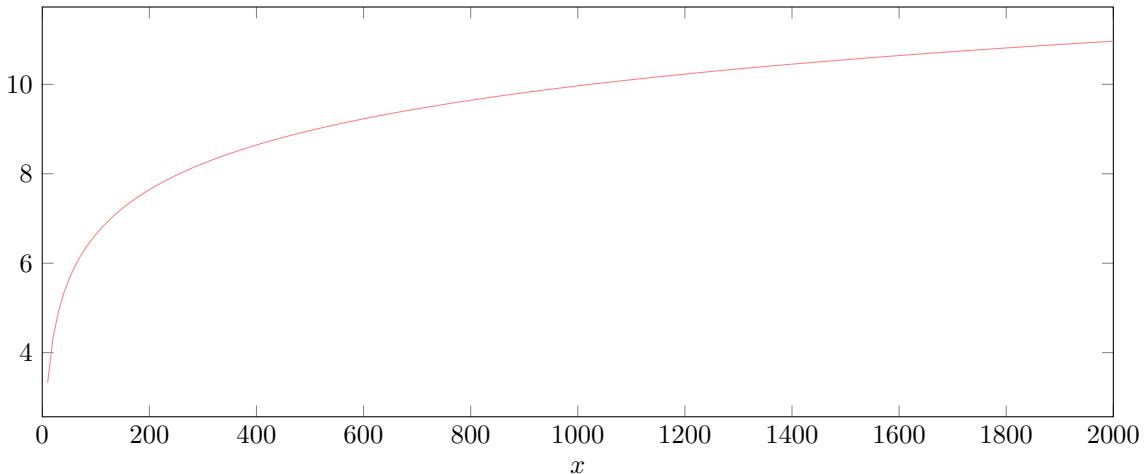
Keď si označí  $m_n$  najväčšie číslo, ktoré vieš  $n$  otázkami uhádnuť, tak sme videli, že  $m_1 = 3, m_2 = 7, m_3 = 15$ . Vždy, keď máš o jednu otázku viac, vieš robiť to isté, preto  $m_{n+1} = 2m_n + 1$  (ak máš  $n+1$  otázok, opýtaš sa na 1 číslo v strede a v každom prípade ti ostane  $n$  otázok na uhádnutie čísla z rozsahu  $m_n$ ). Ľahko si overíme, že  $m_n = 2^{n+1} - 1$ . Pre malé hodnoty to vieme priamo:

$$\begin{aligned}m_1 &= 2^2 - 1 = 4 - 1 = 3 \\m_2 &= 2^3 - 1 = 8 - 1 = 7 \\m_3 &= 2^4 - 1 = 16 - 1 = 15\end{aligned}$$

potom ak už vieme, že  $m_n = 2^{n+1} - 1$ , tak vieme aj, že  $m_{n+1} = 2m_n + 1 = 2(2^{n+1} - 1) + 1 = 2 \cdot 2^{n+1} - 2 + 1 = 2^{n+2} - 1$ . Z toho potom vieme dokázať, že  $m_{n+2} = 2^{n+3} - 1$  a tak ďalej. Teda  $2^{n+1}$  je prvé číslo, na ktoré potrebuješ viac ako  $n$  pokusov.

Naspäť k prvej otázke: ak hádam čísla z rozsahu 1 až  $N$ , koľko otázok potrebuješ? Nech je to číslo  $n$ , potom  $2^{n+1} - 1 \geq N$ , t.j.  $2^n \geq (N+1)/2$ . Takéto číslo má svoje meno: volá sa *logaritmus*. Presne povedané: *logaritmus z čísla  $n$  je také číslo  $x$ , že  $2^x = n$* . V našom prípade umocňujeme dvojkou, preto hovoríme, že používame *logaritmus pri základe 2*. Napr.  $\log 16 = 4$ , lebo  $2^4 = 2 \cdot 2 \cdot 2 \cdot 2 = 16$ . Umocňovanie aj logaritmy sa dajú prirodzene rozšíriť aj na desatinné čísla<sup>1</sup>, preto si môžeme nakresliť graf funkcie  $\log(x)$ :

<sup>1</sup>Ak  $b$  je celé číslo, umocňovanie je jednoduché:  $a^b = \overbrace{a \cdot a \cdots a}^b$ . Preto platí  $a^{b+c} = \overbrace{a \cdots a}^b \cdot \overbrace{a \cdots a}^c = a^b \cdot a^c$ . Podobne  $a^{bc} = \underbrace{\overbrace{a \cdots a}^b \cdot \overbrace{a \cdots a}^b \cdots \overbrace{a \cdots a}^b}_{c} = (a^b)^c$ . Ak chceme, aby tieto pravidlá stále platili, kolko by bolo  $a^{-1}$ ? Musí platí  $a^{-1} \cdot a^1 = a^{-1+1} = a^0 = 1$ . Preto  $a^{-1} = \frac{1}{a}$ . Kolko by bolo  $a^{\frac{1}{2}}$ ? Musí platí  $a = a^1 = a^{\frac{1}{2} + \frac{1}{2}} = a^{\frac{1}{2}} \cdot a^{\frac{1}{2}}$ . Preto  $a^{\frac{1}{2}} = \sqrt{a}$ . Podobne  $a^{\frac{1}{p}}$  je také číslo  $x$ , že  $x^p = a$ . Také číslo voláme  $p$ -ta odmocnina z  $x$  a značíme  $\sqrt[p]{x}$ . Teraz vieme, že  $a^{\frac{p}{q}} = a^{\frac{1}{q}p} = (\sqrt[q]{a})^p$  a  $a^{-\frac{p}{q}} = a^{\frac{p}{q} \cdot (-1)} = ((\sqrt[q]{a})^p)^{-1} = \frac{1}{(\sqrt[q]{a})^p}$ . Ak povieme, že  $\log a$  je také číslo  $b$ , že  $2^b = a$ , máme logaritmus pre všetky čísla, nielen pre mocniny dvojkys. Z pravidiel pre umocňovanie vyplýva, že  $\log(ab) = \log a + \log b$ , pretože  $\log(ab)$  je také číslo, že  $2^{\log(ab)} = ab$ . Lenže  $a = 2^{\log a}$  a  $b = 2^{\log b}$ , preto  $ab = 2^{\log a} 2^{\log b} = 2^{\log a + \log b}$ , takže  $2^{\log(ab)} = 2^{\log a + \log b}$  a preto aj  $\log(ab) = \log a + \log b$ . Rovnako vieš zdôvodniť, že  $\log(a/b) = \log a - \log b$ ,  $\log(a^b) = b \log a$  a pod.



Všimmi si, že stále rastie, ale čím ďalej, tým pomalšie. Napr.  $\log(4398046511104)$  je 42.

Zoberme si teraz programátorskú úlohu, ktorá sa podobá našej hre. Predpokladajme, že máme pole `int a[n]`; v ktorom sú rôzne čísla utriedené od najmenšieho po najväčšie. Našou úlohou je napísat funkciu, ktorá pre nejaké zadané číslo `x` zistí, či sa v poli `a` nachádza alebo nie. Môžeme to spraviť priamočiaro:

```

1 bool find(int n, int *a, int x) {
2     for (int i = 0; i < n; i++)
3         if (a[i] == x) return true;
4     return false;
5 }
```

Táto funkcia má evidentne lineárnu zložitosť. Vieme to spraviť lepšie? Môžeme zvoliť rovnakú stratégiu ako pri hádacej hre: pozrieme sa do stredu poľa, zistíme, či je tam väčšie alebo menšie číslo a podľa toho hľadáme bud' v ľavej alebo v pravej časti. Budeme mať teda cyklus, v ktorom budeme kontrolovať menšie a menšie intervaly. Okraje intervalu si budeme pamätať v premenných `l` a `r`. Stred intervalu `m` vyrátame ako priemer `l` a `r`, t.j.  $m=(l+r)/2$ . Treba si dať trochu pozor na to, kedy skončí, ale celá funkcia môže vyzerať nejak takto:

```

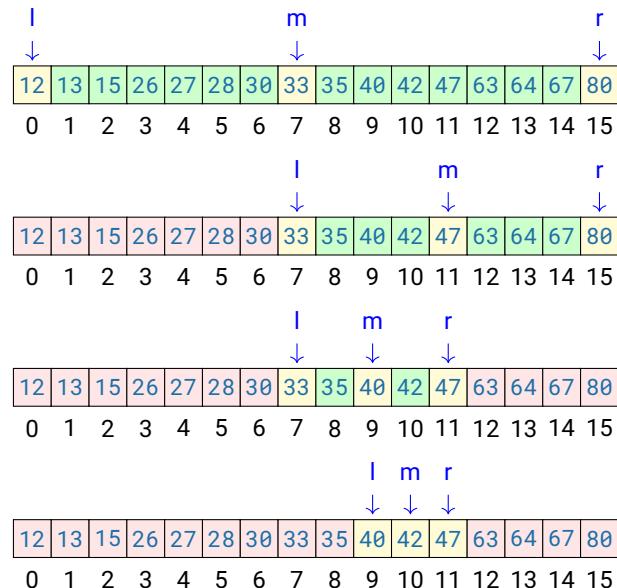
1 bool najdi(int n, int *a, int x) {
2     int l = 0, r = n - 1, m;
3     if (a[l] > x || a[r] < x) return false;
4     if (a[l] == x || a[r] == x) return true;
5     while (l < r - 1) {
6         m = (l + r) / 2;
7         if (a[m] == x) return true;
8         if (a[m] > x)
9             r = m;
10        else
11            l = m;
12    }
13    return false;
14 }
```

Prečo to funguje? Na začiatku skontrolujeme okrajové podmienky. Počas celého cyklu potom vieme, že hľadané číslo je určite ostro väčšie `a[l]` a zároveň ostro menšie ako `a[r]`,<sup>2</sup> preto keď `l` a `r` sú susedné, vieme že hľadané číslo sa v poli nenachádza. Predpokladajme, že v nasledovnom poli hľadáme číslo 42. Výpočet by potom mohol vyzerať takto:

<sup>2</sup>všimni si, že na začiatku cyklu to platí a v cykle vždy meníme `l` a `r` tak, aby to ostalo zachované

## Binárne vyhľadávanie a logaritmy

---



Funkcia [najdi](#) má logaritmickú zložitosť. Môžeš sa pýtať, načo je logaritmická zložitosť dobrá, keď už na samotné prečítanie vstupného poľa potrebujeme lineárnu zložitosť. Častokrát potrebujete počas tvojho programu veľakrát riešiť podobný problém. Napríklad keď hľadáš cestu v bludisku, na každej križovatke sa musíš rozhodnúť, kam odbočiť. Vtedy má zmysel, aby rozhodovacia funkcia mala zložitosť menšiu ako lineárnu, lebo vstup je už raz načítaný v pamäti. Iné použitie binárneho vyhľadávania, ktoré uvidíš, je podobné ako v hre na začiatku kapitoly: v pamäti nie je uložené celé pole, ale máš len funkciu, ktorá ti hovorí *viac* alebo *menej*.

Malá obočka: ak máš zložitejší cyklus, ako napríklad v tomto prípade a chceš sa uistiť, že program funguje správne, je dobré vymyslieť si tzv. *invariant*. Invariant je podmienka, ktorá platí vždy na začiatku cyklu. V tomto prípade invariant hovorí, že ak je hľadané číslo v poli, je napravo od  $l$  a naľavo od  $r$ . Ak invariant platí na začiatku jednej iterácie cyklu, určite platí aj na začiatku ďalšej: ak sa začala ďalšia iterácia, tak hľadané číslo nie je  $a[m]$  a určite sa nachádza medzi novým  $l$  a novým  $r$ . Zároveň vidno, že vzdialenosť  $l$  a  $r$  sa v každej iterácii cyklu zhruba spoločný. Takto by sa dalo dokázať, že program vždy skončí po logaritmickom počte iterácií a dá správnu odpoveď. Podobný spôsob rozmyšľania sme použili pri algoritme *Insertion Sort* v kapitole 12 aj pri úlohe o klaunoch v kapitole 13.

**Úloha 77.** V pamäti je pole  $a$ , ktoré obsahuje  $n$  prirodzených čísel utriedených od najmenšieho. Napiš funkciu, ktorá dostane ako parameter číslo  $k$  a povie, koľkokrát sa  $k$  nachádza v poli. Funkcia by mala mať logaritmickú zložitosť.

**Úloha 78.** V pamäti sú dve usporiadane polia  $a$  a  $b$ , pričom každé obsahuje  $n$  čísel. Napiš funkciu, ktorá zistí, ktorý prvok by bol na  $n$ -tej pozícii, keby sa obe polia spoločne utriedili. Napr. pre  $n = 6$  a polia  $3 \ 6 \ 9 \ 12 \ 15 \ 18$  a  $2 \ 4 \ 6 \ 8 \ 10 \ 12$  je výsledok 8, lebo spoločne utriedené pole je  $2 \ 3 \ 4 \ 6 \ 6 \ 8 \ 9 \ 10 \ 12 \ 12 \ 15 \ 18$  a v ňom na šiestom mieste je osmička. Funkcia by mala mať logaritmickú zložitosť.

Častokrát je pri riešení úloh užitočný prístup "binárne vyhľadávanie nad výsledkami". V ňom, podobne ako v hre s hádaním čísla, nemáme pole, ale iba nejakú funkciu, ktorej sa viem opýtať. Ukážem ti to na príklade. Začnime s úplne nesúvisiacou úlohou:

**Úloha 79.** Na vstupe je číslo  $n$ . Napiš program, ktorý zistí, kolko núl je na konci čísla  $n!$  ( $n!$  znamená faktoriál ako v úlohe 73). Napr. pre vstup 6 je výstup 1, pre vstup 11 je výstup 2 (lebo  $11! = 39916800$ ), pre 123 je výstup 28.

S touto úlohou je ten problém, že  $n!$  veľmi rýchlo rastie a veľmi skoro sa nezmestí ani do premennej typu `unsigned long long int`. Môžeš zobrať riešenie úlohy 73 a jednoducho počet núl na konci zrátať, ale dá sa to urobiť lepšie. Každé prirodzené číslo  $n$  sa dá rozložiť na súčin prvočísel<sup>3</sup> takto: začnem s číslom  $n$ ; ak je

<sup>3</sup>o prvočíslach bola úloha 68

prvočíslo, tak super, inak má deliteľa a platí  $n = a \cdot b$ . No a v rozklade pokračujem pre  $a$  aj  $b$  dokola, až sa dopracujem k prvočíslam. Napríklad  $120 = 20 \cdot 6 = 4 \cdot 5 \cdot 2 \cdot 3 = 2 \cdot 2 \cdot 5 \cdot 2 \cdot 3$ . Je užitočné si uvedomiť, že sice môžem pri rozklade postupovať rôzne, vždy sa dostanem k rovnakým prvočíslam (možno v inom poradí)<sup>4</sup>. Napr.  $120 = 5 \cdot 24 = 5 \cdot 6 \cdot 4 = 5 \cdot 2 \cdot 3 \cdot 2 \cdot 2$ .

Každá nula na konci čísla znamená, že číslo sa dá vydeliť desiatimi, a teda v prvočíselnom rozklade je jedna päťka a jedna dvojka. Napríklad

$$11! = 11 \cdot 10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 \cdot 3 \cdot 2$$

čo keď rozšípeme, budeme mať

$$11! = 11 \cdot 7 \cdot 5^2 \cdot 3^4 \cdot 2^8$$

Na konci teda budú dve nuly. V prvočíselnom rozklade  $n!$  bude vždy viac dvojek ako päťiek, preto stačí rátať päťky. Napr. pre  $26!$  ich narátať 6: po jednej v číslach 5, 10, 15, 20 a dve v číslu 25. Teraz sa už úloha 79 dá vyriešiť ľahko, napr.

```

1 int nuly(int n) {
2     int z = 0;
3     while (n > 1) { // pre každé číslo
4         for (int i = 5; i <= n; i = i * 5) // všetky mocniny 5
5             if (n % i == 0) z++; // ktoré ho delia
6         n--;
7     }
8     return z;
9 }
```

Ako to ale súvisí s našim rozprávaním o binárnom vyhľadávaní? Pozrime sa na "opačnú" úlohu:

**Úloha 80.** Na vstupe je číslo  $x > 0$ . Napíš program, ktorý vypíše najmenšie  $n$  také, že na konci  $n!$  je  $x$  núl.

Mohol by som ísť v cykle postupne od 1, vždy zavolať funkciu `nuly` a zastaviť sa pri prvom číslе, ktoré má aspoň  $x$  núl.

Čísla, ktorých faktoriál má na konci aspoň  $x$  núl budem volať *dobré*, ostatné sú *zlé*. Pretože  $(n+1)! = (n+1) \cdot n!$ , tak všetky delitele  $n!$  sú aj delitele  $(n+1)!$ . To znamená, že pre rastúce  $n$ , núl na

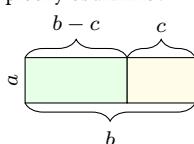
<sup>4</sup>Prečo to platí? Euklides by uvažoval takto: zoberiem si najmenšie číslo  $s$ , pre ktoré môžem mať dva rôzne rozklady, t.j.  $s = p_1 \cdot 2 \cdots p_m = q_1 \cdot q_2 \cdots q_n$ , kde  $p$ -čka aj  $q$ -čka sú prvočísla. Keby nejaké  $p_i = q_j$ , tak ním  $s$  vydelením a dostanem menšie číslo s dvoma rôznymi rozkladmi, takže  $p$ -čka a  $q$ -čka sú rôzne. Dajme tomu, že  $p_1 < q_1$  (keby to tak nebolo, tak premenujem  $p$ -čka a  $q$ -čka). Ak si označím  $P = p_2 \cdot p_3 \cdots p_m$  a  $Q = q_2 \cdot q_3 \cdots q_n$ , tak vidíme, že

$$s = p_1 \cdot P = q_1 \cdot Q$$

Z obidvoch strán môžem odrátať  $p_1 Q$  a stále dostanem nezáporné číslo, lebo  $p_1 < q_1$ . Budem teda mať

$$p_1 P - p_1 Q = q_1 Q - p_1 Q$$

Teraz využijem, že  $ab - ac = a(b - c)$ , čo ľahko vidno z plochy obdĺžnikov



a dostanem

$$p_1(P - Q) = (q_1 - p_1)Q < s$$

Číslo  $(q_1 - p_1)Q$  má jednoznačný rozklad na prvočísla (lebo je menšie ako  $s$ ), a keďže  $p_1$  je prvočíslo, ktoré ho delí, tak niekde v tom rozklade je. Ale nemôže byť v rozklade  $Q$ , lebo ten sa skladá so samých  $q$ -čok a vieme, že  $p$ -čka a  $q$ -čka sú rôzne. Preto  $p_1$  sa nachádza v rozklade  $q_1 - p_1$ . Lenže ak  $p_1$  delí  $q_1 - p_1$ , tak určite delí aj  $q_1$ . Ale to nemôže, lebo  $q_1$  je iné prvočíslo. To celé znamená, že nemôže existovať najmenšie (a teda žiadne) číslo  $s$ , ktoré by nemalo jednoznačný rozklad na prvočísla.

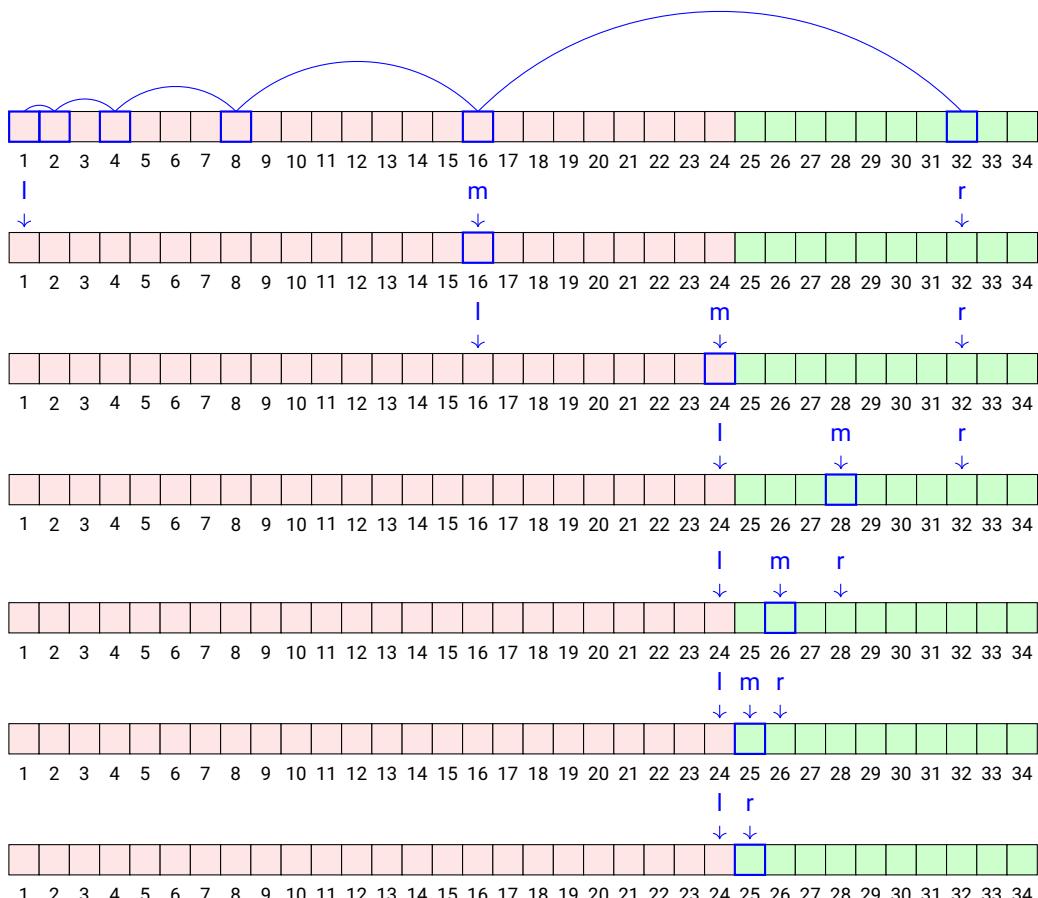
## Binárne vyhľadávanie a logaritmy

konci  $n!$  iba pribúda, nikdy neubúda. Preto v postupnosti  $1, 2, 3, 4, \dots$  sú najprv zlé čísla a potom od istého čísla už sú všetky dobré. Keďže  $x > 0$ , viem, že 1 je určite zlé číslo. Keby som vedel aspoň jedno dobré číslo  $n_0$ , môžem použiť binárne vyhľadávanie: pozriem sa, či je  $n_0/2$  dobré. Ak áno, najmenšie dobré číslo bude medzi 1 a  $n_0/2$ , ak nie, bude medzi  $n_0/2$  a  $n_0$ . Potom budem pokračovať rovnako ako pri binárnom vyhľadávaní: vždy bude platiť, že **l** je zlé a **r** dobré číslo. Keď nakoniec budú **l** a **r** susedné, zjavne **r** je najmenšie dobré číslo. Ako ale nájdem  $n_0$ ? Začнем od jednotky a budem vždy skúšať dvojnásobok:  $2, 4, 8, 16, \dots, 2^i, \dots$ . Časom sa dostanem až k nejakému dobrému číslu  $n_0 = 2^i$ , ktoré určite nebude väčšie ako dvojnásobok najmenšieho dobrého čísla (lebo  $n_0/2$  ešte dobré nebolo). Dostanem sa k nemu na  $i$  pokusov. Keďže  $i$  je také číslo, že  $2^i = n_0$ , platí  $i = \log(n_0)$ . Binárne vyhľadávanie tiež urobí najviac  $\log(n_0)$  pokusov, dokopy teda môj program zavolá funkciu **nuly** najviac  $2 \log(n) + 3$  krát (čiže počet pokusov je logaritmický). Program by mohol vyzerať takto:

```

1 int main() {
2     int x;
3     cin >> x;
4     int l = 1, r = 1, m;
5     while (nuly(r) < x) r = 2 * r;
6     while (l < r - 1) {
7         m = (l + r) / 2;
8         if (nuly(m) < x) l = m;
9         else r = m;
10    }
11    cout << r << endl;
12 }
```

Pre  $x = 5$  by sa postupne skúšali tieto hodnoty (červené sú zlé čísla a zelené dobré, modré štvorčeky sú tie, na ktoré sa volá funkcia **nuly**):



Aký je rozdiel v rýchlosti medzi týmto programom a pôvodnou verziou, ktorá volala funkciu `nuly` zaradom? Pre  $n = 20000$  mi pôvodná verzia bežala minútu a 17 sekúnd, kým tento program zbehol za 48 ms.

Skúsme spoločne spraviť ešte jeden príklad:

**Úloha 81.** Máme  $n$  displejov, ktoré ukazujú hodnoty  $p[0], p[1], \dots, p[n-1]$ . Pre  $i$ -ty displej môžeme zaplatiť  $m[i]$  peňazí, aby sa hodnota na ňom zvýšila o 1. Máme  $k$  dispozíciu  $b$  peňazí. Našim cieľom je, aby displeje ukazovali čo najväčšie čísla, teda aby najmenšie číslo, ktoré ukazuje nejaký displej, bolo čo najväčšie. Napíš program, ktorý pre zadané polia  $p$  a  $m$  a rozpočet  $b$  zistí, aké najväčšie čísla môžu displeje ukazovať. Napr. ak  $p$  je [1 2 3],  $m$  je [2 3 1] a  $b$  je 7, tak výsledok je 3: dvakrát zaplatíme po 2, aby sa prvý displej dostal na 3 a raz zaplatíme 3, aby sa aj druhý displej dostal na 3. Spotrebovali sme akurát 7 peňazí a dosiahli sme, že na všetkých displejoch je aspoň trojka.

Opäť použijeme binárne vyhľadávanie nad výsledkami. Máme nájsť maximálne číslo, ktoré vieme s našim rozpočtom  $b$  dosiahnuť. Ako by sa dalo zistiť, či vieme dosiahnuť nejaké číslo  $k$ ? To je jednoduché: ak je na nejakom displeji číslo  $p[i]$  menšie ako  $k$ , musíme doplatiť  $m[i] * (k - p[i])$ , aby sme ho dostali na hodnotu  $k$ . Toto skontrolujeme pre všetky displeje a overíme, či nám na to vystačia peniaze:

```

1 bool test(int k) {
2     int sum = 0;
3     for (int i = 0; i < n; i++)
4         if (p[i] < k) sum += m[i] * (k - p[i]);
5     return sum <= b;
6 }
```

V hlavnom programe použijeme binárne vyhľadávanie. Budeme udržiavať invariant, že  $l$  vždy dosiahnuť vieme a  $r$  nevieme. Cyklus skončí, ak budú  $l$  a  $r$  susedné, takže  $l$  musí byť najväčšie číslo, ktoré sa dá dosiahnuť.

```

1 while (l < r - 1) {
2     x = (l + r) / 2;
3     if (test(x))
4         l = x;
5     else
6         r = x;
7 }
8 cout << l << endl;
```

Posledná otázka je, ako na začiatku nastaviť  $l$  a  $r$ . Aj keby sme nezaplatili nič, vieme získať hodnotu najmenšieho displeja, preto začneme s  $l=\min(p)$ , kde funkcia `min` vyráta najmenšiu hodnotu z poľa.

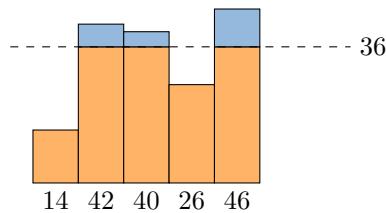
```

1 int min(vector<int> &a) {
2     int res = a[0];
3     for (int i = 1; i < a.size(); i++)
4         if (a[i] < res) res = a[i];
5     return res;
6 }
```

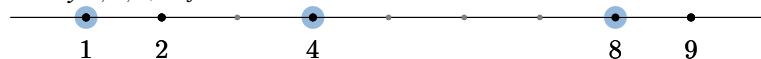
Ak by bol najväčší displej najlacnejší možný (t.j. stačilo by zaplatiť 1 na jeho zváčšenie) a všetky peniaze by sme investovali doňho, aj tak nedosiahneme  $r=\max(p) + b + 1$ .

Tu je niekoľko úloh podobného typu:

**Úloha 82.** Máme  $n$  obdĺžnikov s výškami  $a[0], \dots, a[n-1]$ , ktoré môžeme odpíliť takto: nastavíme výšku  $h$  a z každého obdĺžnika zoberieme časť, ktorá prečnieva cez  $h$ . Napíš program, ktorý pre zadané pole  $a$  a číslo  $m$  nájde najväčšie číslo  $h$ , pri ktorom z odpílených obdĺžnikov zozbierame aspoň  $m$ . Napr. pre obdĺžníky 14 42 40 26 46 a  $m = 20$  je odpoveď 36, lebo s výškou  $h = 36$  pozbierame  $0 + 6 + 4 + 0 + 10 = 20 \geq 20$ .



**Úloha 83.** Na rovnej čiare je  $n$  bodov v pozíciiach  $a[0] \dots a[n - 1]$ . Okrem toho je zadané číslo  $c > 1$ . Treba vybrať  $c$  bodov tak, aby boli od seba čo najďalej (t.j. aby minimálna vzdialenosť medzi dvoma susednými vybratými bodmi bola čo najmenšia). Napíš program, ktorý vypočíta túto vzdialenosť. Napr. pre vstup  $1 \ 2 \ 8 \ 4 \ 9$  a  $c = 3$  je odpoveď  $3$ , lebo keď vyberieme body  $1, 4, 8$ , najbližšie dva sú  $1$  a  $4$  vo vzdialosti  $3$ .



**Úloha 84.** K dispozícii máme  $k$  sád kartičiek s rôznymi znakmi, pričom z  $i$ -tej sady máme  $k$  dispozícii  $a[i]$  kartičiek. Všetkých kartičiek dokopy je  $n$ . Napíš program, ktorý zistí, kolko najviac rovnakých riadkov z kartičiek vieme poskladať. Napr. pre  $k = 3, n = 4$  a vstup  $7 \ 6 \ 3$  je správna odpoveď  $3$ : máme 7 kartičiek **A**, 6 kartičiek **B** a 3 kartičky **C**, preto môžeme spraviť tri riadky **AABC** (a ostanú nám kartičky **ABBB**), ale nemôžeme spraviť 4 riadky.

**Úloha 85.** Máme  $n$  displejov, ktoré majú na začiatku hodnoty  $a[0], \dots, a[n-1]$  a každý z nich sa každú sekundu zníži o 1 až kým nepríde na 0 (a potom tam ostane). Zároveň si každú sekundu môžeme vybrať jeden displej a ten znížiť ešte o  $k$ . Napíš program, ktorý zistí, ako dlho bude trvať, kým všetky displeje klesnú na nulu, ak si vyberáme optimálnym spôsobom. Napr. pre  $k = 5$  a vstup  $2 \ 3 \ 6$  je odpoveď  $2$ , lebo v prvej sekunde si vyberieme druhý displej, ktorý tým dostaneme na nulu, takže po prvej sekunde budú hodnoty  $1 \ 0 \ 5$ . V druhej sekunde si vyberieme tretí displej, takže po druhej sekunde budú všetky displeje na nule.

**Úloha 86.** Na vstupe je  $n$  čísel a číslo  $k$ . Napíš program, ktorý rozdelí čísla do  $k$  súvislých úsekov (každý úsek musí obsahovať aspoň jedno číslo) tak, aby najväčší súčet čísel v jednom úseku bol čo najmenší. Napr. pre vstup  $2 \ 7 \ 2 \ 1 \ 5 \ 9$  a pre  $k = 3$  má program vypísat  $2 \ 7 / 2 \ 1 \ 5 / 9$ .

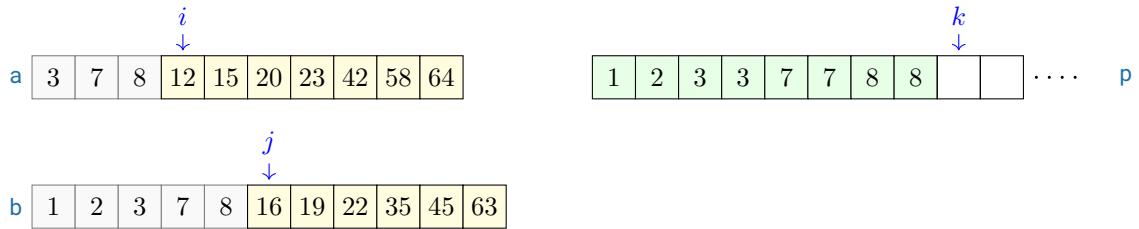
Skôr ako skončíme túto kapitolu o logaritmoch, vrátim sa k úlohe 70 a algoritmu *Merge Sort*. Na to, aby sme utriedili pole čísel, môžeme najprv napisať funkciu `merge` (viď úloha 69) napr. nejak takto:

```

1 void merge(vector<int>& a, vector<int>& b, vector<int>& p) {
2     int i = 0, j = 0, n = a.size(), m = b.size();
3     p.resize(n + m);
4     for (int k = 0; k < n + m; k++) {
5         if (i < n && (j == m || a[i] < b[j])) {
6             p[k] = a[i];
7             i++;
8         } else {
9             p[k] = b[j];
10            j++;
11        }
12    }

```

V nej prechádzame indexom **i** po poli **a**, indexom **j** po poli **b** a vždy menší z aktuálnych prvkov zapíšeme do výsledného poľa **p**:



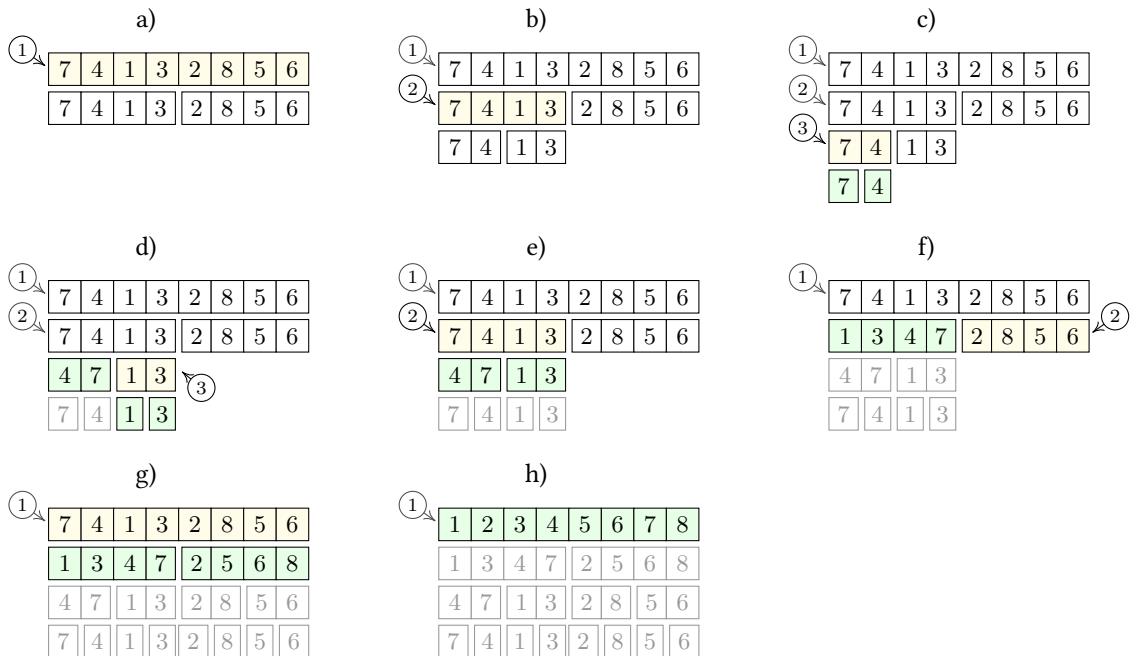
S pomocou funkcie `merge` potom triedenie môže fungovať tak, že vstupné pole rozdelíme na dve polovice, postupne obe utriedíme a na záver spojíme funkciou `merge` do výsledného poľa. Môže to vyzeráť napr. takto:

```

1 void sort(vector<int>& a) {
2     int n = a.size();
3     if (n <= 1) return;
4     int x = n / 2, y = n - x;
5     vector<int> b(x), c(y);
6     for (int i = 0; i < x; i++) b[i] = a[i];
7     for (int i = 0; i < y; i++) c[i] = a[i + x];
8     sort(b);
9     sort(c);
10    merge(b, c, a);
11 }
```

Teraz, keď už vieš, čo sú logaritmy, sa môžeme zamyslieť nad tým, akú má táto funkcia zložitosť. Doteraz si videl príklady lineárnej, kvadratickej a logaritmickej zložitosti, ktoré sa zvyknú značiť  $O(n)$ ,  $O(n^2)$  a  $O(\log n)$ ; zápis  $O(f(n))$  približne znamená *niečo, čo sa celé zmestí pod graf funkcie  $c \cdot f(n)$  pre nejaké číslo  $c$ .*

Poďme sa lepšie pozrieť, čo sa pri výpočte deje. Ak zavoláš `sort` na pole `a = [7 4 1 3 2 8 5 6]`, vytvorí sa svet funkcie `sort` (svet č.1) a v ňom sa vyrobia dve lokálne polia<sup>5</sup> `b = [7 4 1 3]` a `c = [2 8 5 6]` ako na obrázku a).



Pole `c` zatiaľ ostane nedotknuté, vytvorí sa nový svet č.2 a posunie sa doňho ako parameter referencia na `b`. Nový svet začne pracovať na poli `[7 4 1 3]`, opäť vyrobí dve lokálne polia, rozdelí pole na dve časti `[7 4]` a

<sup>5</sup>ktoré potom spolu so svetom zaniknú, zavolajú sa ich deštruktory a uvoľní sa pamäť

## Binárne vyhľadávanie a logaritmy

---

[1 3]. ďalší svet má vstup [7 4], rozdelí ho na dva, vytvoriť opäť nový svet (už č. 4) na jednoprvkové pole [7]. Toto volanie volanie vzápäť skončí, podobne volanie na jednoprvkové pole [4]. Vo svete č.3 sa preto zavolá `merge` a utriedené pole [4 7] sa zapíše do premennej `b` patriacej svetu č.2. Vytvorí sa nový svet č.3 s poľom [1 3] ako na obrázku d) a v ňom sa urobí to isté. Svet č. 3 zanikne a vo výpočte pokračuje svet č.2, ktorý má premenné `b` a `c` pripravené na to, aby zavolal `merge` a skončil, čím sa k slovu opäť dostane svet č.1. Ten zavolá rekurzívne `sort` na pole `b`, takže vznikne opäť nový svet č.2 ako na obrázku f).

Potom sa rovnakým spôsobom spracuje druhá polovica vstuпу, až nastane situácia ako na obrázku g), kde sa k slovu nakoniec dostane svet č.1, ktorý zavolá `merge` a vytvoriť finálny výsledok. Keď sa pozrieš na obrázok h), je tam vlastne tabuľka, ktorá má niekoľko riadkov, každý riadok má dĺžku  $n$ . Táto tabuľka nikdy nebola naraz v pamäti, lebo je tvorená lokálnymi premennými `b` a `c` rôznych svetov, ktoré postupne vznikali a zanikali, ale všetka práca, ktorú násť algoritmus robil, sa dá predstaviť ako písanie do políčok tejto tabuľky: raz sa tam zapísalo, keď sa v nejakom svete pole `a` rozdeľovalo na `b` a `c` a druhýkrát, keď sa robil `merge`. Teda celá práca, ktorú algoritmus spraví, zhruba zodpovedá veľkosti tejto tabuľky. Je jasné, že tabuľka má  $n$  stĺpcov. Ale kolko má riadkov? Toľko, kol'kokrát sa dá rozdeliť  $n$  na polovicu, kým sa nedostaneme k jednotke. Ak sa na to pozriem z opačného konca a počet riadkov nazvem  $h$ , tak v poslednom riadku majú polia dĺžku 1, v predposlednom 2, potom 4, 8, až v  $h$ -tom riadku od konca majú dĺžku  $2^h$ . Preto  $n = 2^h$ , počet riadkov je  $\log n$  a celá zložitosť je  $O(n \log n)$ : horšia ako lineárna, ale oveľa lepšia ako kvadratická (napr. pre  $n = 1000$  je  $n^2 = 1000000$ , ale  $n \log n$  je čosi menej ako 10000).

Pri riešení úloh si si isto všimol, že sú niektoré užitočné funkcie, ktoré sa často opakujú: utriediť pole, nájsť maximum alebo minimum, vymeniť dve premenné, vyhľadať hodnotu v utriedenom poli a podobné. STL ob-sahuje knižnicu `<algorithm>`, kde sú rôzne šikovné algoritmy, ktoré ti pomôžu písat programy rýchlejšie. Napríklad ak máš premennú `vector<int> a`; tak ju vieš utriediť pomocou `sort(a.begin(), a.end())`; V tejto kapitole by som ti chcel vysvetliť, ako a prečo to funguje.

Cieľom návrhárov STL bolo, aby algoritmy boli čo najvšeobecnejšie a dali sa používať v rôznych situáciach. Preto sa ich snažia oddeliť od tzv. *kontajnerov*, čo sú typy, ktoré uchovávajú dátu. Napr. typ `vector` má dátu uložené v jednom dynamicky alokovanom poli, ale mohol by ich mať napr. v piatich rôznych poliach, alebo hocijako inak. Ak chceš napr. utriediť pole, nepotrebuješ na to presne vedieť, ako sú prvky uložené, ale potrebuješ sa v nich vedieť prehŕňať: nejaký konkrétny prvok si zapamätať, vedieť prejsť na ďalší prvok, prejsť na začiatok, povedať, či dva zapamätané prvky sú ten istý a tak podobne. Na to slúžia tzv. *iterátory*. Iterátor nie je súčasť jazyka (ako napr. príkazy a výrazy), je to len označenie: iterátor je hocijaká trieda, ktorá sa vie správnym spôsobom prehŕňať v dátach inej triedy.

Každý kontajner, ktorý chce používať algoritmy STL, musí mať dve metódy `begin()` a `end()`, ktoré vracajú iterátor. Dajme tomu, že máš triedu `struct Puch{ int a,b,c; }`; a chceš tie tri čísla vedieť utriediť volaním funkcie `sort(p.begin(), p.end())`; z STL. Potreboval by si spraviť niečo ako

```

1 struct IteratorPuchu {
2     ...
3 };
4
5 struct Puch {
6     int a,b,c;
7     IteratorPuchu begin() { ... }
8     IteratorPuchu end() { ... }
9 };

```

Pretože `IteratorPuchu` logicky patrí triede `Puch`, využíva sa možnosť C++ definovať typy vovnútri iných typov takto:

```

1 struct Puch {
2
3     struct iterator {
4         ...
5     };
6
7     int a,b,c;
8     iterator begin() { ... }
9     iterator end() { ... }
10};

```

Teraz je `iterator` súčasťou typu `Puch` a celým menom sa volá `Puch::iterator` (dve dvojbodky znamenajú *patriaci*). Typ `vector` to má urobené rovnako, môžeš skompilovať

```

1 vector<int> a;
2 vector<int>::iterator b = a.begin();

```

Funkcie, ako napr. `sort` sú v skutočnosti šablóny, ktoré vytvoria príslušnú funkciu pre hocijaký typ iterátora. Napr. `sort` je definovaná ako<sup>1</sup>

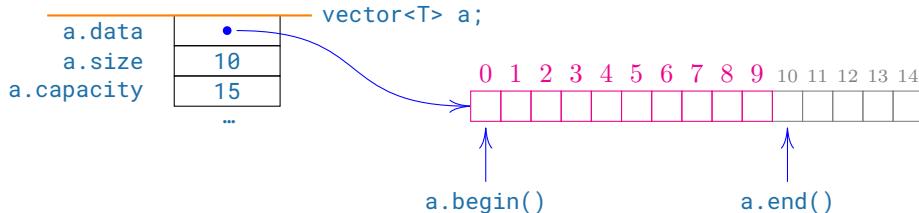
<sup>1</sup>to, že parameter v šabloně sa volá `class` a nie `typename` si nevšímaj, znamená to skoro to isté

## Algoritmy v STL

```
1 template<class RandomIt>
2 void sort(RandomIt first, RandomIt last);
```

Pre triedenie poľa by si mal preto zavolať `sort<vector<int>::iterator>(a.begin(), a.end())`; To tiež funguje, ale stačí zavolať `sort(a.begin(), a.end())`; Je to preto, že ak si komplilátor vie odvodíť typ, dosadí ho do šablóny sám. V tomto prípade vidí, že funkcia `begin()` triedy `vector<int>` vracia typ `vector<int>::iterator`, takže je jasné, že toto je typ, ktorý treba použiť do šablóny. Toto odvodzovanie typov, ktoré komplilátor robí sa dá využiť aj pri definovaní premenných: ak namiesto mena typu použiješ špeciálne meno `auto`, komplilátor sa pokúsi odvodoť typ; napr. namiesto `vector<int>::iterator b = a.begin();` môžeš napísať `auto b = a.begin();`

Každá funkcia z knižnice `algorithm` má povedané, čo presne očakáva od iterátora. Ale vždy treba, aby iterátor vedel vrátiť hodnotu operátorom `*`, aby sa mal operátor `++`, prípadne operátor `+` na posunie o `i` prvkov ďalej a pod. K iterátoru sa preto môžeš správať ako k pointru (a napr. pre typ `vector<int>` to skutočne aj pointer na `int` je). Všetky funkcie STL pracujú na *poluzavertých intervaloch*: začiatočný iterátor je na prvý prvek, koncový iterátor je na prvok za posledným.



Takže tieto programy robia to isté

```
1 for (int i = 0; i < a.size(); i++)
2     cout << a[i] << endl;
3
4 for (int *p = a.data(); p < a.data() + a.size(); ++p)
5     cout << *p << endl;
6
7 for (vector<int>::iterator it = a.begin(); it < a.end(); it++)
8     cout << *it << endl;
9
10 for (auto it = a.begin(); it < a.end(); it++)
11     cout << *it << endl;
```

Len pripomienim, že iterátori spravidla obsahujú pointre do príslušnej dátovej štruktúry, a preto, rovnako ako pointre, môžu prestať byť aktuálne: ak napr. urobíš `auto it = a.end();` a potom `a.push_back(3)`; tak `it` bude ukazovať stále na to isté miesto a nie za koniec vektora `a`. V dokumentácii k jednotlivým kontajnerom sa píše, ktoré iterátori sú pri ktorých operáciach zneplatnené (invalidated). Napr. pri `push_back` sa píše: *ak pri operácii bolo nutné realokovať pole, zneplatnia sa všetky iterátory, inak iba posledný*. To treba mať na pamäti: častá chyba je, že sa v cykle `for (auto it = a.begin(); it < a.end(); it++)` prechádza nejaký kontajner a vnútri cyklu sa robia operácie, ktoré môžu iterátoru zneplatniť.

Ak ťa zaujíma, čo od iterátora potrebuje napr. funkcia `sort`, v súbore `puch.h`<sup>2</sup> je dorobený `Puch::iterator` zo začiatku kapitoly. Program

```
1 #include <algorithm>
2 #include <iostream>
3
4 #include "puch.h"
```

<sup>2</sup><https://github.com/pocestny/programovanie/raw/master/materialy/puch.h>

```

5  using namespace std;
6
7  int main() {
8      Puch p(12, 11, 9);
9      sort(p.begin(), p.end());
10     for (auto it = p.begin(); it < p.end(); ++it) cout << *it << " ";
11     cout << endl;
12 }
```

vypíše 9 11 12. Ak sa pozrieš dovnútra, je tam jedna konštrukcia, ktorú som ti zatiaľ neukazoval: tzv. *type alias*. Ak napišeš<sup>3</sup> `using INT = unsigned long int;` tak `INT` bude všade v programe skratka za `unsigned long int`.

Tu je niekoľko užitočných funkcií z knižnice `algorithm`

<code>find</code>	<code>it find( it first, it last, const T&amp; val )</code> Vráti iterátor na prvý výskyt hodnoty <code>val</code> v intervale <code>[first, last)</code> alebo <code>last</code> ak sa tam <code>val</code> nenachádza.
<code>search</code>	<code>it search( it first, it last, it s_first, it s_last )</code> vráti iterátor na prvý výskyt postupnosti <code>[s_first, s_last)</code> v intervale <code>[first, last)</code>
<code>min_element</code>	<code>it min_element(it first, it last)</code> vráti iterátor na najmenší prvok z rozsahu <code>[first, last)</code>
<code>max_element</code>	<code>it max_element(it first, it last)</code> vráti iterátor na najväčší prvok z rozsahu <code>[first, last)</code>
<code>min</code>	<code>T min(T a, T b)</code> alebo <code>T min({T a1, ... , T an})</code> Vráti najmenší prvok (napr. <code>min(2, 3)</code> alebo <code>min({2, 3, 4, 5})</code> )
<code>max</code>	<code>T max(T a, T b)</code> alebo <code>T max({T a1, ... , T an})</code> Vráti najväčší prvok.
<code>fill</code>	<code>void fill(it first, it last, t val)</code> Vyplní interval <code>[first, last)</code> hodnotou <code>val</code>
<code>copy</code>	<code>it copy(it first, it last, it out)</code> Skopíruje interval <code>[first, last)</code> od iterátora <code>out</code> (musí tam byť miesto). <code>vector&lt;int&gt; b(a.size());</code> <code>copy(a.begin(), a.end(), b.begin());</code>

<sup>3</sup> Podobný alias používame, keď v každom programe píšeme `using namespace std;`; na to, aby sme si ušetrili písanie dvojbodiekov. Zápis s dvojbodkou sme už videli pri vnorených typoch, napr. `vector<int>::iterator` je typ `iterator` definovaný vvnútri typu `vector<int>`. Podobne je to aj s funkciami, `Puch::begin()` je celé meno funkcie `begin()` definovanej v type `Puch`. Ak píšeme metódy typu `Puch`, môžeme ju volať jednoducho iba `begin()`, ale pri volaní zvonka by sme potrebovali písť celé meno. Konštrukcia `namespace` umožňuje schovať viaceré typov, premenných a funkcií do jedného celku so spoločným menom. Je to hlavne kvôli tomu, aby sa vo velkých programoch zabránilo konfliktom, keby sa vyskytlo to isté meno na viacerých miestach. Keby si napísal

```

1  namespace Kacka {
2      int zobak;
3      void kvak() { cout << "kvak" << endl; }
4 }
```

tak v programe bude premenná `Kacka::zobak` a funkcia `Kacka::kvak()`. Nie je to ale typ ako pri `struct` (ani na konci definície nie je bodkočiarka), takže nemôžeš mať premennú `Kacka c;`

Všetky typy z STL sú v `namespace std`, takže by sme mali písť `std::vector`, `std::cout`, `std::endl` a pod. Ak napišeš `using namespace X` znamená to, že všetky veci z `X` chceš vedieť používať aj ich skratenými menami.

## Algoritmy v STL

---

copy_backward	to isté, ale skopíruje interval tak, že končí za iterátorom <code>out</code> <code>vector&lt;int&gt; b(a.size());</code> <code>copy(a.begin(), a.end(), b.end());</code>
merge	<code>it merge(it first1, it last1, it first2, it last2, it out)</code> Urobí operáciu <code>merge</code> na dvoch utriedených intervaloch a výsledok uloží od iterátora <code>out</code> (musí tam byť miesto). Vráti iterátor za posledný vložený prvok. <code>vector&lt;int&gt; c(a.size() + b.size());</code> <code>merge(a.begin(), a.end(), b.begin(), b.end(), c.begin());</code>
replace	<code>it replace(it first, it last, T&amp; old_value, T&amp; new_value)</code> Nahradí všetky výskyty hodnoty <code>old_value</code> hodnotou <code>new_value</code>
swap	<code>void swap(T&amp; a, T&amp; b)</code> Vymení hodnoty v premenných <code>a, b</code> .
reverse	<code>void reverse(it first, it last)</code> Otočí interval <code>[first, last]</code>
sort	<code>void sort(it first, it last)</code> Utriedi interval <code>[first, last]</code>
binary_search	<code>bool binary_search(it first, it last, const T&amp; val)</code> Zistí, či sa v utriedenom poli v rozsahu <code>[first, last]</code> vyskytuje hodnota <code>val</code>
lower_bound	<code>it lower_bound(it first, it last, const T&amp; val)</code> Vráti iterátor na prvú hodnotu v rozsahu <code>[first, last]</code> , ktorá je väčšia alebo rovná ako <code>val</code>
upper_bound	<code>it upper_bound(it first, it last, const T&amp; val)</code> Vráti iterátor na prvú hodnotu v rozsahu <code>[first, last]</code> , ktorá je väčšia ako <code>val</code>

---

## Funkcie ako parametre 25

Funkcie z knižnice `algorithm` sú užitočné, ale ak sa napr. vrátiš k úlohe 71, zistíš, že stále máš problém. Dajme tomu, že máš typ

```
1 struct Bod {  
2     double x, y;  
3 };
```

a chceš utriediť pole bodov podľa  $x$ -ovej súradnice. Ak skúsiš napísť

```
1 vector<Bod> a;  
2 sort(a.begin(), a.end());
```

dostaneš chybovú správu, že typ `Bod` nemá operátor `<`, podľa ktorého by sa mohol triediť. Môžeš to vyriešiť takto:

```
1 struct Bod {  
2     double x, y;  
3     bool operator<(Bod p) { return x < p.x; }  
4 };
```

ale vzápäť sa dostaneš do problémov, ak chceš teraz utriediť body podľa  $y$ -ovej súradnice. Hodil by sa nejaký mechanizmus, ako funkciu `sort` povedať, podľa čoho má triediť. Keďže to má byť všeobecná funkcia, najlepšie by bolo, keby sa jej, ako ďalší parameter, dala posunúť funkcia, ktorú má použiť na porovnávanie dvoch hodnôt namiesto operátora `<`.

Keď skompliluješ program, vytvorí sa z neho postupnosť príkazov pre procesor, ktorá sa pri spustení programu uloží do pamäte, podobne ako sme to mali pri riešení úlohy 64. Keď sa v programe volá funkcia, vytvorí sa v pamäti pre ňu nový svet, a procesor začne spracovávať príkazy na tom mieste pamäte, kde je program pre danú funkciu. Keď funkcia skončí, program pokračuje vo vykonávaní príkazov tam, kde prestal. Funkcia je teda, podobne ako premenná, tiež len postupnosť nul a jednotiek uložená niekde v pamäti, a teda, podobne ako premenná, má adresu. Táto adresa sa dá uložiť do premennej typu *pointer na funkciu*. Pretože pri vytváraní sveta funkcie treba vedieť, aké má parametre, pre každú kombináciu parametrov existuje iný typ, napr. typ *pointer na funkciu, ktorá má jeden parameter int a vracia typ bool*. Premenné typu *pointer na funkciu* sa zapisujú trochu špeciálnym spôsobom. Namiesto `typ nazov`; ako pri `int x`; sa napíše definícia funkcie, ale namiesto názvu má hviezdičku a názov premennej. Takže napr. `bool (*a)(int*)`; vytvorí premennú `a` ktorá bude typu *pointer na funkciu, ktorá má parameter int \* a vracia bool*. Priradíš do takejto premennej sa dá adresa funkcie, t.j. ak máš v programe napr. `bool parne(int* x) { return (*x) % 2 == 0; }`, tak môžeš napísť `a=&parne`; , prípadne ten ampersand môžeš aj vyniechať a písť rovno `a=parne`; . Premenná `a` teda bude pointer na funkciu, a jeho hodnota bude samotná funkcia, ktorú môžeš zavolať, napr. `(*a)(&x)`; (rovnako môžeš hviezdičku vyniechať a funguje aj zápis `a(&x)`; . Napokon, ak chceš viackrát použiť ten istý typ, je príjemné si naňho urobíť skratku pomocou `using`. V nasledujúcom programe je *Funkcia typ pointer na funkciu, ktorá má parameter int a vracia int*. Funkcia `urob` má parameter `int` a pointer na funkciu, ktorú zavolá. Pri volaní `urob(20, pridaj)` sa vypíše 30 a pri volaní `urob(20, uber)` sa vypíše 10.

```
1 #include <iostream>  
2 using namespace std;  
3  
4 using Funkcia = int (*)(int);  
5  
6 int pridaj(int x) { return x + 10; }  
7 int uber(int x) { return x - 10; }  
8  
9 void urob(int x, Funkcia f) {
```

## Funkcie ako parametre

---

```
10    cout << f(x) << endl;
11 }
12
13 int main() {
14     urob(20, pridaj);
15     urob(20, uber);
16 }
```

Tento mechanizmus funguje aj v jazyku C a častokrát sa používa v rôznych knižniciach na tzv. *callbacks*. Napríklad knižnica na prácu so sieťou môže mať funkciu, ktorá čaká, kým prídu nejaké dátá a potom zavolá callback, ktorý dostala ako parameter, na ich spracovanie.

V C++ sú aj ďalšie možnosti, ako odovzdať funkciu ako parameter, napr. pomocou tzv. *funktorov*. Funktor je len honosne znejúce slovo pre triedu, ktorá má definovaný operátor(). Tento program robí to isté, čo ten predchádzajúci:

```
1 #include <iostream>
2 using namespace std;
3
4 struct Pridaj {
5     int operator()(int x) { return x + 10; }
6 };
7
8 struct Uber {
9     int operator()(int x) { return x - 10; }
10};
11
12 template <typename T>
13 void urob(int x, T f) {
14     cout << f(x) << endl;
15 }
16
17 int main() {
18     urob(20, Pridaj());
19     urob(20, Uber());
20 }
```

Treba to čítať tak, že **Pridaj** je trieda, ktorá nemá žiadne premenné a má jedinú metódu, ktorou je operátor (). Ten berie ako parameter **int** a vracia **int**. Funkcia **urob** je šablóna, ktorá má prvý parameter **int** a druhý akéhokoľvek typu **T**. Ten typ ale musí mať príslušný operátor, aby sa dalo zavolať **f(x)**; keby si skúsil vyrobiť typ **urob<int>** kompilátor vyhlási chybu pri použití šablóny. V hlavnom programe je volanie **Pridaj()** konštruktor typu **Pridaj** (kedže si žiadnen nedefinoval, kompilátor si spravil default), ktorý vytvorí premennú typu **Pridaj** a pošle ju ako parameter do funkcie **urob<Pridaj>**, ktorú si vytvorí podľa šablóny. Pri volaní **urob<Pridaj>(20,Pridaj())** si môžem typ v šabloné odpustiť, kompilátor si ho vie domyslieť podľa typu parametra.

Ukážem ti ešte ďalšiu možnosť, ako dosiahnuť to isté, ktorá bude pre naše potreby častokrát najvhodnejšia. Sú ňou tzv. *lambda výrazy*<sup>1</sup>, ktoré umožňujú definovať funkciu formou výrazu a pracovať s pointrom na ňu. Ešte raz ten istý program, tento raz s lambdami:

```
1 #include <iostream>
2 using namespace std;
3
4 template <typename T>
```

<sup>1</sup>Nazvané podľa  $\lambda$ -kalkulu, čo je funkcionálny jazyk navrhnutý v 30tych rokoch minulého storočia Alonsom Churchom ([https://en.wikipedia.org/wiki/Lambda\\_calculus](https://en.wikipedia.org/wiki/Lambda_calculus))

```

5 void urob(int x, T f) {
6     cout << f(x) << endl;
7 }
8
9 int main() {
10    urob(20, [](int x) { return x + 10; });
11    urob(20, [](int x) { return x - 10; });
12 }
```

Skôr ako začneme hovoriť o lambdách, všimni si, že keď sme funkciu `urob` definovali pomocou šablóny, dá sa použiť s pointrom na funkciu, funktorom, aj lambdou, čo sú inak všetko rozdielne zvieratá.

Ako teda fungujú lambdy? Lambda výraz je *výraz*, t.j. ako som povedal úplne na začiatku: *príklad, ktorý sa počas behu programu vyráta a zistí sa jeho výsledok*. Výsledkom lambda výrazu je funkcia: dá sa predstaviť ako pointer na bezmennú funkciu (v skutočnosti to tak aj je), ale v programe má špeciálny typ `function`, definovaný v knižnici `functional`. Typ `function` je šablóna, ktorá sa dá použiť napr. takto:

```

1 #include <functional>
2 using namespace std;
3
4 int main() {
5     function<int(int)> f = [](int x) { return x + 42; };
6 }
```

`f` je typu *funkcia, ktorá má parameter `int` a vracia `int`*. Keď kompilátor vidí tento zápis, vyrobí v programe kód pre funkciu, ktorá vráti parameter zväčšený o 42 a pointer na túto funkciu priradí do premennej `f`. Hranaté zátvorky `[]` hovoria, že ide o lambda výraz. Potom nasledujú v normálnych zátvorkách parametre, ako pri bežnej funkcií a v kučeravých zátvorkách program tela funkcie. Keby bolo treba povedať aj návratový typ (napríklad by nebolo jasné, či sa má vracať `int` alebo `double`), dá sa to urobiť pomocou `->` takto:

```
1 function<double(int)> f = [](int x) -> double { return x + 42; };
```

V oboch prípadoch sa `f` dá zavolať ako normálna funkcia, t.j. `f(10)`.

Hranaté zátvorky nie sú iba značenie lambda výrazu, dovnútra sa píšu tzv. *captures*. Dajú sa predstaviť ako parametre, ktoré sa zafixujú v čase, keď sa funkcia vytvára (na rozdiel od normálnych parametrov, ktoré sa zafixujú až pri volaní). Dajme tomu, že chceme urobiť funkciu `zvacsovac`, ktorá dostane parameter `kolko` a vráti lambdu, ktorá bude mať jeden parameter a pri zavolaní ho zväčší o `kolko`. Prvý pokus by bol takýto:

```

1 function<int(int)> zvacsovac(int kolko) {
2     return [](int x) { return x + kolko; }; // !! chyba
3 }
```

Tu je problém v tom, že pri vyrábaní lambdy je premenná `kolko` lokálna premenná funkcie `zvacsovac`: vo vyrobenej lambde ju nevidno (ako v každej funkcií, aj v lambde vidno iba globálne a lokálne premenné). Keď premennú `kolko` uvedieš ako *capture*, t.j. správši

```

1 function<int(int)> zvacsovac(int kolko) {
2     return [kolko](int x) { return x + kolko; };
3 }
```

bude to fungovať takto: ak sa zavolá napr. `zvacsovac(10)(20)`, vytvorí sa svet pre funkciu `zvacsovac`, v ktorom bude lokálna premenná `kolko` mať hodnotu 10. Volanie `zvacsovac(10)` vráti bezmennú lambda funkciu, v ktorej hodnota `kolko` bude pevne fixovaná konštantou 10: funkcia teda pri svojom volaní robí to, že vráti svoj parameter zväčšený o 10. Pointer na túto funkciu sa vráti ako výsledok volania `zvacsovac(10)` a vzápäť sa funkcia zavolá, t.j. vytvorí sa nový svet, v ktorom premenná `x` bude nastavená na 20. Pretože `kolko`

## Funkcie ako parametre

je navždy fixná konštantá 10, bezmenná funkcia vráti 30. Keby nasledovalo volanie `zvacsovac(30)`, vráti sa nová, nezávislá, bezmenná funkcia, v ktorej je hodnota `kolko` fixovaná konštantá 30.

Ked sa teda ako `capture` napiše meno premennej, napr. `x`, jej aktuálna hodnota sa v lambde zoberie ako fixná konštantá. Ako `capture` sa dá napiisať aj `&x`, a vtedy sa ako fixná konštantá zoberie referencia (t.j. pointer) na `x`. Pozri si tento program:

```
1 int main() {
2     vector<int> a;
3     int kde = 0, kolko = 5;
4
5     auto f = [&a, &kde, kolko](int este) { a[kde] = kolko + este; };
6
7     a.resize(5);
8     vypis(a); // 0 0 0 0 0
9     f(1);
10    vypis(a); // 6 0 0 0 0
11    kde = 3;
12    f(2);
13    vypis(a); // 6 0 0 7 0
14    kolko = 1234;
15    f(2);
16    vypis(a); // 6 0 0 7 0
17 }
```

Hodnota premennej `f` je funkcia s parametrom typu `int`. V tejto funkcií je vektor `a` a premenná `kde` prevzatá referenciou, t.j. v čase, keď sa vyhodnocuje lambda-výraz (t.j. v našom prípade vtedy, keď sa robí priradenie do premennej `f`), sa zoberie adresa `a` a adresa `kde`, a vo vzniknutej funkcií sa každá odvolávka na `a` a `kde` bude odvolávať na premenné na týchto adresách. Na druhej strane, pri premennej `kolko` sa zoberie jej hodnota v čase vytvárania lambdy (t.j. 5), a všetky výskyty `kolko` sa nahradia touto hodnotou. Preto keď sa zmení premenná `kde`, tak nasledujúce volanie `f` bude meniť iný prvok vektora (lebo pri volaní zoberie hodnotu z adresy `kde`, ktorá sa nezmenila). Ale keď sa zmení premenná `kolko`, nič sa nestane, lebo `f` má namiesto `kolko` nastavenú hodnotu 5.

Treba, samozrejme, dávať pozor na to, že referencia môže prestať platíť, napríklad referencia na lokálnu premennú:

```
1 function<void()> quak(vector<int>& x) {
2     int z = 42;
3     // return [&x, &z]() { x[2] = z; }; !! problém
4     return [&x, z]() { x[2] = z; }; // toto je v poriadku
5 }
```

V prvom prípade sa vyrobi bezmenná lambda, do ktorej sa za `z` zafixuje adresa premennej `z` zo sveta funkcie `quak`. Lenže `quak` vzápäť skončí a na tej adrese môže byť čokoľvek. Na záver ešte poznámka: ak chceš do `capture` zoznamu dať všetky lokálne premenné, dá sa použiť `[=]` (resp. `[&]`, ak sa majú brať ich referencie). Ak sa lambda nachádza v rámci triedy, netreba zabudnúť dať do `capture` zoznamu `this`, inak premenné z triedy nebudú viditeľné (lambda je nezávislá funkcia<sup>2</sup>).

Lambda výrazy sú často vhod pri práci s algoritmami v STL. Veľa funkcií má verzie s dodatočným parameterom, ktorý je funkcia na porovnávanie prvkov. Špeciálne nás zaujíma funkcia `sort`. Videl si tvar `sort(a.begin(), a.end())`, ale dá sa pridať aj tretí parameter, ktorý musí byť funkcia, ktorá zoberie dva prvky (takého typu, ako sú v kontajneri, ktorý sa triedi), a vráti `true`, ak je prvý prvok menší. Napríklad:

<sup>2</sup>Presnejšie povedané pre fanjšmekrov: lambda je v skutočnosti iba skrátený zápis pre funktor. Ak máš v programe premennú `int kvak=17` a zavoláš `auto zaba = [kvak](int x){return kvak+x;}`, komplítačka vyrobí nejaký typ, napr. `struct lambda_e14159265e {int kvak; operator()(int x){return kvak+x;}};`, povie si, že `auto zaba` bude `lambda_e14159265e` zaba a uloží tam premennú typu `lambda_e14159265e`, ktorej hodnotu zaba.kvak nastaví na 17.

```

1 int main() {
2     vector<int> a;
3     for (int i = 0; i < 6; i++)
4         a.push_back(2 + 2 * (i / 2) - i % 2);
5
6     vypis(a); // 2 1 4 3 6 5
7     sort(a.begin(), a.end());
8     vypis(a); // 1 2 3 4 5 6
9     sort(a.begin(), a.end(), [](int a, int b) { return a > b; });
10    vypis(a); // 6 5 4 3 2 1
11 }
```

Ak je treba posieláť funkciu ako parameter, treba sa rozhodnúť, či to bude pointer na funkciu, lambda, prípadne nejaký typ funkktora. Táto dilema sa častokrát rieši šablónou. Ak si urobíš napr. šablónu

```

1 template <typename F>
2 void rob(F f) { cout << f(10) << endl; }
```

tak komplilátor bude vedieť vyrobiť funkcie `rob<...>` pre všetky tri verzie. Navyše sa veľakrát nemusíš staráť o to, aký presne typ má tvoj parameter, lebo komplilátor to pri použití šablóny častokrát zistí. Takže môžeš napr. napísat'

```

1 int sedem(int x) { return 7; }
2
3 struct Uberac {
4     Uberac() { y = 42; }
5     int operator()(int x) { y -= x; return y; }
6     int y;
7 };
8
9 int main() {
10     rob( [](int x) { return x + 5; } ); // zavolá sa rob<function<int(int)> >
11     rob( &sedem ); // zavolá sa rob<int (*)(int)>
12
13     Uberac u;
14     rob( u ); // zavolá sa rob<Uberac>
15 }
```

**Úloha 87.** Na vstupe je  $n$  prirodzených čísel. Zorad ich tak, aby najprv nasledovali všetky párne čísla utriedené od najmenšieho po najväčšie a potom všetky nepárne čísla utriedené od najväčšieho po najmenšie. Napríklad pre vstup `7 3 2 6 4 1 5` je výstup `2 4 6 7 5 3 1`.

**Úloha 88.** Na vstupe je číslo  $n$  a potom pole  $n$  čísel. Napiš program, ktorý zistí, ktoré číslo sa najčastejšie opakuje. Napr. pre pole `3 5 7 5 3 2 1 5 6 5` je odpoveď `5`.

**Úloha 89.** Na vstupe je  $n$  prirodzených čísel. Pre každé z nich (v tom poradí, ako sa nachádzajú na vstupe) vypíš, kol'káte v poradí by bolo, keby sa utriedili. Napr. pre vstup `4 10 2 20 1 5 3 6` je výstup `4 7 2 8 1 5 3 6`.

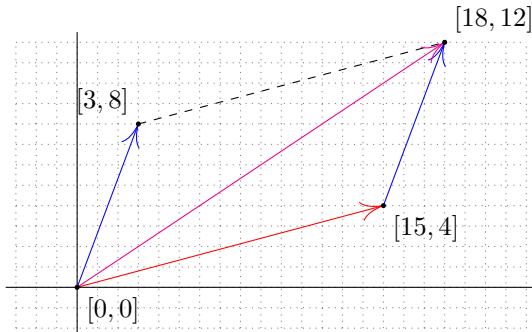
**Úloha 90.** Máme displej, ktorý môže ukazovať kladné aj záporné čísla. Na začiatku ukazuje 0. Na vstupe sú čísla  $n$ ,  $k$  a potom  $n$  čísel, ktoré udávajú, o kolko sa hodnota na displeji zmenila každú nasledujúcu sekundu. Napiš program, ktorý vypíše počet takých dvojíc sekúnd  $i, j$ , že  $i < j \leq k + i$  (t.j.  $i$  a  $j$  sú od seba vzdialé najviac  $k$ ), že displej po  $i$  sekundách a po  $j$  sekundách ukazoval rovnakú hodnotu. Napríklad pre vstup  $n = 9$ ,  $k = 4$  a čísla `2 1 2 -2 1 -4 3 1 -4`, bude po prvej sekunde na displeji hodnota 2, po druhej hodnota 3 atď, celkovo budú hodnoty na displeji postupne `0 2 3 5 3 4 0 3 4 0`. Rovnaké hodnoty sú po nultej, 6. a 9. sekunde (vtedy je hodnota 0), po 2., 4. a 7. sekunde (vtedy je hodnota 3) a po 5. a 8. sekunde (vtedy je hodnota 4). Dvojice, koré od seba nie sú ďalej ako 4 sú preto (2, 4), (4, 7), (5, 8) a (6, 9). Odpoveď je preto 4. Podobne pre vstup  $n = 5$  a  $k = 5$  a čísla `0 0 0 0` je odpoveď 15 (všetky dvojice sú dobré).

## 26 Matematické intermezzo: komplexné čísla

Vieš, že odmocnina z  $x$  (označovaná  $\sqrt{x}$ ) je také číslo  $y$ , že  $y^2 = x$ . Majú všetky čísla odmocninu? Ako sa to vezme. Takú  $\sqrt{2}$  starí Gréci za číslo nepovažovali: vedeli totiž, že  $\sqrt{2}$  sa nedá vyjadriť ako žiadnen zlomok. Ako to mohli vedieť? No nech by  $\sqrt{2} = p/q$ . Keby  $p$  aj  $q$  bolo párnne, tak môžeme obidve vydeliť dvomi a rovnosť stále platí, takže keby sa  $\sqrt{2}$  dala zapísat zlomkom, dá sa zapísat aj takým, že najviac jedno z čísel  $p, q$  je párnne. Keď si rovnosť vynásobíme samu sebou, máme  $2 = p/q \cdot p/q = p^2/q^2$  a preto  $p^2 = 2q^2$ . Keďže  $p^2$  je párnne, aj  $p$  musí byť párnne<sup>1</sup>. Keďže  $p$  je párnne,  $p^2$  je deliteľné 4. Ale  $q$  je nepárne, preto aj  $q^2$  je nepárne, a preto  $2q^2$  nie je deliteľné 4.

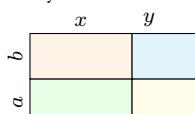
Medzičasom sa  $\sqrt{2}$  dostala medzi čísla (hovorí sa, že je *iracionálne číslo*); za číslo sa dnes považuje kde-čo (napr. si videl takú divočinu ako  $\pi$ ). Môžeme teda povedať, že všetky kladné čísla majú odmocniny. Ale čo záporné? Tie ich mať nemôžu, lebo či už je  $y$  kladné alebo záporné,  $y^2$  je kladné vždy. Takže žiadne  $y$  nemôže byť, trebárs,  $\sqrt{-1}$ . Ale v matematike máme výhodu: ak niečo neexistuje, môžeme si to vymyslieť. Takže si môžeme predstaviť, že nejakým spôsobom  $\sqrt{-1}$  existuje. Nebude to žiadne číslo, ako ich doteraz poznáme (hovoríme im *reálne*), ale bude to *niečo*. Nazvem si ho  $i$ . Chcem, aby sa  $i$  pri násobení a sčítaní správalo, ako sa na čísla patrí, takže napr.  $i + 2 = 2 + i$ ,  $i + i + i = 3i$ ,  $3(i + 7) = 3i + 21$  a pod. so všetkým. Keď mám výraz zložený zo sčítania, násobenia a zátvoriek, vždy ho viem upraviť na tvar  $x + yi$  pre nejaké  $x, y$ . To je preto, že  $i^2 = -1$ , napr.  $(3i + 4 - i)(i + i - 6) + 7i - 8 = (2i + 4)(2i - 6) + 7i - 8 = 4i^2 - 12i + 8i - 24 + 7i - 8 = 3i - 28$ . Zjednodušovanie, samozrejme, platí aj ďalej,  $i^3 = i^2i = -i$ ,  $i^4 = i^2i^2 = -1 \cdot -1 = 1$ , a tak ďalej.

Reálne čísla si zvykneme kresliť na číselnú os. Kam ale nakresliť  $i$ ? na číselnej osi už nie je miesto. Podľme preto do roviny: číslo  $x + yi$  si nakreslíme do bodu so súradnicami  $[x, y]$ , takže napr.  $i$  bude nakreslené v bode  $[0, 1]$ . Keď si rovnako dobre predstavíme číslo  $x + yi$  ako šípku, ktorá ide z bodu  $[0, 0]$  do bodu  $[x, y]$ , tak sčítanie funguje veľmi prirodzene: zoberiem jedno číslo (šípku) a priložím ho na koniec druhého. Napr.  $(15 + 4i) + (3 + 8i) = 18 + 12i$  vidno takto:

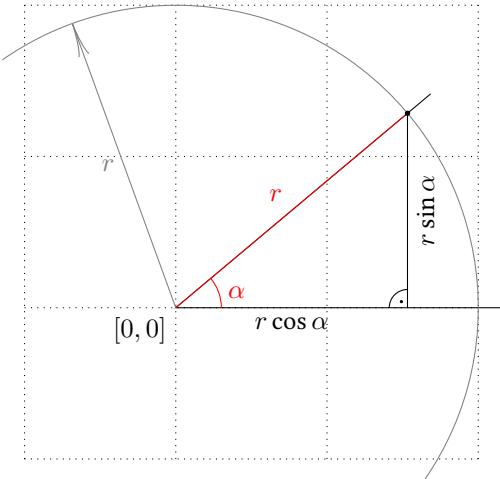


Ako si ale predstaviť násobenie? Napr.  $(1 + 2i)(3 + i) = 3 + i + 6i + 2i^2 = 1 + 7i$ . Lepšie to vidno v tzv. *polárnych súradničiach*. Keď chceme jednoznačne určiť bod v rovine, namiesto  $x$ -ovej a  $y$ -ovej súradnice nám stačí pamätať si, pod akým uhlom a ako ďaleko treba ísť. V kapitole 17 sme hovorili o funkciách  $\sin$  a  $\cos$ , takže vieš, že bod, do ktorého sa ide pod uhlom  $\alpha$  do vzdialenosťi  $r$  má súradnice  $[r \cos \alpha, r \sin \alpha]$ , a teda reprezentuje číslo  $r \cos \alpha + ir \sin \alpha$ .

<sup>1</sup>Lebo súčin dvoch nepárných čísel je vždy nepárny. Tu má zmysel nakresliť obrázok, ktorý budeme za chvíľu aj tak potrebovať:



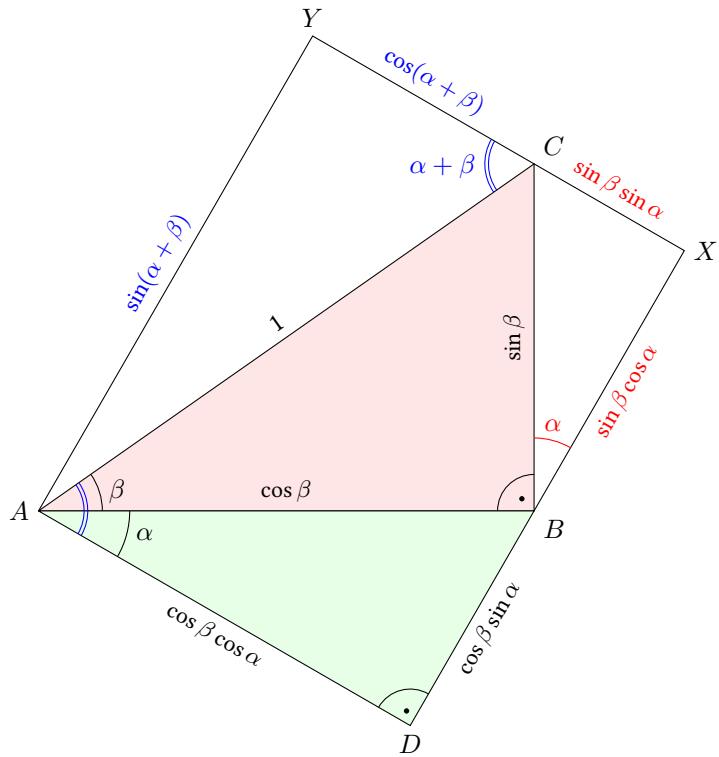
Obdĺžnik na obrázku má obsah  $(a + b)(x + y)$  a skladá sa zo štyroch menších obdĺžnikov, preto  $(a + b)(x + y) = ax + ay + bx + by$ . Keď násobíme dve nepárne čísla, mám  $(2r + 1)(2s + 1) = 4rs + 2r + 2s + 1$ , čo je nepárne.



Zoberme si teraz súčin dvoch čísel  $(r_1 \cos \alpha_1 + ir_1 \sin \alpha_1)(r_2 \cos \alpha_2 + ir_2 \sin \alpha_2)$ . Po zjednodušení z toho dostaneme

$$r_1 r_2 (\cos \alpha_1 \cos \alpha_2 - \sin \alpha_1 \sin \alpha_2) + ir_1 r_2 (\sin \alpha_1 \cos \alpha_2 + \cos \alpha_1 \sin \alpha_2). \quad (26.1)$$

Môžeme to ďalej nejak zjednodušíť? Začneme s pravouhlým  $\triangle ABC$  s uhlom  $\beta$ . Ak  $|AC| = 1$ , tak  $|AB| = \cos \beta$  a  $|BC| = \sin \beta$ . Pod úsečkou  $AB$  zostrojíme pravouhlý  $\triangle ABD$  s uhlom  $\alpha$ , takže  $|BD| = \cos \beta \sin \alpha$ ,  $|AD| = \cos \beta \cos \alpha$ . Nakoniec dorobíme ohraničujúci obdĺžnik  $ADX$ .



Vidno, že  $\angle DBA = 90^\circ - \alpha$ , preto  $\angle XBC = \alpha$ . Pretože  $AD \parallel XY$ , je  $\angle CAD = \angle ACY = \alpha + \beta$ . Z pravouhlého trojuholníka  $\triangle BXC$  doplníme, že  $|BX| = \sin \beta \cos \alpha$  a  $|XC| = \sin \beta \sin \alpha$ . Podobne z pravouhlého  $\triangle CYA$  doplníme, že  $|CY| = \cos(\alpha + \beta)$  a  $|AY| = \sin(\alpha + \beta)$ . Keď si teraz porovnáme, čo máme napísane na stranách obdĺžnika, zistíme, že

$$\sin(\alpha + \beta) = \cos \beta \sin \alpha + \sin \beta \cos \alpha$$

## Matematické intermezzo: komplexné čísla

---

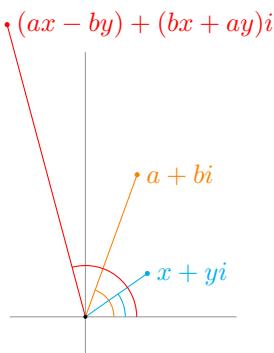
a

$$\cos(\alpha + \beta) = \cos \beta \cos \alpha - \sin \beta \sin \alpha$$

Ked si to porovnáš s (26.1), zistíš, že

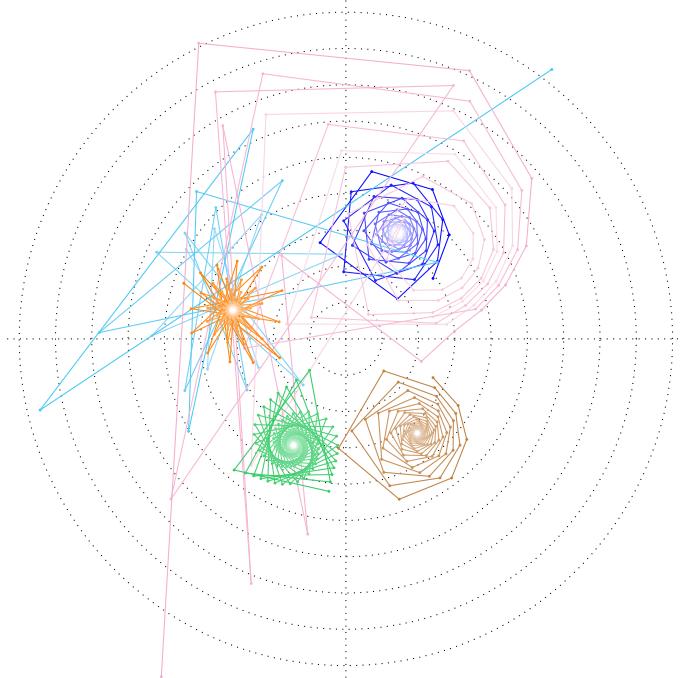
$$(r_1 \cos \alpha_1 + ir_1 \sin \alpha_1)(r_2 \cos \alpha_2 + ir_2 \sin \alpha_2) = r_1 r_2 \cos(\alpha_1 + \alpha_2) + ir_1 r_2 \sin(\alpha_1 + \alpha_2)$$

Inými slovami, ak vynásobíme dve komplexné čísla, dostaneme číslo, ktorého uhol v polárnych súradničach je súčtom uhlov a dĺžka je súčinom dĺžok. V kartézskych súradničach by sme napísali  $(a + bi)(x + yi) = (ax - by) + (bx + ay)i$ , špeciálne  $(a + bi)^2 = (a^2 - b^2) + 2abi$ .



## Projekt: Mandelbrotova množina

Matematik menom Benoit Mandelbrot sa hral takúto hru<sup>2</sup>: zbral si nejaké komplexné číslo  $c$  a začal počítať postupne  $c, c^2 + c, (c^2 + c)^2 + c, ((c^2 + c)^2 + c)^2 + c, \dots$  a tak ďalej; vždy ďalšie (komplexné) číslo získal tak, že to doterajšie umocnil na druhú a prirátal  $c$ . Niektoré čísla stále poskakovali okolo toho istého miesta, iné začali rást a rástli donekonečna:



Mandelbrotu zaujímalo, ktoré čísla sú také, že nikdy neujdú z kruhu s polomerom  $R$ .

<sup>2</sup>V skutočnosti sa tú hru ako prví hrali Robert Brooks a Peter Matelski v r. 1978, ale Mandelbrotovi sa tak veľmi páčila, že je pomenovaná po ňom.

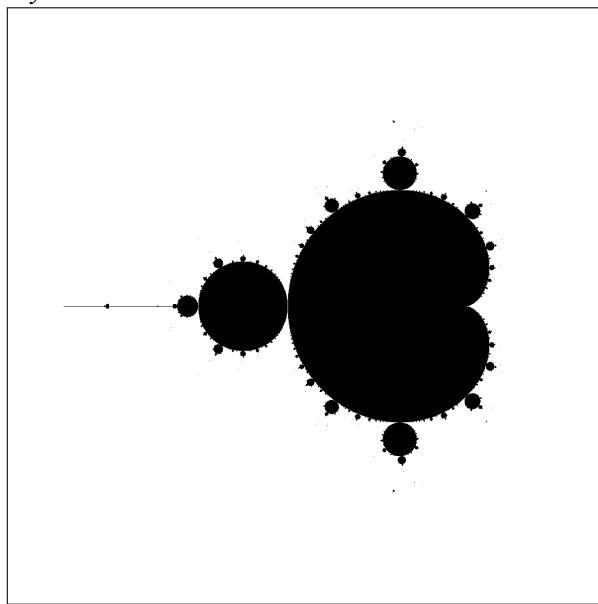
**Úloha 91.** Na vstupe sú čísla  $R$  a  $N$ . Potom nasledujú komplexné čísla v kartézskych súradničach, t.j. číslo  $x + iy$  je zapisané dvojicou  $x$   $y$ . Vstup končí číslo  $0$   $0$ . Napiš program, ktorý pre každé číslo vypíše **ujde** alebo **neujde**. Číslo ujde, ak počas prvých  $N$  Mandelbrotových iterácií jeho veľkosť (t.j.  $\sqrt{x^2 + y^2}$ ) prekročí  $R$ . Napr.

```
100000 100000
0.36 0.25
-0.66 0.37
0.36 0.09
-0.67 0.23
-0.07 -0.63
0.36 -0.16
0 0
```

```
neujde
ujde
ujde
neujde
neujde
neujde
```

Mandelbrot sa ďalej rozhodol, že si skúsi množinu nakresliť: čierne body budú čísla, ktoré neujdú a biele čísla, ktoré ujdú.

**Úloha 92.** Na vstupe sú čísla  $R$ ,  $N$  ako z minulej úlohy. Navyše čísla  $d$ ,  $x$ ,  $y$ ,  $m$ . Napiš program, ktorý vyrobí obrázok rozmerov  $n \times n$  zachytávajúci výsek komplexnej roviny so stredom  $x + iy$  a dĺžkou strany  $m$ . Napr. pre  $x = -0.65$ ,  $y = 0$ ,  $m = 0.3$  by si mal dostať obrázok



Čiernym bodom, ktoré si dostał na obrázku, sa hovorí Mandelbrotova množina a má rôzne vlastnosti, ktoré sa matematikom páčia: napr. je súvislá a je to *fraktál*, t.j. ak sa pozrieš s väčším priblížením, nájdeš v nej zmenšené kópie celej množiny, aj so zmenšenými kópiami ....

Možno si si ale pri hraní sa s parametrami  $R$  a  $N$  všimol, že pre veľké počty iterácií program začína bežať dosť dlho. Na zrýchlenie programov je spravidla najúčinnejšie vymyslieť rýchlejší algoritmus, ale ak to nejde, dá sa povedať kompilátoru, aby sa viac snažil. Aj `g++` aj `clang++` majú prepínač `-O` (ako optimalizácia). Ak teda pri komplilovaní namiesto `g++ program.cc -o program` napišeš `g++ -O3 program.cc -o program`, kompilátor bude pracovať dlhšie, lebo sa bude snažiť prepísať tvoj program tak, aby robil to isté, ale bežal čo najrýchlejšie. Kompilátory sú celkom šikovné, takže niekedy to dokáže spraviť aj  $10\times$  rýchlejšie.

A ešte jedna poznámka. V knižnici `#include<complex>` je definovaný typ `complex` na príjemnejšiu prácu s komplexnými číslami. Je to šablóna, ktorá ako parameter dostane typ súradníc, napr. `complex<double>`. Komplexné čísla môžeš sčítavať a násobiť ako normálne čísla, načítavať zo vstupu v tvare napr. `(-0.65, 0)` a

## Matematické intermezzo: komplexné čísla

---

pod. Ak máš `complex<double>`  $z$ , tak  $z.\text{real}()$  je  $x$ -ová (reálna) súradnica,  $z.\text{imag}()$  je  $y$ -ová (imaginárna) súradnica,  $\text{norm}(z)$  je  $x^2 + y^2$ ,  $\text{abs}(z)$  je veľkosť (t.j.  $\sqrt{x^2 + y^2}$ ) a  $\arg(z)$  je uhol v radiánoch.

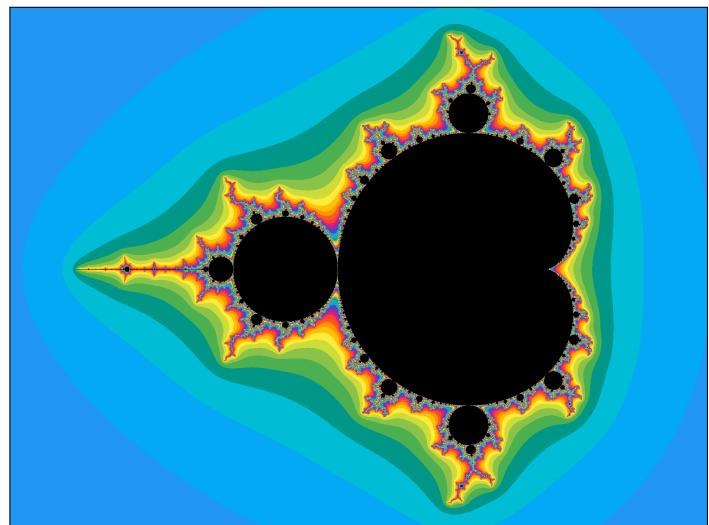
Teraz podme do obrázku pridať trochu farby. Body, ktoré patria do množiny (t.j. sú čierne), necháme čierne. Biele body sú tie, ktoré ušli cez  $R$  po menej ako  $N$  iteráciách. Môžeme im dať farbu podľa počtu iterácií, kedy prekročili  $R$ . Ako konkrétnie priraďovať farbu? Budeme mať pole (vektor) `paleta` prednastavených farieb a farbu príslušného pixela v obrázku určíme ako `paleta[iter % paleta.size()]` kde `iter` je počet iterácií, kedy veľkosť prekročila  $R$ .

Ked dizajnéri navrhujú palety a farebné schémy, často pracujú so zápisom farby pomocou reťazca (tzv. *hex reťazec*). Takisto ho podporujú rôzne "color-picker" nástroje. Skladá sa zo siedmich (RGB) alebo deviatich (RGBA) znakov `#RRGGBB`, pričom každá z dvojíc `RR`, `GG`, `BB` je číslo od 0 do 255 zapísané v šestnásťkovej sústave<sup>3</sup>.

**Úloha 93.** Napiš program, ktorý zo vstupu prečíta veľkosť palety, potom príslušný počet hex reťazcov farieb, potom parametre  $R$ ,  $N$ , rozmer obrázka, stred (komplexné číslo) a parameter  $m$  ako v predchádzajúcich úlohach a výrobí farebný obrázok. Napr. pre tento vstup by obrázok vyzeral takto:

```
16
#f44336 #e81e63 #9c27b0 #673ab7
#3f51b5 #2196f3 #03a9f4 #00bcd4
#009688 #4caf50 #8bc34a #cddc39
#ffeb3b #ffc107 #ff9800 #ff5722

1000000 100000
1600 1200
(-0.65, 0) 0.3
```



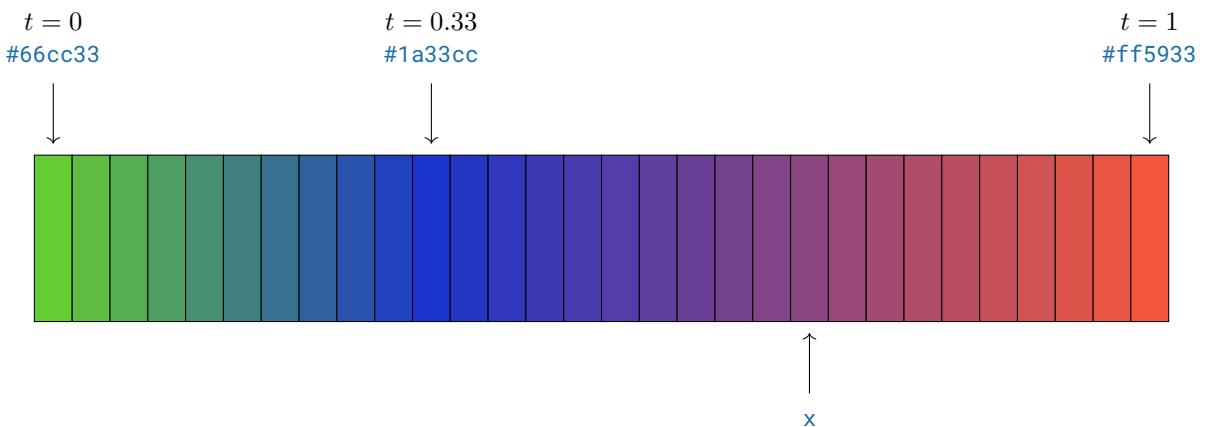
Pekné, ale stále trochu vadí, že prechody medzi farbami sú náhle. Chcel by som, aby zafarbenie bolo plynulé. Na to treba vyriešiť dva problémy: ako výrobiť "paletu", v ktorej sú plynulé prechody a potom ako nahradí počet iterácií niečím, čo sa plynulejšie mení.

Prvý problém vyriešime pomocou *interpolácie* farieb. Interpolácia je vlastne proporcionálne zmiešavanie: ak mám čísla  $a$  a  $b$ , a parameter  $t$ , ktorý vyjadruje časť (t.j.  $0 \leq t \leq 1$ ), môžem ich zmiešať tak, že zoberiem  $t$ -tinu z  $a$  a zvyšok (t.j.  $1 - t$ ) doplním z  $b$ : dostanem  $ta + (1 - t)b$ . Ak je  $t = 0$ , mám  $b$ , ak je  $t = 1$ , mám  $a$  a ak sa  $t$  plynule mení, aj výsledná hodnota plynule prechádza z  $b$  do  $a$  (pre  $t = 1/2$  mám priemer). Pre dve farby môžeme urobiť to isté: nezávisle interpoláciu v červenej, zelenej a modrej zložke<sup>4</sup>. Spravme triedu `Gradient`, ktorá bude obsahovať pole zlomových bodov. Zlomový bod má pozíciu (z rozsahu 0...1) a farbu. `Gradient` potom pre dané `x` zistí<sup>5</sup>, bedzi ktorými zlomovými bodmi sa nachádza, a vráti príslušnú interpoláciu. Gradient s troma zlomovými bodmi vyzerá napríklad takto:

<sup>3</sup>Sestnásťková (*hexadecimálna*) sústava používa cifry 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f (niekedy môžu byť aj veľké písmená). Posledná cifra udáva počet jednotiek, predposledná počet šestnásťok, predpredposledná počet  $16 \cdot 16 = 256$ -tiek atď. Takže napr. 2b v šestnásťkovej sústave je  $2 \cdot 16 + 11 = 43$ .

<sup>4</sup>Len pre informáciu: existuje veľa rôznych farebných modelov, ktoré tiež reprezentujú farbu pomocou niekolkých (spravidla) troch čísel, ale tie čísla nie sú "červená", "zelena", "modrá", ale napr. "farba", "sýtosť", "jas" a pod. Práve RGB model nie je velmi vhodný na interpoláciu po zložkách, na to sú lepšie kolorimetrické modely ako XYZ. Ale na naše účely to stačí v pohode.

<sup>5</sup>napr. binárny vyhľadávaním, aj keď pre mälo zlomových bodov je binárne vyhľadávanie overkill



S pomocou knižnice `cmath`, v ktorej je funkcia `fmod`: zvyšok po delení, ale pre desatinné čísla (t.j. `fmod(t, 1)` je desatinná časť z `t`) môže implementácia vyzeráť napr. takto:

```

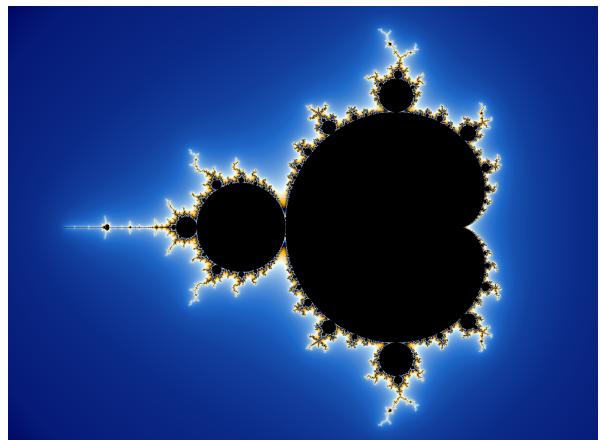
1 struct Stop {
2     double t;
3     RGBA f;
4 };
5
6 struct Gradient {
7     vector<Stop> stops;
8
9     Gradient(const vector<Stop>& _stops) { stops = _stops; }
10
11    RGBA operator()(double t) {
12        t=fmod(t,1);
13        RGBA x;
14        auto it = lower_bound(stops.begin(), stops.end(), Stop{t, {}},
15                               [] (Stop a, Stop b) { return a.t < b.t; });
16
17        if (it == stops.begin()) return stops[0].f;
18        RGBA a = (it)->f, b = (it - 1)->f;
19        t = (t - (it - 1)->t) / (it->t - (it - 1)->t);
20
21        x.r = (Byte)(t * a.r + (1 - t) * b.r);
22        x.g = (Byte)(t * a.g + (1 - t) * b.g);
23        x.b = (Byte)(t * a.b + (1 - t) * b.b);
24        return x;
25    }
26 }
```

Druhý problém je, akú hodnotu použiť namiesto počtu iterácií, aby sa menila spojitejšie. Po troche hrania sa mi osvedčilo toto: pre každý pixel si zapamätám počet iterácií a jeho veľkosť v čase keď prekročil hranicu (kvôli väčšej plynulosťi beriem logaritmus z veľkosti; v knižnici `cmath` je na to funkcia `log`). Potom, keď som mal hotový celý obrázok, tak som zistil všetky počty iterácií, ktoré sa vyskytli a pre každý počet iterácií našiel maximálnu a minimálnu hodnotu veľkosti. Každý pixel potom dostał okrem počtu iterácií aj desatinnú časť, ktorá závisela od jeho veľkosti (relativne k maximálnej a minimálnej veľkosti pre tento počet iterácií).

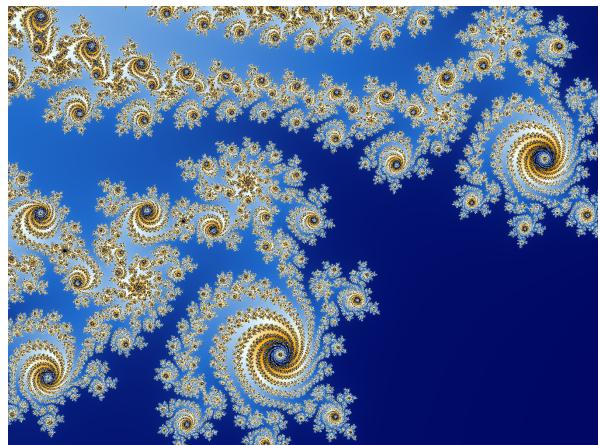
```

1 Gradient g(
2 {{0.0, RGBA(0, 7, 100)},
3 {0.16, RGBA(32, 107, 203)},
4 {0.42, RGBA(237, 255, 255)},
5 {0.6425, RGBA(255, 170, 0)},
6 {0.8575, RGBA(0, 2, 0)},
7 {1.0, RGBA(0, 7, 100)}});

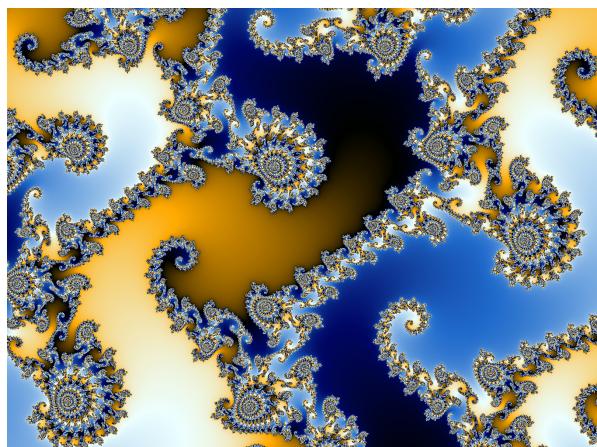
```



stred  $(-0.6928223, 0.3282697)$   
 $m = 508.9334$



stred  $(-0.59990625, -0.4290703125)$   
 $m = 1500$



stred  $(-0.743643887035763, 0.13182590421259918)$   
 $m = 4.3e14$



stred  $(-0.65550685, 0.3783525)$   
 $m = 236.2$

**Úloha 94.** Pohraj sa s Mandelbrotovou množinou.

Pri farbení Mandelbrotovej množiny v závere minulej kapitoly som robil túto vec: "...pre každý počet iterácií [som] našiel maximálnu a minimálnu hodnotu ...". Ak si sa pozrel do môjho programu, tak si videl, že som trochu cheatoval a použil som typ `map` z STL, o ktorom som ti doteraz nehovoril. Ale v tejto kapitole to chceme napraviť. Začnime jednoduchou rozvíčkou:

**Úloha 95.** Na vstupe je číslo  $n$ . A potom postupnosť príkazov. Každý príkaz je tvaru `! x alebo ? x` a posledný príkaz je `#`. Predpokladajme, že máme  $n$  žiaroviek očíslovaných  $1, \dots, n$ , ktoré sú na začiatku vypnuté. Príkaz `? x` znamená, že sme prepĺň žiarovku číslo  $x$  (ak bola vypnutá, bude zapnutá a naopak). Pre každý príkaz `? x` treba vypísať `svieti` alebo `nesvieti` podľa toho, v akom stave je práve žiarovka číslo  $x$ .

Jednou z možností, ako túto úlohu vyriešiť, bolo urobiť si vektor  $n$  hodnôt typu `bool` tak, že v `a[i]` si budeme udržiavať stav  $i$ -tej žiarovky. Predpokladajme teraz ale, že by žiarovky neboli očíslované 1 až  $n$ , ale že by mali iba výrobné čísla. Vstup by mohol vyzerať napríklad:

```
! 3736272
! 843984
? 3736272
! 84872634
! 72876
? 99
? 843984
#
#
```

Problém je, že nevieme, aké výrobné čísla sa na vstupe objavia. Nemôžme si držať v pamäti pole pre všetky možné čísla, takže chceme mať zapamätané iba tie žiarovky, ktoré sme už videli. Teraz ale stoja proti sebe dve veci: ak chceme nájsť stav žiarovky s číslom `id`, hodilo by sa mi mať žiarovky utriedené podľa čísla: mohol by som použiť binárne vyhľadávanie, aby som zistil, či `id` svieti. Ak ich utriedené nemám, musím zakaždým prejsť všetky, čo trvá dlho. Na druhej strane, keď sa objaví nová žiarovka, ľahko ju pridám na koniec, ale potom nebudú utriedené. Musel by som všetky žiarovky v poli poposúvať, čo zase trvá dlho. Riešenie je namiesto pola (vektora) použiť dátovú štruktúru, ktorá nemá všetky dáta v jednom kuse pamäte, ale vo viacerých, ktoré na seba navzájom ukazujú pointrami. Tým sa bude dať docieliť, že pri vkladaní nového prvku bude namiesto posúvania všetkých hodnôt stačiť presmerovať zopár pointrov.

Aby som ti ukázal, ako také štruktúry fungujú, začnime s jednoduchším príkladom a k žiarovkám sa vrátíme na konci kapitoly.

**Úloha 96.** Úradník má kartotéku, kde má uložené spisy očíslované  $0, 1, \dots, n - 1$  (na začiatku v tomto poradí). Keď potrebuje nájsť spis  $i$ , začne prehľadávať zaradom od začiatku kartotéky a vždy prečíta číslo spisu, až kým nenájde ten s číslom  $i$ . Ten potom použije a uloží na začiatok kartotéky. Napiš program, ktorý má na vstupe čísla  $n, m$  a potom  $m$  čísel spisov (medzi  $0$  a  $n - 1$ ). Pre každé číslo spisu vypíš riadok, v ktorom sú čísla spisov, ktoré úradník prečítal pri hľadaní. Napríklad

```
6 4
5
3
4
1
```

```
0 1 2 3 4 nasiel som 5!
5 0 1 2 nasiel som 3!
3 5 0 1 2 nasiel som 4!
4 3 5 0 nasiel som 1!
```

Na riešenie použijeme dátovú štruktúru *spájaný zoznam* (*linked list*). Každý prvok zoznamu je uložený v samostatne alokovanej pamäti a má pointer na nasledujúci prvok. Spravíme si typ

## Vyhľadávacie stromy: <set> a <map> v STL

```
1 struct Node {  
2     int val;  
3     Node *nxt;  
4     Node(int _val = -1, Node *_nxt = nullptr) {  
5         val = _val;  
6         nxt = _nxt;  
7     }  
8 };
```

v ktorom bude jeden prvok zoznamu<sup>1</sup>. V programe budeme mať zoznam zapamätaný iba pointrom na jeho začiatok `Node *head`; Zoznam [0 1 2 3 4 5] by v pamäti vyzeral takto:



Vyrobíme ho jednoducho:

```
1 head = nullptr;  
2 for (int i = 0; i < n; i++)  
3     head = new Node(n - 1 - i, head);
```

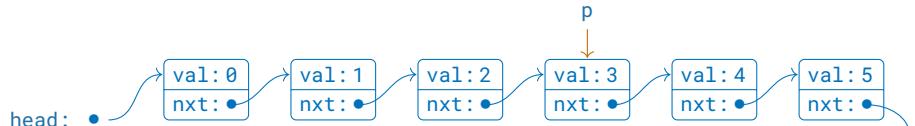
V cykle vždy pri vyhodnocovaní pravej strany priradenia vyrobíme novú premennú typu `Node`, v ktorej nastavíme `val` na správnu hodnotu a `nxt` na rovnaké miesto, ako práve ukazuje `head`. V priradení potom nastavíme `head`, aby ukazoval na túto novú premennú.



Povedzme, že už máme vyrobený zoznam a máme v ňom nájsť hodnotu `x`<sup>2</sup>. Správím si premennú `Node *p = head`; s ktorou pôjdem po zozname

```
1 while (p->val != x) p = p->nxt;
```

Teraz budem v situácii, že `p` ukazuje na hľadaný prvok:



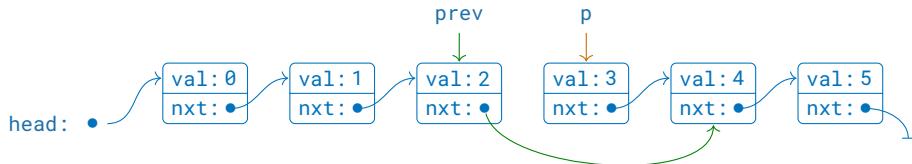
Ako presunúť nájdený prvok na začiatok zoznamu? Najprv ho vypojím a potom napojím na začiatok. Na vypojenie potrebujem nastaviť `nxt` v predchodecovi na `p->nxt`. Preto si upravím cyklus, ktorým prechádzam zoznam tak, aby som našiel aj predchodec `p`:

```
1 Node *p = head, *prev = nullptr;  
2 while (p->val != x) {  
3     prev = p;  
4     p = p->nxt;  
5 }
```

<sup>1</sup>Možno ťa zaskočilo, že súčasťou typu `Node` je premenná `nxt`, ktorá je pointer na typ `Node`. Nie je v tom ale nič zvláštne – pointer je iba číslo adresy v pamäti. Typ pointra (v našom prípade `Node *`) iba hovorí komplilátoru, že na adrese uloženej v premennej `nxt` má očakávať premennú typu `Node`. Pretože všetky pointre sú rovnako dlhé, komplilátoru nevadí, že typ `Node` zatiaľ nie je pripravený, stačí mu vedieť, kolko miesta treba v pamäti vyhradzovať.

<sup>2</sup>a sme si istí, že `x` sa v zozname nachádza, takže nemusíme kontrolovať, či sme prišli na koniec zoznamu

```
6 if (prev != nullptr)
7     prev->nxt = p->nxt;
```



Teraz môžem presunúť **p** na začiatok podobne, ako pri vytváraní zoznamu:

```
1 p->nxt = head;
2 head = p;
```

Celé riešenie potom vyzerá takto:

```
1 int main() {
2     int n, m;
3     cin >> n >> m;
4     Node *head = nullptr;
5     for (int i = 0; i < n; i++)
6         head = new Node(n - 1 - i, head);
7     while (m > 0) {
8         m--;
9         int x;
10        cin >> x;
11        Node *p = head, *prev = nullptr;
12        while (p->val != x) {
13            cout << p->val << " ";
14            prev = p;
15            p = p->nxt;
16        }
17        cout << "nasiel som " << x << " !" << endl;
18        if (prev != nullptr) {
19            prev->nxt = p->nxt;
20            p->nxt = head;
21            head = p;
22        }
23    }
24 }
```

**Úloha 97.** Pri riešení úlohy 70 (MergeSort) v kapitole 23 sme v operácii merge vždy prechádzali dve polia a výsledok kopírovali do tretieho. Napíš program na triedenie algoritmom MergeSort, ktorý používa spájané zoznamy (funkcia **sort** dostane ako parameter začiatok zoznamu) a nekopíruje dátu.

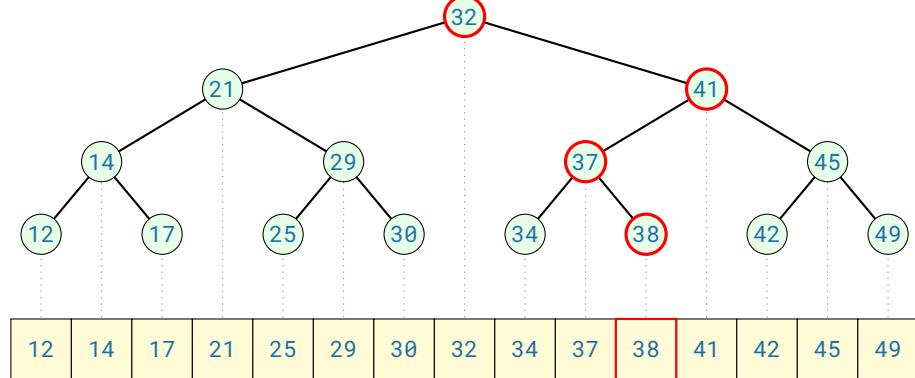
Spájaný zoznam<sup>3</sup>a pole sú v istom zmysle opakom jeden druhého: v poli sa dá jednoducho prejsť na hociktorý prvok, ale zle sa preusporiadava, lebo prvky treba kopírovať. V spájanom zozname sa zle prechádza na

<sup>3</sup>Spájaný zoznam sa dosť často hodí, takže v STL je naprogramovaný v knižnici <forward\_list>. Takže napr. `forward_list<int> a;` bude spájaný zoznam celých čísel, a poskytuje iterátor podobne ako `vector`, ale tento sa dá iba inkrementovať. Napr. `for(auto it = a.begin(); it != a.end(); it++) cout << (*it) << endl;` vypíše prvky zo zoznamu. Na pridávanie a mazanie prvkov v zozname slúžia funkcie `insert_after` a `erase_after`, ktoré ako prvý parameter dostanú iterátor, takže napr. `insert_after(it, 4)` vloží za prvok zoznamu, na ktorý ukazuje `it`, nový uzol s prvkom 4 a `erase_after(it)` zmaže nasledujúci uzol. Na pridanie a zmazanie prvého prvku slúžia funkcie `push_front(x)` a `pop_front()`. Pretože zoznam pracuje s pointermi na uzly, iterátori sa, na rozdiel od vektorov nikdy nezneplatnia (okrem prípadu, keď sa uzol vymaze).

Podobná trieda ako <forward\_list> je <list>. Rozdiel je v tom, že každý uzol si okrem pointeru `nxt` pamäta aj pointer `prev` na predchádzajúci prvok. To znamená, že iterátori sa dajú aj dekrementovať, ale celý zoznam zaberá viac pamäte. Namiesto `insert_after` a `erase_after` má podobné funkcie `insert(it, x)` a `erase(it)`, ktoré ale vkladajú prvok pred zadaný iterátor (takže vloží prvok na koniec zoznamu z sa dá pomocou `insert(z.end(), x)`).

## Vyhľadávacie stromy: `<set>` a `<map>` v STL

konkrétny prvok, lebo treba prejsť celý zoznam, ale dobre sa preusporiadava, lebo stačí poprehadzovať pointre. Skúsme spojiť dobré vlastnosti z oboch. Binárne vyhľadávanie v usporiadanom poli<sup>4</sup> sa dá predstaviť ako rozhodovací strom:



Ked' chceme nájsť číslo 38, najprv sa pozrieme do stredu, porovnám s číslom 32, kedže  $38 > 32$ , vydáme sa vpravo, porovnáme so 41 atď. Našim cieľom teraz bude vyrobiť si takýto rozhodovací strom v pamäti pomocou smerníkov. Ked' sa pozrieš na obrázok, vidiš, že strom sa skladá z krúžkov, ktoré budem volať *uzly* alebo *vrcholy*. V každom uzle je číslo, ktorému budem hovoriť *kľúč*. Najvyšší uzol v strede (na obrázku 32) bude *koreň*, uzly naspodu budú *listy*. Z každého uzla okrem listu vedú dve čiary (*hrany*), ktoré vedú do koreňov menších stromov (tým hovoríme *synovia* uzla). Najdlhšia vzdialenosť od koreňa do nejakého listu sa volá *hĺbka*. Základná vlastnosť, ktorá umožňuje binárne vyhľadávanie, je to, že v celom strome aj vo všetkých podstromoch platí: *ked si zoberiem hocjaký vrchol v, tak všetky kľúče v podstrome, ktorého koreň je ľavý syn v, sú menšie ako kľúč vo v a všetky kľúče v podstrome, ktorého koreň je pravý syn v, sú zase väčšie*. Takýto strom budeme volať *binárny vyhľadávací strom*. Spravme si takýto typ:

```

1 struct Node {
2     int key;
3     Node *left, *right;
4
5     Node(int _key = 0, Node *_l = nullptr, Node *_r = nullptr) {
6         key = _key;
7         left = _l;
8         right = _r;
9     }
10
11     ~Node() {
12         if (left != nullptr) delete left;
13         if (right != nullptr) delete right;
14     }
15 }
16 
```

Vyzerá veľmi podobne ako pri spájanom zozname, len má dva pointre (na ľavého a pravého syna). Pridal som navyše deštruktör: ak sa ide z pamäti odstrániť premenná typu `Node` (či už pri volaní `delete` alebo inak), najprv sa zavolá deštruktör, ktorý sa pozrie, že existuje ľavý a/alebo pravý syn a ak áno, rekurzívne ich zmaže (t.j. zavolá sa deštruktör ľavého syna, ktorý zavolá deštruktör jeho synov atď), až sa zmaže celý strom.

Ak máme utriedené pole prvkov, môžeme z neho vyrobiť strom napríklad takto:

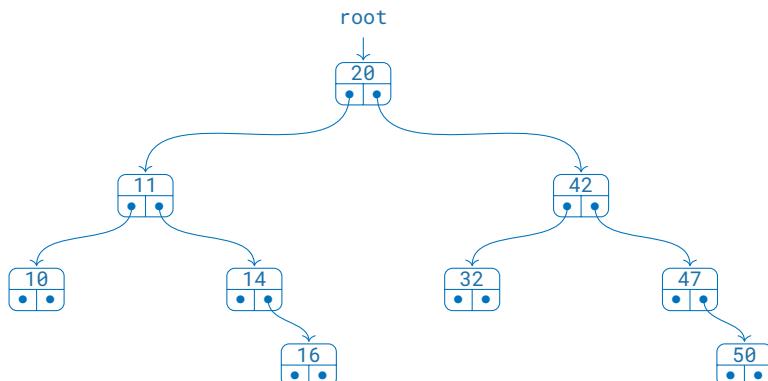
<sup>4</sup>nateraz nás bude zaujímať pole, v ktorom sa hodnoty neopakujú – ako v príklade so žiarovkami

```

1 Node *vyrob(vector<int> &a, int i, int j) {
2     Node *res = new Node();
3     if (i == j) {
4         res->key = a[i];
5     } else {
6         int m = (i + j) / 2;
7         res->key = a[m];
8         if (i != m) res->left = vyrob(a, i, m - 1);
9         if (m != j) res->right = vyrob(a, m + 1, j);
10    }
11    return res;
12}
13

```

Funkcia `vyrob` má ako parametre referenciu na vektor a dva indexy `i` a `j` a vytvorí strom, ktorý bude mať kľúče z úseku `a[i]...a[j]` (vrátane): zistí sa stredný prvok `a[m]` a ak je ľavý alebo pravý úsek neprázdný, rekúzívne sa zavolá `vyrob` a novovytvorené podstomky sa priradia do počítadiel `left` a `right`. Ak mám napr. pole `[10, 11, 14, 16, 20, 32, 42, 47, 50]` a zvolím `Node *root = vyrob(a, 0, a.size() - 1);`, vytvorí sa v pamäti takáto štruktúra zo počítadiel:



Ked' chceme zistiť, či sa nejaký kľúč v strome nachádza, stačí prechádzať stromom od koreňa (napr. `root->find(42);`). Metóda `find` vráti pointer na uzol s daným kľúčom, alebo `nullptr`, ak sa tam kľúč nenačadza.

```

1 struct Node {
2     int key;
3     Node *left, *right;
4     Node(int _key = 0, Node *_l = nullptr, Node *_r = nullptr) {...}
5     ~Node() {...}
6
7     Node *find(int x) {
8         if (x == key) return this;
9         if (x < key && left != nullptr) return left->find(x);
10        if (x > key && right != nullptr) return right->find(x);
11        return nullptr;
12    }
13}

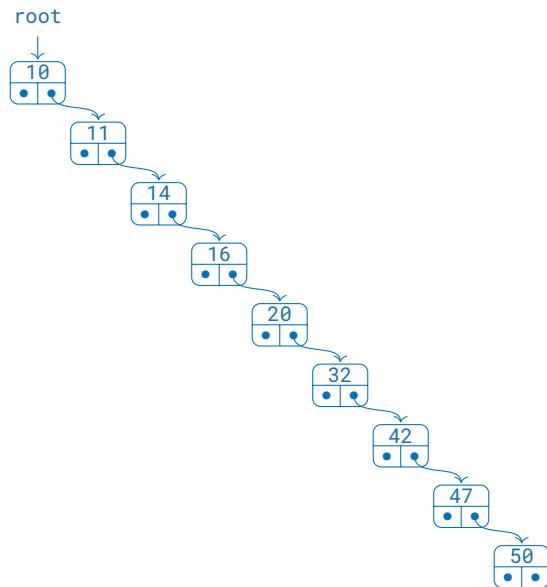
```

Teraz sme docieľili, že v pamäti máme namiesto poľa prvky pospájané smerníkmi a môžeme v nich vyhľadávať rovnako, ako pri binárnom vyhľadávaní v poli (teda v logaritmickej zložitosti). Oproti binárному vyhľadávaniu v poli sme si ale pomohli v tom, že aj pridanie prvku má logaritmickú zložitosť: jednoducho ideme po strome, ako keby sme vyhľadávali, a keď prideme na koniec, vytvoríme nový uzol. Metóda `insert` vráti `true`, ak sa vrchol v skutočnosti pridal a `false`, ak už v strome bol:

## Vyhľadávacie stromy: <set> a <map> v STL

```
1 struct Node {
2     int key;
3     Node *left, *right;
4     Node(int _key = 0, Node *_l = nullptr, Node *_r = nullptr) {...}
5     ~Node() {...}
6     Node *find(int x) {...}
7
8     bool insert(int x) {
9         if (x == key) return false;
10        if (x < key && left != nullptr) return left->insert(x);
11        if (x > key && right != nullptr) return right->insert(x);
12        Node *nd = new Node(x);
13        if (x < key) left = nd;
14        else right = nd;
15        return true;
16    }
17
18};
```

Zdá sa, že všetko funguje, a to je podozrivé. A naozaj. Keď začneš s jednoprvkovým poľom [10] a postupne voláš `insert` pre 11, 14, 16, 20, 32, 42, 47, 50, v pamäti to bude vyzerať takto:



Takže mám obyčajný spájaný zoznam a vyhľadávanie má lineárnu zložitosť. Problém je, samozrejme, v tom, že pridanie prvku môže spôsobiť, že strom prestane byť *vyväžený*, t.j. prestane platiť, že v každom uzle je hĺbka ľavého a pravého podstromu skoro rovnaká. Tomu sa dá zabrániť tak, že po každom vložení prvku sa zavolá *vyvažovacia operácia*, ktorá zabezpečí, že strom bude "rozumne"<sup>5</sup> vyväžený. Je veľa spôsobov, ako toto vyvažovanie urobiť (AVL stromy, Red-Black stromy, ...). Ukážem ti jeden spôsob, zvaný AA stromy.

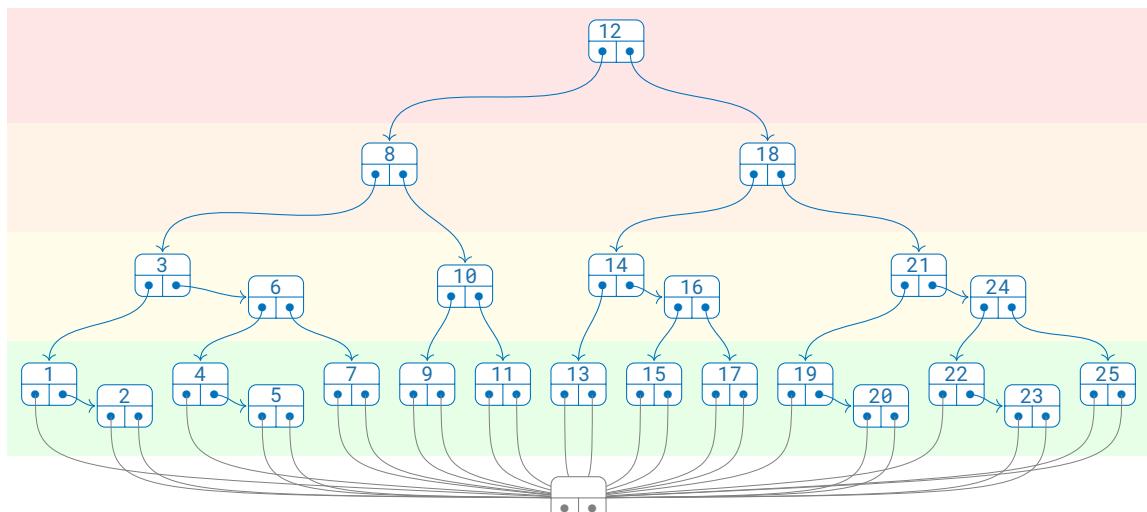
Celý prístup vyzerá takto: najprv si poviem pravidlá, ktoré musí AA strom spĺňať. Potom sa presvedčím, že každý strom, ktorý tieto pravidlá spĺňa, má logaritmickú hĺbku. Nakoniec naprogramujem operácie (vkladanie, vyhľadávanie, mazanie) tak, aby udržovali splnené pravidlá a navyše ich zložitosť bola úmerná hĺbke. Tak podme na to.

V AA-strome má každý uzol okrem kľúča aj "poschodie" (*level*). Pre zjednodušenie použijeme techniku zárážky a vyrábíme jeden špeciálny uzol "*prízemie*", ktorý bude na leveli 0. Pravidlá pre dobrý AA-strom sú takéto:

<sup>5</sup>t.j. tak, aby bolo zaručené, že operácie majú logaritmickú zložitosť

1. Je to binárny vyhľadávací strom: v každom uzle platí, že všetky uzly v ľavom podstrome majú menšie kľúče a všetky uzly v pravom podstrome väčšie.
2. Každý uzol (okrem prízemia) má dvoch synov.
3. Ľavý syn je na leveli o 1 menšom ako rodič.
4. Pravý syn je na rovnakom alebo o 1 menšom leveli ako rodič.
5. Ak je pravý syn na rovnakom leveli ako rodič, jeho pravý syn je na menšom leveli.

Pravidlo 5 hovorí, že nemôžu byť na rovnakom leveli tri uzly rodič – pravý syn – jeho pravý syn. Zároveň, keďže prízemie má level 0, tak podľa pravidla 3, listy musia byť na leveli 1 atď. AA-strom teda vyzerá nejak takto (farebné pásy sú levely):



Teraz chcem vyrátať, akú najväčšiu hĺbku môže mať AA strom, v ktorom je  $n$  uzlov. Spomeň si, že hĺbka je dĺžka najdlhšej cesty z koreňa do nejakého listu. Keď začнем z koreňa a postupujem smerom k listu, tak nikdy nejdem na vyšší level: bud idem na nižší (napr. z 18 do 14) alebo ostávam na tom istom (napr. zo 14 do 16). Ale takisto nemôžem ísť cez viac ako dva uzly na jednom leveli. Takže hĺbka je najviac dvojnásobok počtu levelov. Dobre. Podme teda vyrátať, koľko najviac levelov môže mať AA-strom, ktorý má  $n$  uzlov. Najprv sa opýtam tak trochu opačnú otázku: kolko najmenej listov môže mať AA-strom, ktorý má  $\ell$  levelov? Koreň je na najvyššom leveli (teda na leveli  $\ell$ ). Jeho ľavý syn je koreň AA-stromu s  $\ell - 1$  levelmi. Ak je pravý syn koreňa na nižšom leveli, tak je tiež koreňom nejakého (iného) AA-stromu s  $\ell - 1$  levelmi. Ak je pravý syn koreňa na rovnakom leveli (na leveli  $\ell$ ), tak jeho ľavý syn je koreň AA-stromu s  $\ell - 1$  levelmi. V každom prípade, v AA-strome s  $\ell$  levelmi viem nájsť dva rôzne AA-stromy s  $\ell - 1$  levelmi. Preto ak si označím  $L(\ell)$  najmenší počet listov, ktorý môže mať AA-strom s  $\ell$  levelmi, tak  $L(\ell) \geq 2L(\ell - 1)$ . Dostávam teda  $L(1) = 1$ ,  $L(2) \geq 2$ ,  $L(3) \geq 4$ ,  $L(4) \geq 8$ , ..., t.j.  $L(\ell) \geq 2^{\ell-1}$ . Teda AA-strom s  $\ell$  levelmi musí mať aspoň  $2^{\ell-1}$  uzlov. Vráťme sa teraz k pôvodnej otázke: kolko najviac levelov má AA-strom s  $n$  uzlami? Keby ich mal aspoň  $2 + \log n$ , tak by musel mať  $L(2 + \log n) \geq 2^{2+\log n-1} = 2n$  listov. To ale nemôže, lebo má len  $n$  uzlov. Takže AA-strom s  $n$  uzlami musí mať menej ako  $2 + \log n$  levelov, a preto má aj logaritmickú hĺbku.

Ostáva už len posledné: naprogramovať vkladanie, vyberanie a vyhľadávanie do AA-stromu tak, aby ich zložitosť zodpovedala hĺbke stromu. Keďže AA-strom je binárny vyhľadávací strom, vyhľadávanie je ľahké. Najprv si správim typ Node podobne ako predtým. Rozdiel bude v tom, že uzol si navýše bude pamätať svoj level. Konštruktor aj metóda `find` okrem kľúča dostanú aj pointer na prízemie (`base`). Namiesto deštruktora budem mať procedúru `destroy`, lebo deštruktory nemôžu mať parametre a ja potrebujem vedieť, čo je zarážka.

```

1 struct Node {
2     int level, key;
3     Node *left, *right;

```

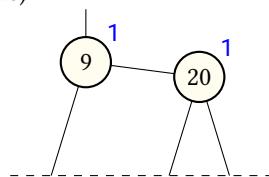
## Vyhľadávacie stromy: <set> a <map> v STL

```
4     Node(int k, Node *base) { level = 1; key = k; left = right = base; }
5
6     void destroy(Node *base) {
7         if (this == base) return;
8         left->destroy(base);
9         right->destroy(base);
10        delete this;
11    }
12
13    bool find(int k, Node *base) {
14        if (this == base) return false;
15        if (key == k) return true;
16        if (k < key)
17            return left->find(k, base);
18        else
19            return right->find(k, base);
20    }
21 }
22 };
```

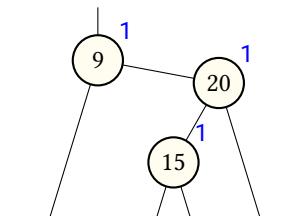
Urobím si aj nový typ `Tree` pre celý strom, v ktorom budem mať zapamätaný koreň a prízemie:

```
1 struct Tree {
2     Node *root, *base;
3
4     Tree() {
5         base = new Node(0,nullptr); // vytvoríme si prízemie
6         base->level = 0;
7         base->left = base->right = base;
8         root = base;
9     }
10
11 ~Tree() { root->destroy(base); delete base; }
12
13     bool find(int key) { return root->find(key, base); }
14 }
```

Pridávanie ale začne byť trochu komplikovanejšie. Zoberme si jednoduchú funkciu `insert` z predchádzajúceho prípadu. Začнем s prázdnym stromom a pridám doňho čísla 9 a 20. Bude to vyzerať nejak takto (modré číslo znamená level, čiarkovaná čiara je prízemie):

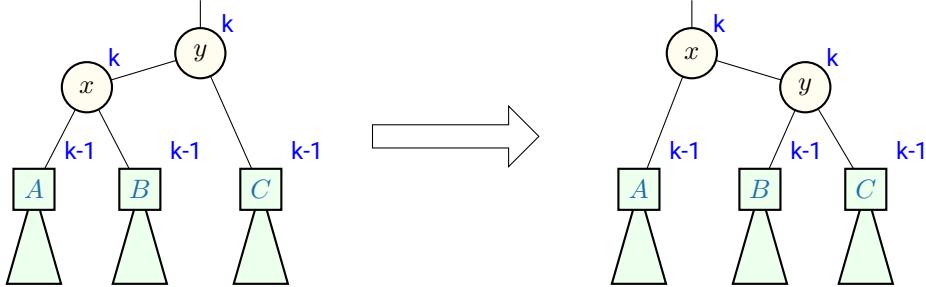


Ked chcem pridať číslo 15, vznikne problém:



Uzol 15 musí byť na leveli 1, lebo je to list, ale zároveň je ľavým synom uzla 20, preto 20 musí mať level 2. Lenže pravý syn 20ky je prízemie (s levelom 0), preto 20 musí mať level 1. Takže je zrejmé, že nestačí len aktualizovať

leveley, ale bude treba prerobiť aj celý strom. V tomto príklade môžem na uzol 20 použiť operáciu *skew* (*rotácia*), ktorá vyzerá takto:

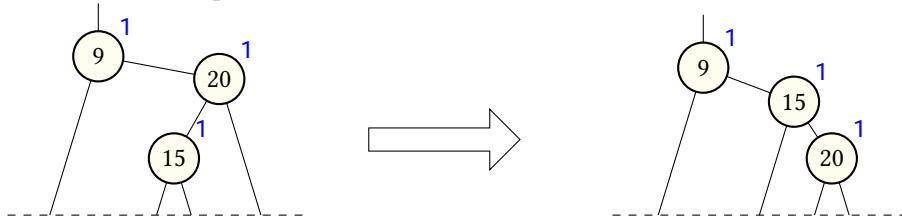


Povedané slovami: ak mám podstrom s koreňom  $y$  na leveli  $k$ , ktorý má ľavého syna  $x$  na leveli  $k$  (čím sa porušilo pravidlo 3) a  $x$  má obidvoch synov na leveli  $k-1$ , môžem za koreň podstromu zvoliť  $x$  a pravého syna  $x$  prevesiť pod  $y$ . Po operácii posun strom ostane dobrý vyhľadávací strom, lebo všetky kľúče v podstrome  $B$  sú menšie ako  $y$ . Operáciu *skew* si naprogramujem ako metódu `Node`. Zavolám ju v koreni podstromu (napr. `root=root->skew()`) a bude vracať pointer na nový koreň stromu:

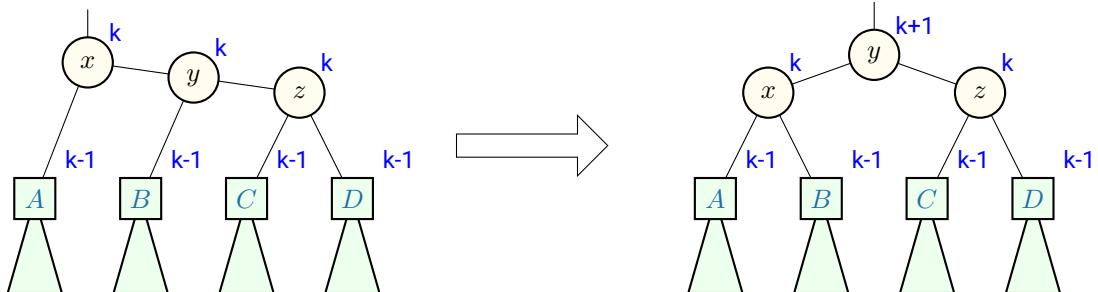
```

1 Node *skew() {
2     if (left->level != level) return this;
3     Node *res = left, *tmp = left->right;
4     left->right = this;
5     left = tmp;
6     return res;
7 }
```

Ak aplikujem *skew* na uzol 20 z príkladu, dostanem



Teraz som vyriešil problém v pravom podstrome uzla 9, ale problém je v samotnej 9ke: uzly 15 aj 20 sú na leveli 1, a tým porušujú pravidlo 5. To ošetrím druhou operáciou, ktorú nazvem *split* (*rozdelenie*) a vyzerá takto:



Slovami, ak mám postupne dvoch pravých synov na rovnakom leveli ako rodič, môžem stredného posunúť na vyšší level. Opäť to naprogramujem rovnako, ako predtým, t.j. ako metódu, ktorá vracia pointer na nový koreň:

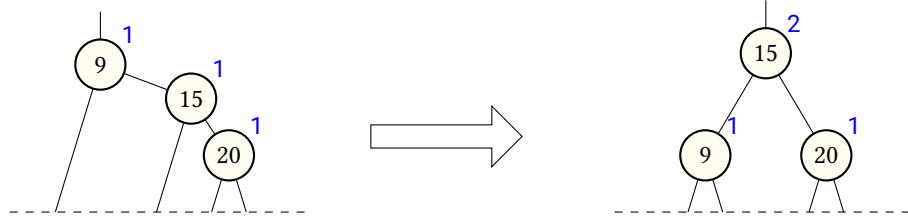
```

1 Node *split() {
2     if (right->right->level != level) return this;
3     right->level++;
4     Node *res = right;
5     right = right->left;
6     res->left = this;
```

## Vyhľadávacie stromy: `<set>` a `<map>` v STL

```
7     return res;  
8 }
```

Ked' aplikujem `split` na vrchol 15, výsledný strom bude



Pridaním listu a následným volaním `skew` a `split` sa mi mohlo stať, že sa porušia pravidlá vyššie v strome. V mojom príklade je teraz koreň namiesto uzla 9 s levelom 1, uzol 15 s levelom 2. Keby to neboli celý strom, ale nad ním by bol napr. uzol 30 s levelom 2, porušilo by sa mi v ňom pravidlo 3. Možnosti, ako sa to môže stať, sú ale rovnaké, ako pri pridávaní listu<sup>6</sup>. Preto celé pridávanie môžem napísť jednoducho takto:

```
1 struct Node {  
2     int level, key;  
3     Node *left, *right;  
4  
5     Node(int k, Node *base) { ... }  
6     void destroy(Node *base) { ... }  
7     bool find(int k, Node *base) { ... }  
8  
9     Node *skew() { ... }  
10    Node *split() { ... }  
11  
12    Node *insert(int k, Node *base) {  
13        if (this == base) return new Node(k, base);  
14        if (k == key) return this;  
15        if (k < key)  
16            left = left->insert(k, base);  
17        else  
18            right = right->insert(k, base);  
19        return skew()->split();  
20    }  
21};  
22  
23 struct Tree {  
24     Node *root;  
25     Node *base;  
26  
27     Tree() { ... }  
28     ~Tree() { ... }  
29  
30     void insert(int key) { root = root->insert(key, base); }  
31     bool find(int key) { return root->find(key, base); }  
32 };
```

Uzly, ktoré sú v strome, sa dajú odstrániť podobným spôsobom. Ak mám odstrániť list, nájdem ho ako pri vyhľadávaní, zmažem ho, a po zmazení dovyvažujem strom. Problém je, čo urobiť, ak mám vymazať uzol, ktorý nie je list. Pomôže drobná finta: ak mám vymazať uzol  $x$ , ktorý má ľavého syna, nájdem si uzol  $y$  s najväčším kľúčom v podstrome ľavého syna (t.j. najpravejší list). Viem, že  $y$  je väčší, ako všetky kľúče v ľavom podstrome  $x$  a zároveň menší, ako všetky kľúče v pravom podstrome. Môžem teda vymeniť hodnoty  $x$  a  $y$ , a

<sup>6</sup>všimni si, že jediné miesto, kde level môže stúpnutť, je pri operácii `split`, ale potom sú obaja synovia na menšom leveli

už mi stačí vymazať list. Ak  $x$  nemal ľavého syna, ale iba pravého, urobím to symetricky (hľadám najľavejší list). V súbore `aatree.h`<sup>7</sup> je strom aj s dopísanou operáciou `erase`: pri postupe smerom nadol si v premennej `deleted` uchovávam posledný uzol, ktorého kľúč je  $\geq k$ . Keď prídem do príslušného listu (najľavejšieho alebo najpravejšieho), tak vymením hodnoty a dovyvažujem strom (skús si premyslieť, ako to funguje).

Vráťme sa teraz k úlohe o žiarovkách s výrobnými číslami. S použitím AA stromu je to ľahké.

**Úloha 98.** Na vstupe je postupnosť príkazov. Každý príkaz je tvaru `! x alebo ? x a posledný príkaz je #`. Predpokladajme, že máme žiarovky, ktoré sú na začiatku vypnuté. Príkaz `! x` znamená, že sme prepíšli žiarovku s výrobným číslom  $x$  (ak bola vypnutá, bude zapnutá a naopak). Pre každý príkaz `? x` treba vypísať, `svieti` alebo `nesvieti` podľa toho, v akom stave je práve žiarovka číslo  $x$ .

Skús si napísať rekurzívnu funkciu na nasledujúcu úlohu:

**Úloha 99.** Napíš funkciu, ktorá vypíše všetky kľúče z AA stromu v utriedenom poradí.

V STL sú podobné stromy naprogramované v type, ktorý sa volá *monžina* (`set`). Na použitie treba pridať `#include <set>`. Je to šablóna, ktorá má ako parameter typ kľúča, takže napr. `set<int> s`; je strom, ktorý má kľúče typu `int`. Namiesto `int` to môže byť hocjaký iný typ, ktorý má definované porovnanie. Pri vlastných typoch treba `operator<` definovať ako `const`, napr. takto:

```

1 struct Osoba {
2     string meno, priezvisko;
3
4     bool operator<(const Osoba& x) const {
5         if (priezvisko == x.priezvisko) return meno < x.meno;
6         return priezvisko < x.priezvisko;
7     }
8 }
```

`const` pri parametri `x` znamená, že `x` je referencia na premennú typu `Osoba`, ale funkcia `operator<` slúbuje, že `x` nebude meniť. Podobne `const` za menom funkcie slúbuje, že sa nebude meniť `*this`, t.j. ak zavolám `x.operator<(y)`, tak sa nezmení ani `x` ani `y`. Pri takejto definícii môžem použiť `set<Osoba>` a budem mať v strome osoby utriedené podľa priezviska a pri rovnakom priezvisku podľa mena.

Iná možnosť je použiť šablónu `set` s dvoma parametrami, kde druhý parameter je typ porovnávacej funkcie, a porovnávacia funkcia je potom parametrom konštruktora. Keby typ `Osoba` nemal definované porovnanie, alebo keby som chcel mať strom usporiadaný najprv podľa mena a potom podľa priezviska, môžem napísať (potrebujem pridať `#include <functional>`):

```

1 set<Osoba, function<bool(const Osoba &, const Osoba &)>> s(
2     [](const Osoba &x, const Osoba &y) {
3         if (x.meno == y.meno) return x.priezvisko < y.priezvisko;
4         return x.meno < y.meno;
5     });

```

Tým hovoríme, že chceme vyrobiť vyhľadávací strom s uzlami s hodnotou `Osoba` a na porovnávanie budem používať funkciu (lambdu), ktorá zoberie referencie na dve premenné typu `Osoba` a vráti `bool`.

Podobne ako v našom type `Tree`, typ `set` má operácie `s.insert(x)` a `s.erase(x)`. Keďže je to kontajner, má definované aj iterátory<sup>8</sup>, takže napr.

```

1 for (auto it = s.begin(); it != s.end(); it++) cout << *it << " ";

```

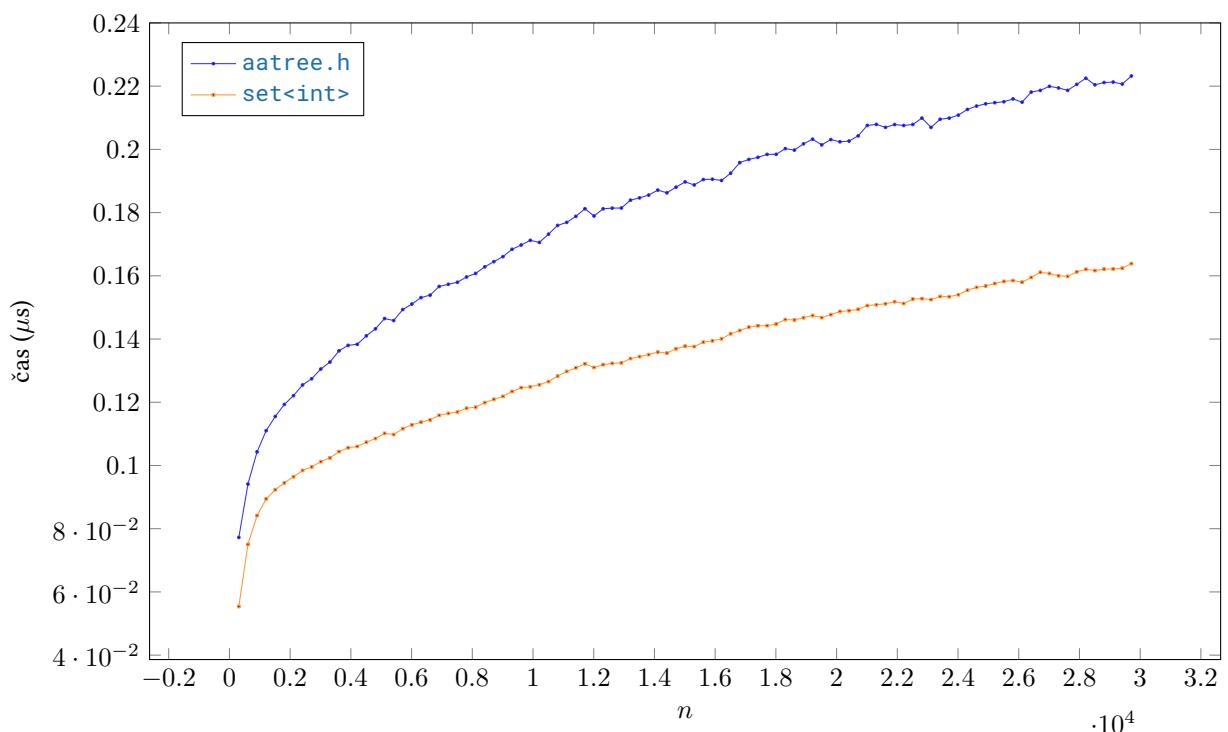
<sup>7</sup><https://github.com/pocestny/programovanie/raw/master/materialy/aatree.h>

<sup>8</sup>Na rozdiel od vektora, iterátor pre `set` sa dá posunúť len na susedný prvok, t.j. môžem mať `it++` alebo `it--`, ale nie `it+10`

## Vyhľadávacie stromy: `<set>` a `<map>` v STL

vypíše všetky prvky. Navyše je zaručené, že pri prechode iterátorm budú prvky usporiadane od najmenšieho po najväčší. Vyhľadávanie prvku robí metóda `find(x)`, ktorá vráti iterátor na prvak s klúčom `x`, alebo `end()`, ak tam prvak s daným klúčom nie je. Takže zistíť, či je klúč `x` v množine, sa dá pomocou `if (s.find(x) != s.end())`. Metóda `erase` má dve verzie: buď vymaze prvak s daným klúčom alebo prvak s daným iterátorom. Napr. `s.erase(s.begin())` vymaze najmenší prvak. Hodí sa ešte funkcia `upper_bound(x)` (resp. `lower_bound(x)`), ktorá vráti iterátor na najmenší prvak väčší (resp. väčší alebo rovný) ako `x`.

Ako presne je `set` naprogramovaný je na konkrétnom komplilátore, ale musí zaručiť, aby zložitosť všetkých operácií (vyhľadávanie, vkladanie, mazanie) bola logaritmická. Existujúce komplilátory väčšinou používajú tzv. *Red-Black* stromy, ktoré majú trochu iný spôsob vyvažovania, ale v podstate sú veľmi podobné AA stromom. V nasledovnom grafe je porovnanie časov. Pre rôzne hodnoty  $n$  som povkladal do stromu  $n$  rôznych čísel (v náhodnom poradí) a potom som ich (v náhodnom poradí) zo stromu vymazal. Meral som si priemerný čas, ktorý trvala jedna operácia. Vidno, že `set` je trochu rýchlejší<sup>9</sup>, ako nás `Tree`, ale zase nie až tak o veľa. V obidvoch prípadoch rastie čas, ktorý treba na vloženie (vymazanie) iba veľmi pomaly v závislosti od veľkosti stromu.



V STL je aj niekoľko podobných dátových štruktúr. `multiset` sa od `set` líši tým, že dovoľuje mať viacero prvkov s rovnakým klúčom<sup>10</sup>.

Podobný ako `set` je aj typ `map`, v ktorom každý uzol má okrem klúča aj hodnotu. Je to šablóna s dvoma parametrami, napr. `map<string, int> m`; používa ako klúč `string`<sup>11</sup> a každý uzol má hodnotu typu `int`. K uloženým hodnotám sa dá príjemne pristupovať pomocou operátora `[ ]`, podobne ako pri vektore, takže napr. `m["zaba"] = 5`; vloží do stromu uzol s klúčom `"zaba"` a hodnotou 5 (ak tam už uzol s takým klúčom bol, zmení jeho hodnotu). Podobne `x = m["zaba"]`; uloží do premennej `x` hodnotu, ktorá je v uzle s klúčom `"zaba"` (ak taký uzol nie je, tak sa vytvorí, a jeho hodnota bude 0; v tom prípade sa teda do `x` uloží 0). Metódy `find` a `erase` fungujú rovnako ako pri type `set`: `find` vráti iterátor na uzol s daným klúčom a `erase` vymaze buď uzol s daným klúčom, alebo iterátor.

<sup>9</sup>a aj konzistentnejší, modrá krivka je viac roztriasená

<sup>10</sup>Tu si treba dať pozor: `s.erase(x)` vymaze všetky prvky s klúčom `x`. Na vymazanie jedného treba použiť `s.erase(s.find(x))`, t.j. nájsť iterátor na nejaký konkrétny uzol s klúčom `x` a potom vymazať ten.

<sup>11</sup>`string` má definovaný operátor porovnania, ktorý dva reťazce porovná podľa abecedy ako slová v slovníku

Iterátory v type `map` sú trochu zložitejšie. Kým iterátor pre `set` sa dá chápať ako pointer na klúč (napr. ak mám `auto it=s.begin();`, tak `*s` je klúč), iterátor na `map` potrebuje ukazovať na dve veci: klúč a hodnotu. Dať dve veci do jednej premennej je treba často a v STL je na to špeciálny typ `pair`, ktorý vyzerá takto:

```
1 template<class T1, class T2>
2 struct pair {
3     T1 first;
4     T2 second;
5 };
```

Sú v ňom definované rôzne pomocné funkcie (napr. `operator==` a pod.). Taktiež aj funkcia (šablóna) `make_pair`, ktorá ušetrí písanie `<T1, T2>`. Kompilátor si totiž vie odvodíť typy, takže ak napišeš napr. `make_pair(3, 4.7)`, tak vráti `pair<int, double>`.

```
1 template<class T1, class T2>
2 pair<T1, T2> make_pair(T1 x, T2 y) {
3     return pair<T1, T2>(x, y);
4 }
```

Iterátor na `map` sa správa ako pointer na `pair`, kde prvá zložka je klúč a druhá hodnota. Napr. keby som chcel mať vyhľadávací strom, v ktorom podľa zvuku viem nájsť živočícha, použil by som `map<string, string>` napr. takto:

```
1 #include <iostream>
2 #include <map>
3 #include <string>
4 using namespace std;
5
6 int main() {
7     map<string, string> m;
8     m["vauu"] = "vlk";
9     m["mnau"] = "macka";
10    m["kvak"] = "zaba";
11    m["singularna\u010dkohomologia"] = "matematik";
12
13    for (auto it = m.begin(); it != m.end(); it++)
14        cout << it->second << "\u010duhovori\u010du" << it->first << " " << endl;
15 }
```

Všimni si, že program bude vypisovať živočíchy utriedené abecedne podľa zvuku, ktorý vydávajú. No a posledná vec, iterátory na `set` aj `map` sú, na rozdiel od vektora `const`. Preto ak mám napr. `vector<int> a;`, tak môžem napísaať

```
1 auto it = a.begin();
2 it++;
3 *it=10;
```

a bude to to isté, ako `a[1]=10`, ak by som mal `set<int> a;`, tak `*it=10`; napísaať nemôžem<sup>12</sup>.

Predtým, ako sa pustíme do úloh, ešte jedna poznámka o zložitosti. Ukázali sme si, že `set` a `map` sú naprogramované tak, že zložitosť všetkých operácií je logaritmická. To je pravda, ale ak to povieme presnejšie, tak logaritmický je počet volaní porovnania. Ak by si mal napr. `set<string>`, tak pri vyhľadávaní sa v každom kroku musí porovnať hľadaný reťazec s klúčom príslušného uzla. A to môže trvať dosť dlho, ak máš dlhé reťazce. Takže počet porovnaní sice je logaritmický, ale jedno porovnanie môže byť veľmi drahé. Na to si treba dávať pozor, ak používaš ako klúč zložitejšie typy.

<sup>12</sup>To, samozrejme, dáva zmysel: nemôžem len tak zmeniť hodnotu v uzle vyhľadávacieho stromu; tým sa mi veci môžu pokaziť.

## Vyhľadávacie stromy: <set> a <map> v STL

**Úloha 100.** Na vstupe je číslo  $n$  a potom  $n$  riadkov, z ktorých každý opisuje výsledok hry. Riadok v tvare **Jano Fero 10** znamená, že hráč **Jano** vyhral od hráča **Fero 10** tokenov (t.j. **Jano** má o 10 viac a **Fero** o 10 menej). Napíš program, ktorý vypíše všetkých hráčov v abecednom poradí a pre každého jeho záverečný zisk (t.j. kladné číslo, ak má viac tokenov ako na začiatku a záporné ak má menej). Napr.

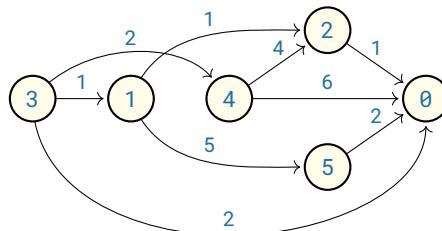
```
5
Jano Fero 10
Fero Mara 42
Mara Jano 14
Dream Technoblade 20
Technoblade Fero 84
```

```
Dream: 20
Fero: -52
Jano: -4
Mara: -28
Technoblade: 64
```

Podme teraz spolu vyriešiť takúto úlohu:

**Úloha 101.** Máme za úlohu vykonať  $n$  činností očíslovaných  $0, 1, \dots, n-1$ . Zároveň máme daných  $m$  podmienok tvaru  $i \xrightarrow{t} j$ , ktoré hovoria, že činnosť  $j$  sa dá urobiť najskôr  $t$  dní po tom, ako sa urobí činnosť  $i$ . V jeden deň sa dá urobiť hocikolko veľa činností, ak to podmienky dovoľujú. Napíš program, ktorý vypíše poradie, v akom sa dajú činnosti vykonať, alebo napíše, že sa to nedá. Napr.

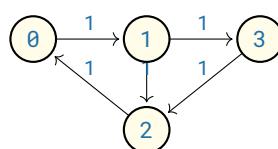
```
6 9
3 0 2
1 2 1
4 0 6
3 1 1
4 2 4
2 0 1
3 4 2
1 5 5
5 0 2
```



```
1. den: 3
2. den: 1
3. den: 4
7. den: 5 2
9. den: 0
```

alebo

```
4 5
0 1 1
1 2 1
1 3 1
3 2 1
2 0 1
```



neda sa

```
1 struct Sipka {
2     int j, t;
3 };
```

Všetky podmienky budem mať uložené v premennej `vector<vector<Sipka>> G`. To znamená, že `G` je vektor, ktorého prvky sú zase vektory, v ktorých sú uložené šípky. `G[i]` bude vektor, v ktorom budem mať zapamätané šípky, ktoré odchádzajú z  $i^{13}$ . Na začiatku si inicializujem `G` na veľkosť  $n$ , t.j. bdu obsahovať  $n$  prázdnych vektorov. Potom vždy prečítam jednu podmienku  $i \xrightarrow{t} j$  a pridám `Sipka{j, t}` na koniec vektora `G[i]` takto:

<sup>13</sup>preto som si v type `Sipka` zapamätať iba cieľovú činnosť. Odkiaľ šípka vychádza bude dané tým, kde je uložená.

```

1 cin >> n >> m;
2 vector<vector<Sipka>> G(n);
3 while (m-- > 0) {
4     int i, j, t;
5     cin >> i >> j >> t;
6     G[i].push_back(Sipka{j, t});
7 }
```

Keď mám prečítaný vstup, idem úlohu riešiť. Priamočiary prístup by bol prechádzať postupne dni, vždy skontrolovať, ktoré činnosti môžem urobiť, a tie urobiť<sup>14</sup>. Na začiatku môžem urobiť všetky činnosti, do ktorých nevchádza žiadna šípka. Keď ich urobím, odstránim z nich odchádzajúce šípky, a tým sa mi môžu objaviť nové činnosti, do ktorých žiadna šípka nevchádza. Tie ale môžem urobiť až potom, ako vyčkám patričný čas. Preto si o každej činnosti budem pamätať dve veci: *stupeň*, t.j. koľko šípok do nej ešte vchádza, a čas, dokedy treba čakať, aby sa splnili už vybavené šípky. Budem mať typ

```

1 struct Info {
2     int deg, t;
3 }
```

a `vector<Info> a`, ktorý si pri čítaní vstupu nastavím takto:

```

1 cin >> n >> m;
2
3 vector<Info> a(n);
4 for (int i = 0; i < n; i++) {
5     a[i].t = 0;
6     a[i].deg = 0;
7 }
8
9 vector<vector<Sipka>> G(n);
10 while (m-- > 0) {
11     int i, j, t;
12     cin >> i >> j >> t;
13     G[i].push_back(Sipka{j, t});
14     a[j].deg++;
15 }
```

Spracovať udalosť `i` potom znamená pre každú udalosť `j`, do ktorej viedie z `i` šípka, znížiť stupeň a upraviť čas, kedy ju môžem najskôr urobiť:

```

1 for (Sipka s : G[i]) {
2     a[s.j].deg--;
3     a[s.j].t = max(a[s.j].t, a[i].t + s.t);
4 }
```

Tu som použil nový zápis `for (Sipka s : G[i])` prikaz, ktorý je skratkou za

```

1 for (auto it = G[i].begin(); it != G[i].end(); it++) {
2     Sipka s = *it;
3     <prikaz>
4 }
```

<sup>14</sup>Je totiž jasné, že ak nejakú činnosť môžem urobiť, neoplatí sa ju odkladať. Keby som mal napríklad ale obmedzený počet činností, ktoré za deň môžem urobiť, všetko by bolo úplne inak.

## Vyhľadávacie stromy: <set> a <map> v STL

Všeobecne môžem písť `for( auto w : x)`, kde `x` môže byť hocičo, čo má príslušné iterátory (`vector`, `set`, `map`, vlastný typ, ...). Namiesto vypísania typu `w` môžem použiť `auto`<sup>15</sup>.

Pokračujme ale v riešení našej úlohy. Kedže časy čakania môžu byť veľmi veľké, nie je dobrý nápad robiť cyklus, v ktorom by sme išli deň po dni. Namiesto toho chceme spôsob, ako nájsť činnosti, do ktorých nevchádza žiadna šípka a spomedzi nich navyše takú, ktorá má najmenší čas. Vyriešim to tak, že si pre typ `Info` definujem operátor porovnania:

```
1 struct Info {
2     int deg, t;
3     bool operator<(const Info& x) const {
4         if (deg != x.deg) return deg < x.deg;
5         return t < x.t;
6     }
7 }
```

Teraz nemusím prechádzať deň po dni a kontrolovať, ktoré činnosti viem urobiť, ale vždy si vyberiem činnosť `i` s najmenšou hodnotou `a[i]`. Keby som mal namiesto vektora hodnoty `Info` uložené vo vyhľadávacom strome, najmenšiu by som mal vždy poruke (bola by v koreni). V strome mám ale problém pri spracovaní udalosti, lebo neviem, kde v strome sú koncové udalosti šípok uložené. To sa dá vyriešiť rôzne, ja to teraz spravím tak, že v strome nebudem mať uložené celé `Info`, ale iba indexy do poľa `a`. Na porovnávanie kľúčov v strome budem používať vlastnú funkciu, takže si vytvorím množinu `s` a uložím do nej indexy všetkých prvkov z poľa `a`.

```
1 multiset<int, function<bool(int, int)>> s([&](int i, int j) { return a[i] < a[j]; });
2
3 for (int i = 0; i < n; i++) s.insert(i);
```

Parametre šablóny pre moju multimnožinu hovoria, že v nej budem ukladať typ `int` a ako funkciu na porovnávanie použijem `function<bool(int, int)>`, ktorá bude parametrom konštruktora. Do konštruktora posielam lambdu, čo využíva porovnávanie, ktoré som dal do typu `Info`. Moja lambda porovnáva prvky v (globálном, preto potrebujem `[&]`) poli `a`. Všimni si, že aj keď plánujem mať v strome uložené rôzne prvky, použil som `multiset`. To preto, že moja porovnávacia funkcia nie je úplná, všetky činnosti s rovnakým stupňom a časom sú rovnaké. To vlastne znamená, že v strome môžem mať viac prvkov s rovnakým kľúčom.

Ostáva už len premyslieť si, ako budem ukladať výstup. Urobím si dve premenné

```
1 vector<int> ts(1, 0);
2 vector<vector<int>> out(1);
```

V `ts` si budem ukladať dni, v ktorých nejaké činnosti robím a `out[i]` bude vektor všetkých činností, ktoré urobím v deň `ts[i]`. Hlavná časť programu bude vyzerať takto:

<sup>15</sup>Poznámka pre fajnšmekrov: Keby som mal napr. `vector<vector<int>> G`; a napíšem `for (auto w : G) w[0]=7;`, tak `w` bude typu `vector<int>` a pri vykonávaní neviditeľného `vector<int> w = *it` vovnútri cyklu sa vo `w` vytvorí lokálna kópia vektora `*it`. Výsledkom bude, že dátá sa budú veľakrát kopírovať a hodnoty v `G` sa nezmenia. Ak napíšem `for (auto &w : G) x[0]=7;`, tak `w` bude typu `vector<int>&`, t.j. referencia (pointer) na vektor a preto sa hodnoty z `G` nebudú kopírovať, ale sa budú meniť priamo.

```

1 while (s.size() > 0) {
2     int v = *s.begin();
3
4     if (a[v].deg > 0) {
5         cout << "nedasa" << endl;
6         exit(0); // volanie exit() ukončí celý program
7     }
8     if (a[v].t != ts.back()) { // ak treba, začнем nový deň
9         out.push_back(vector<int>{});
10    ts.push_back(a[v].t);
11 }
12 out.back().push_back(v); // uložím činnosť do aktuálneho dňa
13 s.erase(s.begin());
14 for (Sipka r : G[v]) {
15     s.erase(r.j);
16     a[r.j].deg--;
17     a[r.j].t = max(a[r.j].t, a[v].t + r.t);
18     s.insert(r.j);
19 }
20 }
```

Všimni si, že na riadkoch 15 a 18 prvok `r.j` najprv zo stromu vymažem a potom ho tam vložím. To je preto, že keď zmením `a[r.j]`, porovnania medzi prvkami v strome sa môžu zmeniť a strom by nebol konzistentný. Toto si treba uvedomiť vždy, ak porovnávacia funkcia pre strom používa globálne premenné.

Nakoniec iba vypíšem pole `out` a hotovo:

```

1 for (int i = 0; i < ts.size(); i++) {
2     cout << 1 + ts[i] << ".den:";
3     for (int x : out[i]) cout << " " << x;
4     cout << endl;
5 }
```

Tu je zopár príkladov, kde sa dajú výhodne použiť typy `set` a `map`.

**Úloha 102.** Na vstupe je  $n$  celých kladných čísel. V jednom kroku si môžem zvoliť hocjaké párné číslo  $c$  a všetky výskytu  $c$  vydeliť dvoma. Napr. ak mám pole  $3 \ 6 \ 4 \ 12 \ 5 \ 12 \ 6$  a zvolím si  $12$ , dostanem  $6 \ 6 \ 4 \ 6 \ 5 \ 6 \ 6$ . Napiš program, ktorý pre zadané pole zistí, kolko najmenej krokov treba na to, aby všetky čísla v ňom boli nepárne. Napr. pre vstup  $3 \ 6 \ 4 \ 12 \ 5 \ 12 \ 6$  je odpoveď  $4$ : postupne zvolím  $12, 6, 4, 2$ .

**Úloha 103.** Na vstupe je pole  $n$  čísel. V jednom kroku môžem vybrať hocjaké dve čísla, odstrániť ich z poľa a pridať do poľa ich súčet. Cena kroku je pridaný súčet. Napr. ak mám pole  $6 \ 4 \ 1 \ 3$  a vyberiem si  $6$  a  $3$ , tak budem mať pole  $4 \ 1 \ 9$  a zaplatím  $9$ . Napiš program, ktorý zistí, akú najmenšiu cenu treba na to, aby v poli ostalo jediné číslo. Napr. pre vstup  $6 \ 4 \ 1 \ 3$  je výsledok  $26$ : najprv vyberiem  $1$  a  $3$ , pričom zaplatím  $4$  a budem mať pole  $6 \ 4 \ 4$ . Potom vyberiem  $4$  a  $4$ , zaplatím  $8$  a dostanem pole  $6 \ 8$  a nakoniec vyberiem  $6$  a  $8$  a zaplatím  $14$ .

**Úloha 104.** V systéme sú rôzne produkty, každý má meno skladajúce sa z písmen a číselnú verziu. Verzie sa postupne zvyšujú od 1. Produkt a jeho verzia sa oddelujú pomlčkou, takže nap. `hrniec-3` je verzia 3 produktu `hrniec`. Na vstupe sú tri typy príkazov: `? produkt`, `* produkt-verzia`, `+ produkt`. Vstup je ukončený príkazom `!`. V prvom prípade treba odpovedať číslom aktuálnej verzie (alebo 0, ak produkt neexistuje), v druhom prípade je odpoveď `ano/nie` podľa toho, či v systéme existuje produkt s danou verzou (možno aj staršou) a v treťom prípade pridať novú verziu s číslom o 1 väčším ako doteraz aktuálna verzia.

```
+ hrniec
+ hrniec
+ ponozka
* hrniec-1
* ponozka-4
? ponozka
? hrniec
!
```

```
ano
nie
1
2
```

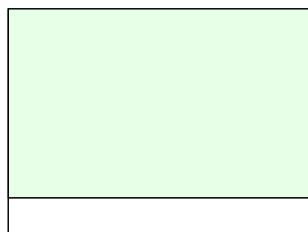
**Úloha 105.** Na vstupe je niekolko riadkov, na každom riadku je veta tvaru [<Podmet>](#) je [<privlastok>](#). alebo [<Podmet>](#) [nie je <privlastok>](#). Posledný riadok vstupu obsahuje iba znak '!'. Napíš program, ktorý prečíta vstup a pre každý podmet vypíše všetky prívlastky.

```
Slon je velky.
Stolicka je tvrda.
Zaba nie je tvrda.
Slon je makky.
Zaba je zelena.
Stolicka nie je zelena.
Slon je hladny.
Zaba je makka.
!
```

```
Slon je hladny, makky a velky.
Stolicka je tvrda.
Zaba je makka a zelena.
```

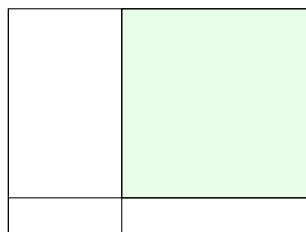
**Úloha 106.** Na prvom riadku vstupu sú tri čísla:  $w, h, n$ . Začíname s obdĺžnikom rozmerov  $w \times h$ . Nasleduje  $n$  riadkov, ktoré popisujú čiary cez obdĺžnik. Riadok [H y](#) znamená, že nakreslíme horizontálnu (vodorovnú) čiaru cez celý obdĺžnik vo výške [y](#). Riadok [V x](#) znamená, že nakreslíme vertikálnu (zvislú) čiaru vo vzdialosti [x](#). Napíš program, ktorý po každom vstupe vypíše plochu najväčšieho dielika. Napríklad pre  $w = 8, h = 6$  môže byť takáto postupnosť:

[H 1](#)



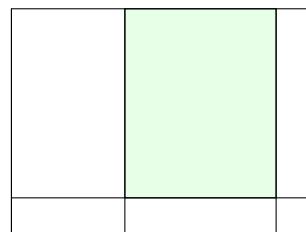
najväčší dielik je 40

[V 3](#)



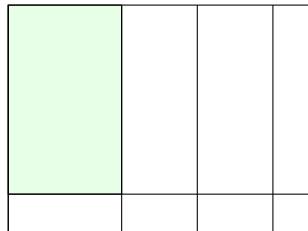
najväčší dielik je 25

[V 7](#)



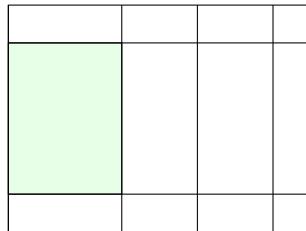
najväčší dielik je 20

[V 5](#)



najväčší dielik je 15

[H 5](#)



najväčší dielik je 12

**Úloha 107.** V prvom riadku vstupu sú tri čísla  $n, k, a$ , ktoré predstavujú súčiastku, ktorú treba testovať. Je to rúrka dlhá  $n$  a v nej je  $k$  valčekov dĺžky  $a$ , ktoré sa nedotýkajú. Napr. pre  $n = 16, k = 3, a = 3$  niektoré správne súčiastky môžu vyzerať takto:

## Vyhľadávacie stromy: <set> a <map> v STL

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Ďalej je dané číslo  $m$  a potom  $m$  pozícií (číslovaných od 1), na ktorých sa súčiastka testovala a zistilo sa, že na tej pozícii nie je valček. Napr. ak by sa testovali pozície 4, 5, 6, 10, 14, súčiastka by mohla vyzeráť ako vľavo hore. Keby ale na žiadnej z pozícii 4, 5, 6, 10, 14, 1 valček neboli, tak je jasné, že súčiastka je chybná. Napíš program, ktorý prečíta jednotlivé testy a vypíše, po ktorom teste začne byť jasné, že súčiastka je chybná.

Typy `set` a `map` sú sice naprogramované pomocou vyhľadávacích stromov, ale zámerom je, že spôsob, akým sú naprogramované, je vecou komplítora a teba ako programátora to nemá čo zaujímať; máš len zaručené, že operácie vkladania, vyhľadávania a mazania majú logaritmickú zložitosť. Preto ani nemáš v programe prístup priamo k uzlom stromu. Niekedy by sa to ale hodilo. Nasledovný príklad ľahko vyriešiš, ak upravíš náš AA strom `Tree` tak, že každý uzol si bude navyše pamätať, kolko listov je v jeho podstrome.

**Úloha 108.** Napíš program, ktorý vydelenie postupnosť príkazov: `pridaj x, uber x a ? x`, kde  $x$  je nejaké nezáporné celé číslo. Príkazy `pridaj` a `uber` pridávajú a uberajú čísla do/z množiny. Pre každý príkaz `? x` treba vypísať, kolko čísel menších ako  $x$  je momentálne v množine. Posledný príkaz je `!`.

**Úloha 109.** Napíš program, ktorý vydelenie postupnosť príkazov: `pridaj x, uber x a ? x`, kde  $x$  je nejaké nezáporné celé číslo. Príkazy `pridaj` a `uber` pridávajú a uberajú čísla do/z množiny. Pre každý príkaz `? x` treba vypísať  $x$ -té najmenšie číslo (brané od 0), ktoré sa v množine nenachádza. Posledný príkaz je `!`.

Napr. ak v množine sú čísla  $\{2, 3, 6, 8\}$  tak `? 0` vráti `0` (lebo 0 nie je v množine), `? 3` vráti `5` lebo v množine nie sú čísla 0, 1, 4, 5 a `? 5` vráti `9`.

## 28 Binárne súbory a kompresia

Doteraz sme na vstup a výstup používali objekty `cin` a `cout`, ktoré sú tzv. *streamy*. Predstavujú štandardný vstup a výstup, za obvyklých okolností sú to klávesnica a konzola, ale pri spúšťaní z príkazového riadka sa dajú presmerovať. Niekoľko ale chceme priamo z programu písat do súboru. Na to existuje knižnica `fstream` (`#include <fstream>`), v ktorej sú (okrem iného) definované triedy `ifstream` a `ofstream` na čítanie a zapísavanie z/do súborov. S podobnou triedou `stringstream` sme sa stretli na str. 79. Triedy `ifstream` a `ofstream` sa používajú podobne.

```
1 #include <fstream>
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     ifstream s("pako.txt");
7     int x = -1;
8     s >> x;
9     cout << s.good() << " "
10    << s.eof() << " "
11    << s.fail() << x << endl;
12 }
```

V tomto programe som vytvoril premennú `s` typu `istream` zviazanú so súborom `pako.txt` v aktuálnom adresári (t.j. tam, kde sa spúšťa binárka). Potom z neho skúšam načítať číslo. Trieda `istream` má metódu `bool good()`, ktorá vždy povie, či je stream pripravený a metódu `bool eof()`, ktorá hovorí, či bol dosiahnutý koniec súboru. Ďalšia užitočná funkcia je `fail()`, ktorá hovorí, či pri poslednom načítavaní nastala chyba. Vyskúšaj si spustiť tento program za rôznych okolností: ak súbor `pako.txt` neexistuje, ak existuje, ale na začiatku nemá číslo, ale písmeno, ak existuje, na začiatku má číslo a za ním nejaké znaky (napr. koniec riadka) a napokon ak existuje a je v ňom iba jedno číslo (bez konca riadka). Všimni si, že ak raz nastane chyba a stream prestane byť `good`, už taký ostane, a `eof` sa dosiahne, až keď sa pokúsi načítať za posledný znak v súbore. Ak chceš skončiť prácu so streamom, môžeš zavolať `s.close()`. Ak to nespravíš, všetko potrebné sa spraví v deštruktore (v našom prípade na konci programu). Nasledovný program postupne prečíta zo súboru všetky čísla a vypíše ich:

```
1 #include <fstream>
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     ifstream s("pako.txt");
7     int x = -1;
8     while (true) {
9         s >> x;
10        if (s.fail()) break;
11        cout << x << endl;
12    }
13 }
```

To isté sa dá docieliť aj jednoduchšie:

```
1 ifstream s("pako.txt");
2 int x = -1;
3 while (s >> x)
4     cout << x << endl;
```

Čo znamená to `while (s >> x)`? Podobne, ako sme v kapitole 22 pre nás typ `Tabuľka` definovali

```
1 ostream& operator <<(ostream& str , const Tabulka& o)
```

tak aj `ifstream::operator>>(...)` vracia referenciu `ifstream&`. Zároveň má `ifstream` definovaný

```
1 operator bool() {return !fail();}
```

ktorým sa dá hodnota `ifstream` pretypovať na `bool` (a výsledok je `true` práve vtedy, ak volanie `fail` vráti `false`). Takže zápis `while (s >> x)` hovorí *Prečítaj z s celé číslo, ulož ho do x a vráť referenciu na s. Príkaz while očakáva výraz typu bool, preto zavolaj pretypovací operátor z ifstream a použi výslednú hodnotu.* Cyklus sa teda vykonáva, kým bolo načítanie úspešné.

Pretože `ifstream` pracuje so súborom, nemusí, na rozdiel od `cin` čítať súvisle za sebou. Dá sa predstaviť tak, že po súbore sa pohybujie kurzor, ktorý určuje miesto, z ktorého sa práve ide čítať. Pozíciu kurzora vracia funkcia `tellg()`. Kurzor sa dá posúvať pomocou `seekg()`, ktorá má dva parametre: prvý určuje o kolko sa má kurzor posunúť a druhý odkiaľ sa má počítať: `ios::beg` od začiatku súbora, `ios::end` od konca, `ios::cur` od súčasnej pozície. To `ios::beg`, `ios::end`, `ios::cur` sú nejaké konštanty, ktoré sú definované v rámci `namespace std::ios`; nebudem teraz rozoberať, čo to presne je. Vyskúšaj si napísat súbor `pako.txt` s číslami, medzerami a prázdnymi riadkami a spustiť si tento program:

```
1 #include <fstream>
2 #include <iostream>
3 using namespace std;
4
5 int main() {
6     ifstream s("pako.txt");
7     int x = -1;
8     s.seekg(0, ios::end);
9     cout << "subor ma " << s.tellg() << " bytov." << endl;
10    s.seekg(0, ios::beg);
11    while (s >> x)
12        cout << x << " a kurzor je na " << s.tellg() << endl;
13 }
```

Trieda `ofstream` funguje podobne, s tým rozdielom, že vždy vytvorí nový súbor (ak súbor s daným menom existoval, tak ho prepíše) a bude doňho zapisovať. Kvôli efektívite sa nezapisuje ihneď, ale najprv do vyhrazeného pola v pamäti, ktoré sa do súboru zapíše až pri volaní `close`. Dá sa to ale urobiť aj skôr, zavolaním `flush()`. Ak nechceš, aby sa súbor premazal, môžeš použiť druhý parameter v konštruktore (resp. v `open`), ktorým môžeš nastaviť rôzne vlastnosti. Ak zavolás `ofstream s("pako.txt", ios::app);`, bude sa zapisovať na koniec existujúceho súboru (*append*). Presúvať kurzor sa dá rovnako, ale namiesto `seekg` a `tellg` treba<sup>1</sup> použiť metódy `seekp` a `tellp`.

Súbory sa, rovnako ako pamäť, dajú predstaviť ako pole nul a jednotiek. Ak máš premennú `int x = 42`, v pamäti sú bity nastavené podľa dvojkového zápisu: `00101010` (a ešte ďalších 7 bytov samých nul, lebo `int` má 8 bytov). Operátory `>>` a `<<` ale čítajú a píšu textovú reprezentáciu: ak máš `ofstream s`; a napíšeš `s << x`, do súboru sa napíšu dva byty: `00110100` (číze 52) a `00110010` (číze 50), čo sú ASCII kódy pre znak '`4`' a znak '`2`'. Typy `ifstream` a `ofstream` umožňujú zapisovať a čítať aj priamo byty<sup>2</sup> pomocou funkcií `read` a `write`. Tie dostávajú ako parameter pointer na `char` a číslo `n` a prečítajú resp. zapíšu `n` bytov do/z pamäte. Samozrejme,

<sup>1</sup>v skutočnosti netreba. Pre súborové streamy je vždy *put-pointer* a *get-pointer* to isté, takže `seekg` a `seekp` urobí to isté. Ale napr. pre `stringstream` to už neplatí, takže je dobrý zvyk *put-* a *get-pointer* rozlišovať.

<sup>2</sup>Ak pracuješ na inom systéme ako na Linuxe, operačný systém môže niektoré znaky (napr. koniec riadku) prekladať inak. To znamená, že sa do súboru zapíše viac/iných znakov. Ak tento preklad nechceš, treba pri otváraní súboru použiť prepínač `ios::binary`, napr. `ifstream is("pako.txt", ios::in | ios::binary);` alebo `ofstream os("pako.txt", ios::out | ios::binary);`. Všimni si, že na použitie viacerých prepínačov používam bitové OR. To je častá finta, ako v jednej premennej typu `int` zapamätať niekolko prepínačov: ak ich hodnoty sú mocniny dvojky, napr. `int a=1, b=2, c=4, d=8`; tak každý z nich má v dvojkovom zápisе iba jednu jednotku. Preto napr. `int x = b|d` je v dvojkovej sústave `0010 | 1000`, číze `1010`, t.j. 10. Preto pre `x=10` sú zapnuté prepínače `b` a `d`. Otestovať, či je niektorý bit nastavený sa dá napr. `if (x&d != 0) ...`

## Binárne súbory a kompresia

---

treba si dávať pozor, aby tá pamäť bola vhodne alokovaná, inak sa môže prepísať nejaká iná premenná (prípadne program spadne na segfault).

Vyskúšaj spustiť nasledovný program:

```
1 #include <fstream>
2 #include <iostream>
3 using namespace std;
4
5 void zapis(long int x) {
6     ofstream s("pako.txt", ios::out | ios::binary);
7     s.write((char *)&x, sizeof(x));
8     s.close(); // netreba, urobil by deštruktor
9 }
10
11 void citaj(long int &x) { // toto & znamená referenciu
12     ifstream s("pako.txt", ios::in | ios::binary);
13     s.read((char *)&x, sizeof(x)); // ale toto & znamená adresu
14     s.close(); // netreba, urobil by deštruktor
15 }
16
17 int main() {
18     long int x = 7020664791986103674;
19     zapis(x);
20     long int y;
21     citaj(y);
22     cout << y << endl;
23     cout << sizeof(y) << endl;
24 }
```

Teraz sa v nejakom editore pozri, čo sa vytvorilo v súbore pako.txt.

Najmenšia jednotka, ktorá sa dá zapisovať a čítať je jeden byte, t.j. 8 bitov. Aj typ `bool`, ktorému by inak stačil 1 bit, má v pamäti (a v súbore) veľkosť 1 byte. Nasledovný kus programu

```
1 bool a[40];
2 for (int i = 0; i < 20; i++) {
3     a[2 * i] = false;
4     a[2 * i + 1] = true;
5 }
6 ofstream s("pako.txt", ios::out | ios::binary);
7 s.write((char *)a, sizeof(a));
8 s.close();
```

preto do súboru `pako.txt` zapíše 40 bytov s hodnotami striedavo 0 a 1. Ak chceme zapisovať do súboru priamo byty, musíme to spraviť "ručne". Vyrobíme si triedu `BitWriter`, ktorá bude mať otvorený súbor na zápis a zároveň si bude pamätať buffer `buff` na zapisovanie bitov: to môže byť napr. 8-bitový `unsigned char` alebo 32-bitový `unsigned int` alebo niečo iné, čo budeme mať ako šablónu Buffer. Zároveň si budeme pamätať pozíciu bitu, ktorý práve nastavujeme ako `int sz`, takže ak máme pridať bit v na koniec, zvýšime `sz` a správime `(buff<<1)|(int)v`, čiže posunieme binárny zápis `buff` o 1 pozícii doľava a na posledné miesto zapíšeme pomocou OR hodnotu `v`. Ak zaplníme všetky byty, tak buffer zapíšeme do súboru a vynulujeme. Ostáva jeden problém: na konci treba do súboru zapísat zostávajúci buffer, ktorý doplníme nulami. Potom ale nevieme rozlísiť, koľko nul bolo doplnených. Môžeme to spraviť tak, že na začiatok súboru napíšeme napr. `long int len`, kde si budeme pamätať celkovú dĺžku. Ako ju napísať na začiatok? Najprv tam zapíšeme 0, aby sme mali v súbore urobené miesto a pred zatvorením súboru nastavíme kurzor pomocou `seekp` na začiatok a zapíšeme. Na zapisovanie bitu môžeme definovať `operator<<` a môžeme prihodiť aj verziu `<<` pre čísla, ktorá postupne zapíše všetky byty. Výsledok môže vyzerať nejak takto:

```

1 template <typename Buffer = unsigned int>
2 struct BitWriter {
3     ofstream f;
4     Buffer buff;
5     int sz;
6     long int len;
7     const int N = sizeof(Buffer);
8
9     BitWriter(const string &fname) {
10         f.open(fname, ios::out | ios::binary);
11         sz = 0;
12         buff = (Buffer)0;
13         len = 0;
14         f.write((char *)&len, sizeof(len));
15     }
16
17     ~BitWriter() {
18         if (sz > 0) { // zapíš do súboru zostávajúci buffer
19             buff <= 8 * N - sz;
20             f.write((char *)(&buff), N);
21         }
22         f.seekp(0); // na začiatok zapíšeme výslednú veľkosť
23         f.write((char *)&len, sizeof(len));
24         f.close();
25     }
26
27     BitWriter &operator<<(bool v) {
28         buff = (buff << 1) | ((int)v);
29         sz++;
30         len++;
31         if (sz == 8 * N) {
32             f.write((char *)(&buff), N);
33             sz = 0;
34             buff = (Buffer)0;
35         }
36         return *this;
37     }
38
39     template <typename T>
40     BitWriter &operator<<(const T &v) {
41         for (int i = 8 * sizeof(T) - 1; i >= 0; i--)
42             (*this) << ((v & (1 << i)) != 0);
43         return *this;
44     }
45 };

```

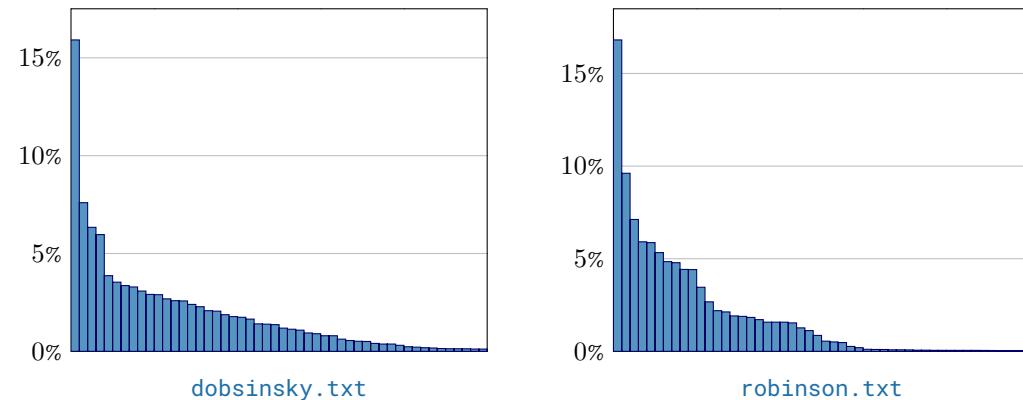
**Úloha 110.** Naprogramuj analogickú triedu `BitReader`, ktorá bude čítať (pomocou operátora `>>`) to, čo `BitWriter` zapísal. Pridaj `operator bool() {...}` ktorý vráti `true`, ak sa ešte nedočítalo do konca.

Nejaká verzia tried `BitWriter` a `BitReader` je v súbore `bitwriter.h`<sup>3</sup>

<sup>3</sup><https://github.com/pocestny/programovanie/raw/master/materialy/bitwriter.h>

## Projekt: pakovač

Asi poznáš rôzne programy<sup>4</sup>, ktoré zoberú veľký súbor a urobia z neho menší<sup>5</sup>. Ako to môže fungovať? Finta je v tom, že nemôže. Povedzme, že chceme vedieť každý súbor s dĺžkou aspoň 1 MiB (t.j.  $1024 \times 1024 = 1048576$  bytov) skrátiť o 10 bytov, t.j. vyrobiť kratší súbor, z ktorého vieme ten pôvodný zrekonštruovať. Je jasné, že ak z dvoch rôznych pôvodných súborov  $A, B$  dostaneme rovnaký skrátený súbor  $X$ , už je to celé v keli, lebo nevieme, či z  $X$  sa má zrekonštruovať  $A$  alebo  $B$ . Čiže pre rôzne pôvodné súbory potrebujeme mať rôzne skrátené súbory. Lenže všetkých súborov s dĺžkou  $N$  bytov je  $256^N$  (lebo jeden byte má  $2^8 = 256$  rôznych hodnôt) a o 10 bytov kratších súborov je iba  $256^{N-10}$ . Teda iba malý zlomok  $\frac{256^{N-10}}{256^N} = \frac{1}{256^{10}} \approx 0.0000000000000000000008271\%$  všetkých súborov vieme skrátiť. Zostávajúcu drívivú väčšinu  $\approx 99.99999999999999999999991729\%$  súborov neskrátime ani len o 10 bytov, nech by sme robili čokoľvek. Tak prečo tie programy fungujú? Súbory, ktoré používame, nie sú náhodné; väčšinu možných súborov nikto nikdy nevytvorí, takže je nám srdečne jedno, že sa nedajú skrátiť. Nám záleží na tom, aby sa dalo skrátiť tých zopár málo, ktoré používame. Zober si napr. súbor [demacek.txt](#)<sup>6</sup>: má 81571 bytov, ale používa iba 11 rôznych znakov: '.', ',', '-' , '=' , '+' , '\*' , '#' , '%' , '@' a k tomu medzera a koniec riadku. Pomocou jedného bitu vieme rozlíšiť dva znaky. Pomocou dvoch bitov štyri (lebo sú 4 kombinácie nuly a jednotky). Pomocou  $k$  bitov vieme rozlíšiť  $2^k$  znakov. Preto na uloženie textu, ktorý používa  $n$  rôznych znakov by nám malo stačiť použiť  $\log n$  bitov na zakódovanie jedného znaku. Nás súbor ale na uloženie jedného znaku používa byte, t.j. 8 bitov, aj keby stačilo  $\log(11) \approx 3.45$  bitu<sup>7</sup>. Keby sme si na začiatok súboru zaznačili znaky, ktoré používame, a potom používali iba ich čísla, súbor by sme skrátili na menej ako polovicu. Väčšina súborov, ktoré nás zaujímajú, ale používa veľa rôznych znakov, takže touto fintou veľa neušetríme. Ale pravda je, že v súboroch spravidla nie sú všetky znaky zastúpené rovnomerne. Napr. v slovenskom texte (s diakritikou) [dobsinsky.txt](#)<sup>8</sup> tvorí 18 najfrekventovanejších znakov<sup>9</sup> dokopy viac ako 80% textu. V anglickom teste [robinson.txt](#)<sup>10</sup> je to ešte výraznejšie, 18 najfrekventovanejších znakov<sup>11</sup> zaberá až viac ako 88% textu. Aj v rámci frekventovaných znakov sú veľké rozdiely, nasledujúce grafy ukazujú rozloženie početnosti najfrekventovanejších znakov:



<sup>4</sup>napr. [zip](#), [rar](#), [pkzip](#), [bzip2](#), [gzip](#), [7-zip](#) a mnohé iné

<sup>5</sup>väčšina z nich robí ešte veľa ďalších vecí, napr. že zabalí celý adresár do jedného súboru, ale to nás teraz nebude zaujímať

<sup>6</sup><https://github.com/pocestny/programovanie/raw/master/materialy/huff/demacek.txt>, Ondrej Demácek bol legendárny učiteľ informatiky, ktorý vychoval niekoľko generácií slovenských programátorov.

<sup>7</sup>Bit je predsa nula alebo jednotka, tak ako môžeme použiť 3.45 bitu? Myslime tým priemernú dĺžku na jeden znak. Ak by sme chceli zakódovať každý z 11 znakov, potrebovali by sme 4 byty na znak (3 nestácia, lebo nimi rozlíšime iba  $2^3 = 8$  znakov). Ak by sme si ale povedali, že budeme kódovať dvojznakové kombinácie, ktorých je  $11 \cdot 11 = 121$ , na jeden dvojznak by nám stačilo 7 bitov ( $2^7 = 128$ ), teda v priemere  $\frac{7}{2} = 3.5$  bitu na znak. Čím dlhšiu kombináciu znakov budeme kódovať naraz, tým viac sa budeme bližiť k hodnote 3.45 na znak.

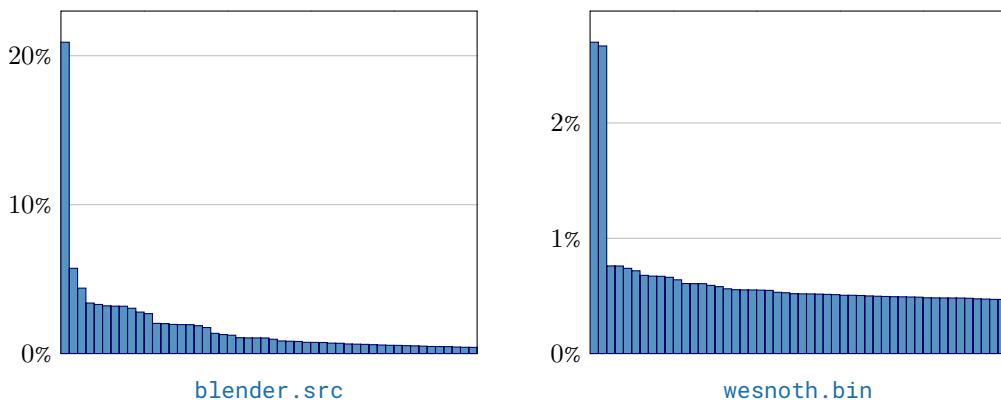
<sup>8</sup><https://github.com/pocestny/programovanie/raw/master/materialy/huff/dobsinsky.txt>, tri diely Dobšinského rozprávok zo servera <https://zlatyfond.sme.sk/>

<sup>9</sup>medzera, a, o, e, i, l, t, n, s, r, k, v, d, m, u, p, čiarka, h

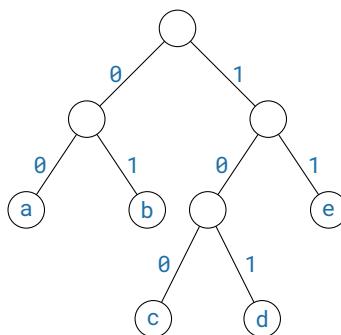
<sup>10</sup><https://github.com/pocestny/programovanie/raw/master/materialy/huff/robinson.txt>, román Robinson Crusoe zo servera <https://www.gutenberg.org>

<sup>11</sup>medzera, e, t, o, a, n, h, i, s, r, d, l, u, m, w, čiarka, c, f

Programy sa správajú podobne ako prirodzený jazyk, vľavo je frekvencia početnosti znakov pre súbor `blender.src`<sup>12</sup>, čo je zdrojový kód (v jazyku C++) grafického programu `blender`<sup>13</sup>. Pre skompliované binárky je situácia horšia (graf vpravo je súbor `wesnoth.bin`<sup>14</sup>, čo je hlavná binárka hry `The Battle for Wesnoth`<sup>15</sup>), ale aj tam sa niektoré byty vyskytujú častejšie ako iné.



Ako vieme jednoducho využiť nerovnomernosti v zastúpení jednotlivých bytov<sup>16</sup>? Namiesto toho, aby sme každý znak zakódovali 8-bitovým číslom, môžeme znakom priradiť kódy (postupnosti nul a jednotiek) rôznej dĺžky. Znakom, ktoré sa vyskytujú častejšie, chceme priradiť kratšie kódy. Treba si ale dávať pozor, aby sme to, čo zakódujeme, vedeli aj dekódovať. Napr. keby sme priradili kódy  $a \mapsto 01$  a  $b \mapsto 0101$ , tak nevieme, či  $0101$  má byť  $aa$  alebo  $b$ . Navýše chceme, aby sme zakódovaný súbor vedeli dekódovať efektívne. Povedzme, že máme kód  $a \mapsto 00$ ,  $b \mapsto 001$ ,  $c \mapsto 01$  a  $d \mapsto 10$ . Ked vidíme začiatok postupnosti  $0010101\cdots$  tak ju nevieme dekódovať, kým sa nepozrieme na koniec, lebo  $00101\cdots01$  sa dekóduje ako  $bccc\cdots c$ , ale  $00101\cdots010$  sa dekóduje ako  $addd\cdots d$ . Dobrý spôsob, ako zaručiť, že zakódovanú postupnosť budeme vedieť efektívne dekódovať, je na vrhnúť kódy tak, aby žiadny neboli začiatkom (*prefixom*) iného (napr. v našom poslednom príklade je kód  $a$  čka  $00$  prefixom kódu  $b$ čka  $001$ ). Každý takýto *bezprefixový* kód si vieme nakresliť do stromu: koreň stromu bude zodpovedať všetkým kódom, jeho ľavý syn všetkým kódom začínajúcim nulou a pravý syn kódom začínajúcim jednotkou. Napríklad tento strom:



reprezentuje kód, ktorý priradí  $a \mapsto 00$ ,  $b \mapsto 01$ ,  $c \mapsto 100$ ,  $d \mapsto 101$ ,  $e \mapsto 11$ . Uvedom si, že ak hodnoty jednotlivých znakov sú v listoch, tak kód je naozaj bezprefixový. Naopak, každému stromu vieme priradiť bezprefixový kód. Takže kód a strom je v podstate to isté. Naša úloha je teraz takáto: máme daný text a chceme k nemu nájsť taký kód (t.j. strom), že po zakódovaní dostaneme najkratší možný výstup. Ako to urobiť? Zoberme si nasledovný text<sup>17</sup>:

<sup>12</sup>[https://github.com/pocestny/programovanie/raw/master/materialy/huff/blender\\_src](https://github.com/pocestny/programovanie/raw/master/materialy/huff/blender_src)

<sup>13</sup><https://www.blender.org/>

<sup>14</sup><https://github.com/pocestny/programovanie/raw/master/materialy/huff/wesnoth.bin>

<sup>15</sup><https://www.wesnoth.org/>

<sup>16</sup>V slovenskom teste je vzťah bytov a znakov trochu zložitejší, lebo používa kódovanie UTF8, v ktorom jeden znak (napr. keď má diakritiku) môže zaberáť viac bytov; navyše, ten istý znak (*glyph*) sa dá niekedy zapísat rôznymi bytami. Nič z tohto nebudeme brať do úvahy a budeme pracovať s bytami.

<sup>17</sup>Nie je to práve literárny skvost, ale chcel som, aby tam bolo čo najmenej rôznych znakov.

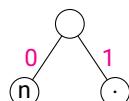
## Binárne súbory a kompresia

ema ma mamu. mama ma malu emu ale ema nema lamu. mama ma lamu. mama a  
ema melu malu lamu. mala lama nelame emu. emu lame eme lana. nela ema  
a lena melu lan. mama ma nulu. mama lame eme lamu. lama ma lunu. ela ma  
umelu lamu. lama ma mlela.

Jednotlivé znaky sa v texte vyskytujú takto:

'a'	51	21.4286 %
'_'	50	21.0084 %
'm'	47	19.7479 %
'l'	27	11.3445 %
'e'	24	10.084 %
'u'	19	7.98319 %
'.'	12	5.04202 %
'n'	8	3.36134 %

Ako bude vyzeráť dobrý strom? Určíte písmeno '**n**', ktoré sa v teste vyskytuje najmenej, dostane najdlhší kód. Prečo? Dajme tomu, že '**n**' dostane kód dĺžky 3, ale '**l**' dĺžky 5. Výskyty písmen '**n**' a '**l**' preto dokopy dajú  $8 \cdot 3 + 27 \cdot 5 = 159$  bitov. Keby som ale všetky ostatné kódy nechal tak, a vymenil kódy '**n**' a '**l**', písmená '**n**' a '**l**' budú dávať iba  $8 \cdot 5 + 27 \cdot 3 = 121$  bitov. Podobne druhý najdlhší kód bude mať druhý najzriedkavejší znak, '.'. Dva najdlhšie kódy v strome majú znaky, ktoré sú v dvoch najspodnejších listoch. Teraz si všimni, že vždy sa dajú nájsť dva najhlbšie listy, ktoré majú spoločného rodiča. To znamená, že zatiaľ sice neviem, ako bude výsledný strom vyzeráť, ale viem, že znaky '**n**' a '.' budú na spodku stromu napr. takto (nezáleží na tom, ktorý je vpravo a ktorý vľavo):

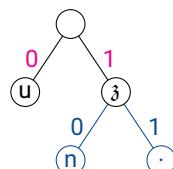


Ako bude vyzeráť zvyšok stromu? Viem, že '**n**' a '.' majú rovnaký kód, ktorý sa lísi iba tým, že jeden má na konci nulu a druhý jednotku. Takže viem, že výsledný text bude mať 8-krát nulu (z kódov '**n**') a 12-krát jednotku (z kódov '.'). Keď si odmyslím tieto posledné znaky, '**n**' a '.' sa správajú ako jedno písmenko, napr. '**ž**'. Preto ak chcem nájsť najlepší strom pre celý text, stačí mi nájsť najlepší strom pre nasledovný text:

ema ma mamuž mama ma malu emu ale ema žema lamuž mama ma lamuž mama a ema  
melu malu lamuž mala lama želame emuž emu lame eme lažaž žela ema a leža  
melu lažaž mama ma žuluzž mama lame eme lamuž lama ma lužuž ela ma umelu  
lamuž lama ma mlelaž

Spravím rovnakú úvahu: pozriem si početnosti jednotlivých znakov, zoberiem dva najmenej početné (v tomto prípade '**ž**' a '**u**') a aktualizujem strom

'a'	51	21.42866 %
'_'	50	21.0084 %
'm'	47	19.74799 %
'l'	27	11.34454 %
'e'	24	10.084 %
'ž'	20	8.403368 %
'u'	19	7.98319 %

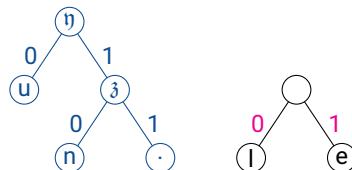


Opäť v teste spojím znaky '**ž**' a '**u**' do nového znaku '**ň**' a dostanem text

ema ma mamij mama ma malij emj ale ema nema lamij mama ma lamij mama a  
 ema melij malij lamij mala lama nelame emij emj lame eme lajaj nela ema  
 a leja melij lajij mama ma nyljij mama lame eme lamij lama ma lnyjij ela ma  
 nmelij lamij lama ma mlelaj

Zistím si početnosti a zarádím dva najmenej početné znaky do stromu:

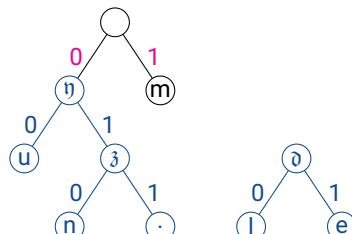
'a'	51	21.42866 %
'u'	50	21.0084 %
'm'	47	19.74799 %
'n'	39	16.3866 %
'l'	27	11.3445 %
'e'	24	10.084 %



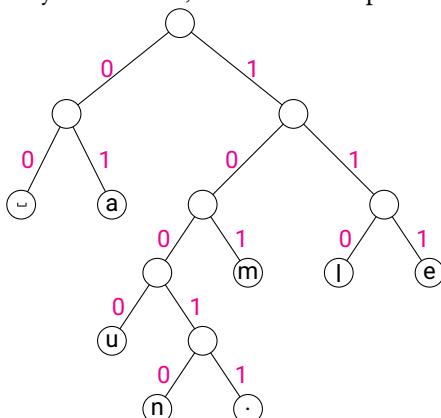
Teraz som dostał dva nesúvislé kusy stromu, ale to mi nijak nevadí: viem, že vo výsledom strome bude kdesi visieť jeden kus a kdesi inde druhý. Pokračujem rovnako ďalej, spojím 'l' a 'e' do jedného znaku 'ð':

ðma ma mamij mama ma madij ðmij add ðma ñðma ðamij mama ma ðamij mama a  
 ðma mððij madij ðamij mama ðama ñððam ðmij ðmij ðam ðm ðða ða ðma  
 a ððja mððij ða ðjij mama ma ñððjij mama ðam ðm ðamij ðama ma ðjij ðða ma  
 ñððjij ðamij ðama ma mðððaj

'a'	51	21.42866 %
'ð'	51	21.4286 %
'u'	50	21.0084 %
'm'	47	19.74799 %
'n'	39	16.3866 %



Takto budem postupne zdola nahor vytvárať strom: vždy spojím dve najmenej početné znaky do jedného a zlepím príslušné kusy stromu spoločným vrcholom, až dostanem napr. strom:



Táto metóda je známa pod menom *Huffmanovo kódovanie*. Poďme ju teraz skúsiť naprogramovať. Najprv sa treba rozhodnúť, ako budeme v pamäti reprezentovať strom, ktorý vytvárame. Spravme si takýto typ:

```

1 struct Item {
2     unsigned char val;

```

## Binárne súbory a kompresia

```
3     int par = -1, freq = 0;
4     int l = -1, r = -1;
5 }
```

v ktorom si o každom znaku budeme pamätať jeho hodnotu (`char val`) a početnosť (`int freq`). Tieto znaky budeme mať uložené v poli a hodnoty `par`, `l`, `r` budú indexy rodiča, ľavého a pravého syna. Takže strom z obrázka hore by mohol vyzerať napr. takto:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
val	'1'	'.'	'u'	'n'	'_'	'a'	'm'	'e'							
freq	27	12	19	8	50	51	47	24	20	39	51	86	101	137	238
par	10	8	9	8	12	12	11	10	9	11	13	13	14	14	-1
l	-1	-1	-1	-1	-1	-1	-1	-1	1	8	7	6	5	10	13
r	-1	-1	-1	-1	-1	-1	-1	-1	3	2	0	9	4	11	12

Ako takéto pole vyrobíme? Začiatok je ľahký: prečítame vstupný súbor do poľa znakov, prejdeme cez neho a pre každý znak (je ich iba 256 možných) si v pomocnom poli `freq` budeme pamätať, kolkočrát sa tam príslušný znak vyskytuje. Nakoniec pre každý znak, ktorý má nenulový počet výskytov pridáme jeden `item`:

```
1 ifstream f(src, ios::in | ios::binary);
2 f.seekg(0, ios::end);
3 vector<unsigned char> text(f.tellg());
4 f.seekg(0, ios::beg);
5 f.read((char*)(text.data()), text.size());
6
7 vector<int> freq(256, 0);
8 for (unsigned char c : text) freq[c]++;
9
10 vector<Item> items;
11 for (int i = 0; i < 256; i++) if (freq[i] > 0)
12     items.push_back({(unsigned char)i, -1, freq[i], -1, -1});
```

Ďalej budeme potrebovať opakovane spájať najmenej početné znaky. Budeme na to potrebovať dátovú štruktúru, kde vieme 1) vložiť znak a 2) odobrať znak s najmenšou početnosťou. Takáto dátová štruktúra sa volá *halda* (*heap*) a ľahko by sme si ju vedeli naprogramovať v poli, ale keď máme v ruke kladivo (typ `set`), tak všetko vyzerá ako klinec: spravíme si premennú `heap` typu `set`. Treba si ale premyslieť, čo budeme do nej ukladať. Na jednej strane potrebujeme, aby boli prvky usporiadané podľa početnosti. Na druhej strane, keď zoberieme najmenej početný znak (hodnotu `*heap.begin()`), potrebujeme na základe tej vedieť upraviť pole `items`. Použijeme jednoduchú fintu, ktorú umožňuje verzia typu `set` s porovnávacou funkciou: v premennej `heap` si budeme ukladať iba indexy do poľa `items` a porovnávacia funkcia sa bude rozhodovať podľa hodnoty `items[i].freq`:

```
1 multiset<int, function<bool(int, int)>> heap(
2     [&items](int i, int j) {
3         return items[i].freq < items[j].freq;
4     });

```

Tento zápis hovorí, že používam šablónu na ukladanie typu `int` s tým, že v konštruktore pridám porovnávaciu funkciu typu `function<bool(int, int)>`, t.j. lambdu, ktorá zoberie dva parametre typu `int` a vráti `bool`. Vzápäť takú premennú s menom `heap` vyrábim, a v konštruktore jej pridám lambdu, ktorá má ako *capture* referenciu<sup>18</sup> na pole `items`: `[&items](int i, int j) { ... }`. Preto v tele lambdy viem pristupovať k poľu `items` a porovnávať dva prvky `i` a `j` podľa početností znakov `items[i].freq` a `items[j].freq`. Tu si ale

<sup>18</sup>to je dôležité: zápis `[&items](int i, int j) { ... }` by do lambdy zobrať hodnotu `items`, t.j. lokálnu kópiu poľa `items`, ako vyzeralo v čase volania konštruktora

treba dať dobrý pozor a uvedomiť si čo sa deje: v type `set` sa pracuje s vyvažovaným stromom, kde v uzloch sú uložené čísla. Vždy, keď treba dve čísla porovnať, tak namiesto `<` sa zavolá príslušná lambda. To znamená, že ak by sme zmenili hodnoty `freq` v poli `items`, strom, ktorý je vyrobený v `heap` už nebude správny. Preto hodnoty v `items` musíme meniť tak, že najprv odoberieme z `heap` index, ktorého frekvenciu chceme zmeniť, potom zmeníme frekvenciu, a potom index znova pridáme do `heap`, takže všetky porovnania, ktoré sa v strome `heap` robia, budú konzistentné. Ešte si treba uvedomiť, že v `heap` sice budeme mať uložené rôzne čísla (indexy do `items`), ale porovnávame ich podľa našej funkcie, a keďže môžeme mať viacero znakov s rovnakou frekvenciou, potrebujeme použiť typ `multiset`. Pri vytváraní poľa `items` si zároveň pripravíme pole `idx`, ktoré použijeme neskôr pri samotnom vytváraní kódu: pre každý znak c mám v `idx[c]` uložené, kde v `items` je príslušný list. Celý kus kódu môže vyzerať takto:

```

1 vector<int> idx(256, -1);
2 for (int i = 0; i < 256; i++) if (freq[i] > 0) {
3     int n = items.size();
4     items.push_back({(unsigned char)i, -1, freq[i], -1, -1});
5     heap.insert(n);
6     idx[i] = n;
7 }
8 int n_chars = items.size(); // počet listov stromu
9
10 while (heap.size() > 1) {
11     int i = pop(heap);
12     int j = pop(heap);
13     int n = items.size();
14     items.push_back({0, -1, items[i].freq + items[j].freq});
15     items[i].par = items[j].par = n;
16     heap.insert(n);
17 }
```

Funkcia `pop` je len jednoduchá pomocná funkcia, aby som ušetril písanie, napr. takto:

```

1 template <typename T>
2 int pop(T &heap) {
3     int u = *heap.begin();
4     heap.erase(heap.begin());
5     return u;
6 }
```

Zatiaľ máme v poli `items` nastavených rodičov, ale ešte sme nenastavovali položky `l` a `r`. Tie môžeme nastaviť napr. takto:

```

1 for (int i = 0; i < items.size(); i++) {
2     if (items[i].par != -1) {
3         if (items[items[i].par].l == -1)
4             items[items[i].par].l = i;
5         else
6             items[items[i].par].r = i;
7     }
```

Ked máme hotové pole `items`, chceme pre každý znak vyrobiť jeho kód. To je jednoduché: postupujeme smerom k rodičovi a podľa toho, či to bol ľavý alebo pravý syn, priradíme 0 alebo 1. Nakoniec kód obrátíme, lebo ho chceme mať smerom od koreňa k listom:

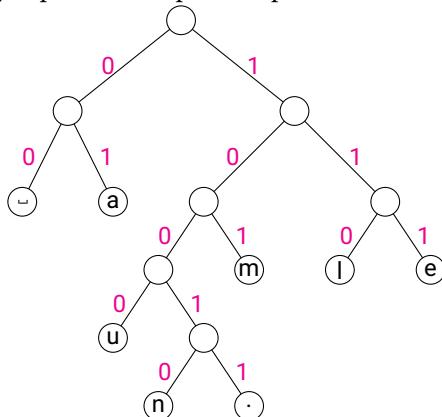
```

1 vector<vector<bool>> code(n_chars);
2
3 for (int i = 0; i < n_chars; i++)
4     for (int j = i; items[j].par != -1; j = items[j].par)
```

## Binárne súbory a kompresia

```
5     if (items[items[j].par].l == j)
6         code[i].push_back(true);
7     else
8         code[i].push_back(false);
9
10    for (auto &x : code) reverse(x.begin(), x.end());
```

Do výstupného súboru potrebujeme zapísat kód každého znaku zo vstupu tak, ako idú za sebou. Potrebujeme ale nejak zapísat aj "slovník", t.j. pre každý možný znak zapísat jeho kód. Ten chceme zapísat tak, aby sme použili čo najmenej bitov a pritom bolo jednoduché ho zapísat aj prečítať. Dá sa to urobiť napr. tak, že zapíšeme rekúrzívne celý strom: ak sme v liste, zapíšeme bit **1** a za ním znak (8 bitov). Ak sme vo vnútornom vrchole, zapišeme bit **0** a za ním rekúrzívny zápis ľavého a pravého podstromu. Strom z nášho príkladu:



by sme zapísali **001\_1a0001u01n1.1m01l1e**. Pri dekódovaní najprv prečítame slovník do premennej typu **\*Node**

```
1 struct Node {
2     unsigned char val;
3     Node *l, *r;
4     Node() : l(nullptr), r(nullptr) {}
5     ~Node() {
6         if (l) delete l;
7         if (r) delete r;
8     }
9 };
```

Samotné dekódovanie je priamočiare: čítame bity, prechádzame stromom a vždy, keď prídeme do listu, vypíšeme príslušný znak:

```
1 void decompress(const string &src, const string &dst) {
2     BitReader<> in(src);
3     ofstream dstf(dst, ios::out | ios::binary);
4     Node *root = readTree(in);
5     for (Node *p = root; in;) {
6         bool v;
7         in >> v;
8         if (v)
9             p = p->l;
10        else
11            p = p->r;
12        if (p->l == nullptr) {
13            dstf.write((char *)&(p->val), 1);
14            p = root;
```

```

15    }
16    }
17 }
```

Nakoniec by sme chceli vyrobiť program, ktorý vie kódovať a dekódovať podobne, ako štandardné programy, teda funguje z príkazového riadka (*CLI, command line interface*). Zatial vieme skomplilovať program, ktorý ked sa spustí, číta zo štandardného vstupu alebo zo súboru. Ale veľakrát vidíš programy, ktoré čítajú parametre priamo pri spustení, napr. `gzip -k subor.txt` Tieto parametre operačný systém posielá programu ako parametre funkcie `main`. Ak v programe namiesto `int main() ...` napišeš `int main (int argc, char** argv)...` premenná `argc` bude obsahovať počet argumentov a premenné `argv[0], argv[1],...,argv[argc-1]` postupne všetky parametre (ako refazce znakov `char *` alokované kdeši v systémovej pamäti) s tým, že `argv[0]` je názov binárky, t.j. pre príklad `gzip -k subor.txt` je `argc==3, argv[0]==“gzip”, argv[1]==“-k” a argv[2]==“subor.txt”`.

**Úloha 111.** Naprogramuj Huffmanovu kompresiu. Program má dostať ako parametre zdrojový súbor a cieľový súbor. Ak je spustený s prepínačom `-d` má dekódovať, inak kódovať.

Po naprogramovaní vyzerajú veľkosti súborov takto:

súbor	pôvodne (B)	Huffman (B)	bit/znak	kompresia
<code>blender.src</code>	54259819	35354132	5.213	65.16 %
<code>demacek.txt</code>	81571	31013	3.042	38.02 %
<code>dobsinsky.txt</code>	2045382	1266856	4.955	61.94 %
<code>robinson.txt</code>	659018	369272	4.483	56.03 %
<code>wesnoth.bin</code>	22159352	16074583	5.803	72.54 %

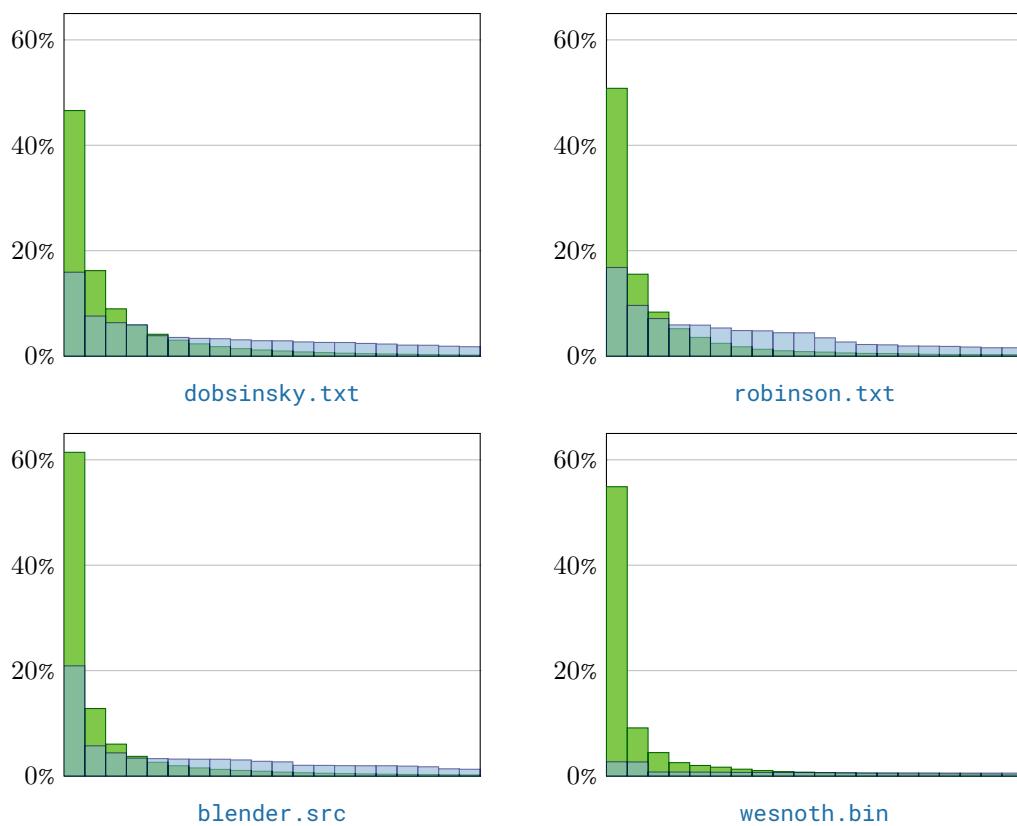
Jednou z nevýhod tohto kódovania je, že neberie do úvahy závislosti medzi znakmi. Napr. v slovenčine je '`y`' pomerne častý znak, ale po '`š`' nejde takmer nikdy. Preto '`y`' by mohlo dostávať rôzne dlhé kódy v závislosti od toho, čo bolo pred ním. Mohli by sme to urobiť tak, že si vyrobíme veľa nezávislých Huffmanových kódov: pre každé predchádzajúce písmenko (tzv. *kontext*) jeden a pri kódovaní a dekódovaní vždy použijeme ten správny. Problém je, že ak by sme chceli mať kontext dlhší, napr. 2, 3, či viac predchádzajúcich písmen, počet Huffmanovských stromov (a teda aj slovníkov, ktoré musíme mať zapísané v súbore) bude veľmi narastať. Iná možnosť, ako docieliť podobný efekt, je z takejto úvahy: ak by som mal program (*prediktor*), ktorý vie vždy z doterajšieho textu povedať, aké písmenko bude nasledovať, tak netreba nič komprimovať, prediktor mi súbor rovno vygeneruje. Nedúfam, že taký prediktor budem mať, ale čo ak by som mal nie celkom dokonalý, ale stále dosť dobrý prediktor? Taký, ktorý dosť často trafi, aké písmenko bude nasledovať, a ak sa aj netrafi, nebude od skutočného písmenka moc ďaleko. Na zrekonštruovanie pôvodného textu by som potom nepotreboval kódovať znaky zo vstupu, stačilo by mi pre každý vstupný znak zakódovať chybu, ktorú prediktor spravil. Na základe toho by som už vedel znak zo vstupu zrekonštruovať<sup>19</sup>. Ak je prediktor dobrý, môžem dúfať, že chyby budú väčšinou nuly, a teda sa budú dať skomprimovať lepšie, ako pôvodný súbor. Ako spraviť prediktor? Môžem si napríklad pre každý kontext (niekoľko predchádzajúcich písmeniek) pamätať znaky usporiadane podľa početnosti, koľkokrát som ktorý v danom kontexte videl. Prediktor potom vráti poradie znaku v tomto usporiadani. Ak je napr. kontext `nsk`, a pamätám si usporiadanie písmen `ý, é, á, ú, o, a, ...` a na vstupe je ako ďalšie písmenko `o`, prediktor vráti 4, lebo to je poradie `o` v tomto kontexte.

**Úloha 112.** Naprogramuj takýto prediktor.

Ak sa pozrieš, ako tento prediktor funguje pre kontext troch písmeniek, vidno, že po predspracovaní súborov sa frekvencie znakov výrazne zmenia (priesvitné modré stĺpce sú pôvodné frekvencie bez prediktora, zelené s prediktorem)

<sup>19</sup>Pri kódovaní sa vždy pozriem, aké písmenko predikuje prediktor a aké písmenko je v skutočnosti na vstupe. Chybu zakódujem na výstup a aktualizujem prediktor. Pri dekódovaní je to podobné: pozriem sa, čo hovorí prediktor, zo vstupu dekódujem chybu, z nej vyrobím skutočné písmenko, vypíšem ho na dekódovaný výstup a zároveň pomocou neho aktualizujem prediktor.

## Binárne súbory a kompresia



Ked' teraz Huffmanovým pakovačom skomprimujem výsledky prediktora, dostnem lepšiu kompresiu:

súbor	Huffman		Huffman s prediktorom		gzip	
	bit/znak	kompresia	bit/znak	kompresia	bit/znak	kompresia
<code>blender.src</code>	5.213	65.16 %	2.451	30.63 %	1.558	19.48 %
<code>demacek.txt</code>	3.042	38.02 %	1.36	17 %	1.207	15.09 %
<code>dobsinsky.txt</code>	4.955	61.94 %	3.025	37.81 %	3.085	38.56 %
<code>robinson.txt</code>	4.483	56.03 %	2.91	36.38 %	2.993	37.41 %
<code>wesnoth.bin</code>	5.803	72.54 %	3.708	46.35 %	2.673	33.41 %

Ako vidíš, prediktor výrazne pomohol kompresii, v niektorých prípadoch je výsledok lepší ako `gzip`<sup>20</sup>. Väčšina kompresných algoritmov používa buď Huffmanovo kódovanie kombinované s tzv. *run-length encoding*<sup>21</sup> alebo algoritmy, ktoré kódujú nie jednotlivé znaky, ale celé skupiny znakov (napr. `LZ77`<sup>22</sup>).

S kompresiou sa môžeš v rámci projektu hrať ďalej: ak vymyslíš lepší prediktor, kompresia bude lepšia. Zaujíma vás súťaž je [Hutter Prize](#)<sup>23</sup>, kde môžeš vyhrať cenu, ak skomprimuješ jeden konkrétny súbor (úryvok z wikipédie) lepšie ako ostatní. Ak vieš, že v súbore je anglický text, prediktor môže vychádzať zo slovníka, štruktúry vety, .... Kedže ide o to, skomprimovať konkrétny súbor, čo ti bráni celý súbor vložiť priamo do programu ako jeden veľký `cout << "cely_text_co_mam_vypisat"`? Nič, môžeš to tak urobiť, ale hodnotí sa celková skomprimovaná dĺžka vrátane binárky, ktorá rozbaluje (v skutočnosti tam pošleš len binárku, ktorá keď sa spustí, má ten konkrétny súbor vygenerovať). No a binárka so zapísaným dlhým stringom bude veľká.

<sup>20</sup>aj keď beží o dosť pomalšie

<sup>21</sup>Huffmanov kód z princípu nevie urobiť menšiu ako 8-násobnú kompresiu, lebo na každý znak použije aspoň 1 bit. Ak je vo vstupnom súbore dlhá postupnosť rovnakých znakov, napr. `000000000000`, je lepšie ju nahradíť [13]0 s významom *trinásťkrát zopakuj nulu*.

<sup>22</sup>[https://en.wikipedia.org/wiki/LZ77\\_and\\_LZ78](https://en.wikipedia.org/wiki/LZ77_and_LZ78)

<sup>23</sup><http://prize.hutter1.net/>

Náhoda a pravdepodobnosť 29

Čo je to náhoda? Dalo by sa povedať, že niečo (napr. keď hodíme kockou) je náhodné, ak sa nedá dopredu povedať, ako to dopadne. Existuje náhoda? To je ľažká otázka. Ako totiž rozlíšim, či sa niečo dopredu nedá povedať, alebo to iba neviem? Ak sa traja kamaráti dohodnú, že zavolajú štvrtému v priebehu jednej minúty, jemu to bude pripadať ako veľká náhoda, že z celého dňa si všetci vybrali akurát túto minútu. Aj pri hádzaní kockou by klasická fyzika povedala, že ak presne viem všetky parametre (veľkosť a hustota kocky, tvar ruky, silu a smer trasenia, odpor vzduchu,...) tak viem dopredu vypočítať, čo na kocke padne. Preto sa fyzici dlho domnievali, že náhoda neexistuje, sú len tzv. skryté parametre. S nástupom kvantovej fyziky sa pohľad zmenil a väčšina si skôr myslí, že náhoda (t.j. javy, ktoré sa dopredu naozaj nedajú predpovedať, lebo od ničoho predtým nezávisia) existuje. Ako je to naozaj sa asi nikdy nedozvieme, ale to nám nebráni s náhodou počítať. Keď vieme, že o náhodnom javе sa dopredu nedá nič povedať, znamená to, že keď vidíme výsledok, dozvieme sa niečo, čo sme predtým nemohli vedieť (získame informáciu). Napr. ak mám mincu, na ktorej úplne náhodne môže padnúť 0 alebo 1, a vidím, že padla 1, dozviem sa 1 bit informácie. Ak by som mal mincu, na ktorej stále padá 1, nedozviem sa nič, čo by som už dopredu nevedel. Ak mám mincu, kde nula padá veľmi zriedka a vidím, že padla 1, niečo sa dozviem, ale je to menej ako 1 bit. Ak si pripomeneš predchádzajúcu kapitolu, tak prediktor bol spôsob, ako z doterajšieho vstupu povedať niečo o budúcom. Ak mám dobrý prediktor, viem veľa predpovedať, teda sa málo informácie dozviem a vstup je málo náhodný. A naopak. Takže ak mám mincu, kde stále padá 1, budem ňou dlho hádzať a budem si písat výsledky, dostanem takúto postupnosť:

na ktorú mám 100% prediktor. Ak som si skúsil vygenerovať súbor 10 MB jednotkových bitov, [gzip](#) ho spakoval na 10 216 B. Keby som mal úplne náhodnú mincu, výsledky hodov by mohli vyzeráť napr. takto

a žiadom prediktor by mi nepomohol. A 10 MB náhodných bitov **gzip** spakoval na 10 487 390 B (teda vôbec). Nakoniec, keby som mal mincu, kde jednotka padá iba v 1% prípadov, výsledky by vyzerali nejak takto:

Prediktor, čo vždy predikuje nulu je veľmi dobrý – pomýli sa iba v 1% prípadov. A aj [gzip](#) spakuje 10 MB takýchto bitov na 1 343 524 B.

To znamená, že sice neviem, či nejaká skutočne náhodná postupnosť bitov existuje, ale viem povedať, že ak by existovala, tak žiaden prediktor mi nepomôže skomprimovať ju. Z toho viem napr. povedať, že keď by som veľakrát hádzal náhodnou mincou, tak vo výsledku je zhruba rovnako veľa núl a jednotiek. Ak by tam bolo viac jednotiek, tak prediktor, čo hovorí vždy 1, by mi pomohol.

S náhodou súvisí aj *pravdepodobnosť*. Kým náhodnosť je vlastnosť nejakého procesu (napr. náhodné hádzanie kocky), pravdepodobnosť je vlastnosťou nejakého javu (napr. že padne číslo deliteľné troma).

Ak mám náhodný proces (napr. hádzanie kockou), ktorý môže mať  $n$  rôznych výsledkov (napr. na kocke môže padnúť 6 rôznych čísel) a mám jav, ktorý nastane pri  $k$  z nich (napr. číslo deliteľné troma na kocke je len 3 a 6, t.j.  $k = 2$ ), tak poviem, že pravdepodobnosť tohto javu je  $\frac{k}{n}$ .

Z toho, čo sme hovorili, je zrejmé, že ak mám jav s pravdepodobnosťou napr. 0.01, tak pri veľkom počte opakovania budem ten jav vidieť v 1% prípadov.

Napr. aká je pravdepodobnosť, že na kocke padne šestka? Jedna šestina, t.j. asi 0.167. Ked veľakrát hádžem kockou, v šestine prípadov hodím šestku. Aká je pravdepodobnosť, že keď hodím dvakrát, obidva razy hodím šestku? Všetkých možných výsledkov dvoch hodov je  $6 \cdot 6 = 36$  a dve šestky sú iba v jednom z nich, t.j. pravdepodobnosť je  $\frac{1}{36} = \frac{1}{6} \cdot \frac{1}{6} \approx 0.0278$ . Aká je pravdepodobnosť, že hodím šestku päťkrát za sebou?  $\left(\frac{1}{6}\right)^5 \approx 0.00013$ . To platí vždy: ak mám náhodný jav s pravdepodobnosťou  $p$  a zopakujem ho  $k$ -krát, tak pravdepodobnosť, že nastane vždy, je  $p^k$ . Ale stále platí to, že náhodný jav nijak nezávisí od minulosti: ak hádžem kockou a päťkrát za sebou mi padne šestka, aká je pravdepodobnosť, že padne aj šiestykrát? Zdalo by sa, že to bude strašne malá pravdepodobnosť. Ale kocka si žiadnu históriu nepamäta. Čo bolo, bolo, zase je to len jedna šestina.

Je to veľmi prekvapivé, ale náhoda nám v skutočnosti môže pomôcť niečo rátať. Predstav si, že Bob a Bobek majú každý na disku veľký súbor, povedzme 1 TB<sup>1</sup>. Tie súbory by mali byť rovnaké, ale oni si nie sú istí, či sa náhodou medzičasom nezmenili, tak by to potrebovali overiť. Jedna možnosť je, aby Bob posielal Bobkovi po sieti svoj súbor a Bobek u seba oba súbory porovná. Ale posielat 1 TB po sieti je priveľa. Spravia teda toto. Bobov súbor je postupnosť  $8 \cdot 2^{40} = 2^{43}$  núl a jednotiek (1 byte má 8 bitov). Bob si predstaví, že celý súbor je jedno naozaj obrovské číslo  $N$  zapísané v dvojkovej sústave.  $2^{43} = 8796093022208$ , teda  $N$  môže byť až  $2^{2^{43}} = 2^{8796093022208}$ , čo je v desiatkovej sústave zhruba jednotka a za ňou 2.6 milióna núl. To je sice veľa, ale to nevadí. Bob si vymyslí náhodné 64-bitové prvočíslo<sup>2</sup>  $p$  a vyráta zvyšok z  $N$  po delení  $p$  (ktorý si označí ako  $z$ ). To sa dá naprogramovať rovnako, ako keď počítas delenie na papier – súborom stačí raz prejsť jedným cyklom.

**Úloha 113.** Naprogramuj funkciu, ktorá pre dané meno súboru a číslo  $p$  zráta zvyšok po delení čísla  $N$  číslom  $p$ , kde  $N$  je (veľmi veľké) číslo, ktoré predstavuje daný súbor.

Namiesto celého súboru pošle Bobkovi iba  $p$  a  $z$ , t.j. dve 64-bitové čísla. Bobek si predstaví svoj súbor ako obrovské číslo  $N'$  a vypočíta zvyšok  $z'$  po delení  $p$  a porovná  $z$  a  $z'$ . Ak sa rovnajú, vyhlási súbory za rovnaké, ak nie, povie, že sú rôzne.

Bude takéto porovnanie fungovať? Nie vždy. Ak sú súbory skutočne rovnaké, t.j. ak  $N = N'$ , tak pochopiteľne nech je  $p$  hocjaké, tak aj zvyšok po delení bude rovnaký. Ale môže sa stať, že súbory sú rôzne a napriek tomu vyjdu po delení rovnaké zvyšky. Aká je pravdepodobnosť, že sa Bobek na konci pomýli? Ak  $N$  a  $N'$  majú po delení  $p$  rovnaký zvyšok, potom  $N - N'$  má po delení  $p$  zvyšok 0. Aj  $N$  aj  $N'$  sú  $2^{43}$ -bitové čísla, preto aj ich rozdiel je  $2^{43}$ -bitové číslo. Ak označíme  $n = 2^{43}$ , tak  $N - N' \leq 2^n$ . Predstavím si prvočíselný rozklad  $N - N' = p_1 \cdot p_2 \cdots p_z \leq 2^n$ . Každé  $p_i$  je aspoň 2, preto  $z < n$  (keby bolo v rozklade viac ako  $n$  prvočísel, ich súčin by bol väčší ako  $2^n$ ). Bobek sa pomýli, ak  $p$  delí  $N - N'$ , t.j. ak Bob nešťastnou náhodou za  $p$  zvolil jedno z tých najviac  $n$  prvočísel z rozkladu  $N - N'$ .

Napríklad, ak by králici mali takéto (krátke) súbory:

Bobov súbor:

	P	r	a	l		s	i		v	o		v	a	n	i	?
ASCII	80	114	97	108	32	115	105	32	118	111	32	118	97	110	105	63

Bobkov súbor:

	K	r	a	l	i	c	i		n	a		v	i	n	e	!
ASCII	75	114	97	108	105	99	105	32	110	97	32	118	105	110	101	33

tak

$$\begin{aligned} N &= 106932137465082389165385936127520565567 \\ N' &= 100285997508730872704130855374389404961 \\ N - N' &= 6646139956351516461255080753131160606 = \\ &= 2 \cdot 7 \cdot 3323 \cdot 3571161089 \cdot 40003838440072490937307 \end{aligned}$$

V rozklade  $N - N'$  je len 5 prvočísel, z nich iba 4 sa zmestia do 64 bitov. V tomto príklade sa preto Bobek pomýli iba pri štyroch volbách  $p$ .

<sup>1</sup>To je 1000 GB, alebo  $2^{40}$  bytov, takže na uloženie dĺžky takého súboru už nestačí 32-bitové číslo, preto budeme používať 64-bitové.

<sup>2</sup>To sa zdá ako fažká úloha, ale keby Bob mal k dispozícii mincu, ktorá mu dáva náhodne nuly a jednotky, tak s trochou šikovnej matematiky, ktorú tu teraz nebudem ukazovať, sa to dá naprogramovať pomerne ľahko.

Pravdepodobnosť, že Bobek sa pomýli, je teda najviac  $n/P$ , kde  $n$  je dĺžka súboru v bitoch a  $P$  je počet 64-bitových prvočísel. Aby sme túto pravdepodobnosť vedeli odhadnúť, potrebujeme vedieť, koľko je 64-bitových prvočísel. Presne to ľahko povedať, ale platí<sup>3</sup>, že pre hocjaké číslo  $x$ , počet prvočísel menších ako  $x$  (čo sa zvykne označovať ako  $\pi(x)$ ) je  $\pi(x) > \frac{x}{\log(x)}$ . Keď si za  $x$  dosadíme  $2^{64}$ , dostaneme, že 64-bitových prvočísel je viac ako  $415828534266394360 \approx 2^{58.5}$ . Preto ak si Bob vyberie náhodné 64-bitové prvočíslo, tak pravdepodobnosť, že Bobek sa pomýli, bude najviac  $\frac{n}{415828534266394360}$ . Pre terabajtový súbor je  $n = 2^{43}$ , takže po dosadení máme pravdepodobnosť chyby  $\approx 0.0000211531$ . Preto ak by porovnávali veľa súborov, tak najviac zhruba 2 z 10000 budú porovnané zle. To je pomerne málo, ale čo ak je ten súbor tak dôležitý, že Bob a Bobek nechcú riskovať ani takúto malú chybu? Bob môže celý postup zopakovať dvakrát, to znamená, že Bobkovi pošle štyri 64-bitové čísla  $p, z, p', z'$ . Bobek vie, že ak sú súbory rovnaké, tak v obidvoch prípadoch mu musí vyjsť rovnaký zvyšok. Preto ak mu čo len raz vyjde rôzny, vie, že súbory sa naozaj líšia. Takže pravdepodobnosť, že sa pomýli teraz, je  $\approx (0.0000211531)^2 = 0.00000000044745363961$ . Ak Bob celý postup zopakuje dvadsaťkrát, t.j. pošle Bobkovi 40 64-bitových čísel, pravdepodobnosť, že sa Bobek pomýli bude iba  $\approx 3.217 \cdot 10^{-94}$  (to je nula, desatinná čiarka, 93 nul a potom 3217). To je oveľa mensia pravdepodobnosť, ako že sa počítac sám odseba pokazí.

Je ešte veľa iných príkladov toho, ako náhodné čísla môžu pomôcť urobiť lepšie programy, nateraz sa ale usporojíme s tým, že náhodné čísla sú užitočné a chceli by sme ich mať. Ako ich ale získať? Rôzne operačné systémy poskytujú rôzne spôsoby, ako ich získavať, spravidla tak, že sa systém meria časy rôznych udalostí (stláčania kláves, prijatie sietových paketov a pod., v skutočnosti sa v pozadí robia pomerne zložité veci), ktoré sa dajú považovať za náhodné podobne ako hádzanie kockou. Napr. v linuxe existuje súbor </dev/urandom>, do ktorého systém píše náhodné bity. Takže ak programuješ v linuxe, môžeš napísať napr.

```

1 int main() {
2     unsigned int x;
3     ifstream rand("/dev/urandom", ios::binary);
4     for (int i = 0; i < 10; i++) {
5         rand.read((char *)&x, sizeof(x));
6         cout << x % 100 << " ";
7     }
8     cout << endl;
9 }
```

a vždy, keď ten program spustíš, vypíše sa 10 náhodných čísel z rozsahu  $0, \dots, 99$ .

Iná možnosť je skúsiť náhodné čísla vyrobiť priamo nejakým programom. Môže to fungovať? Zjavne nie: hovorili sme, že náhodné veci sú také, pre ktoré nemám dobrý prediktor, ale program, ktorý tie čísla generuje, by bol úplne spoľahlivý prediktor. Na druhej strane, ak si pamätaš, ako poskakovali niektoré body v úlohe 91, tak to vyzeralo pomerne chaoticky. Ak by si nevedel presný bod, ale iba jeho veľkosť (t.j. vzdialenosť od 0), tak predpovedať veľkosť nasledujúceho bodu v Mandelbrotovej iterácii by bolo veľmi ľahké. Tak fungujú tzv. *pseudonáhodné generátory*. Aj keď nevieme náhodu vyrobiť, môžem ju vedieť namnožiť. Pseudonáhodný generátor je program, ktorý dostane krátky náhodný vstup (tzv. *seed*) a z neho vyrobí postupnosť čísel, ktoré súce nie sú náhodné, ale žiadnen program s polynomiálnou zložitosťou ich nevie rozoznať od skutočne náhodných. To znamená, že pseudonáhodné čísla by sme mohli pre všetky rozumné účely použiť rovnako dobre ako náhodné. Či takýto dokonalý pseudonáhodný generátor existuje, zatiaľ nikto nevie, ale existuje viacero takých, ktoré sú dosť dobré v tom zmysle, že nikto zatiaľ nevymyslel, ako ich výstup rozlíšiť od náhodných čísel.

Jedna z možností, ako navrhnutý pseudonáhodný generátor je napr.<sup>4</sup>

```

1 using Int = unsigned long int;
2
3 template <Int a, Int c, Int m>
```

<sup>3</sup>opäť jedna pekná matematická úvaha, ktorú tu teraz nepoviem

<sup>4</sup>Tu som použil inú formu šablóny. Doteraz sme ako parameter šablóny používali typ, ale parametrom môže byť aj číslo. Toto funguje rovnako ako pri typoch – pre každú hodnotu parametra sa vyrobí nový typ, v ktorom sa v čase komplikácie parameter nahradí príslušnou hodnotou.

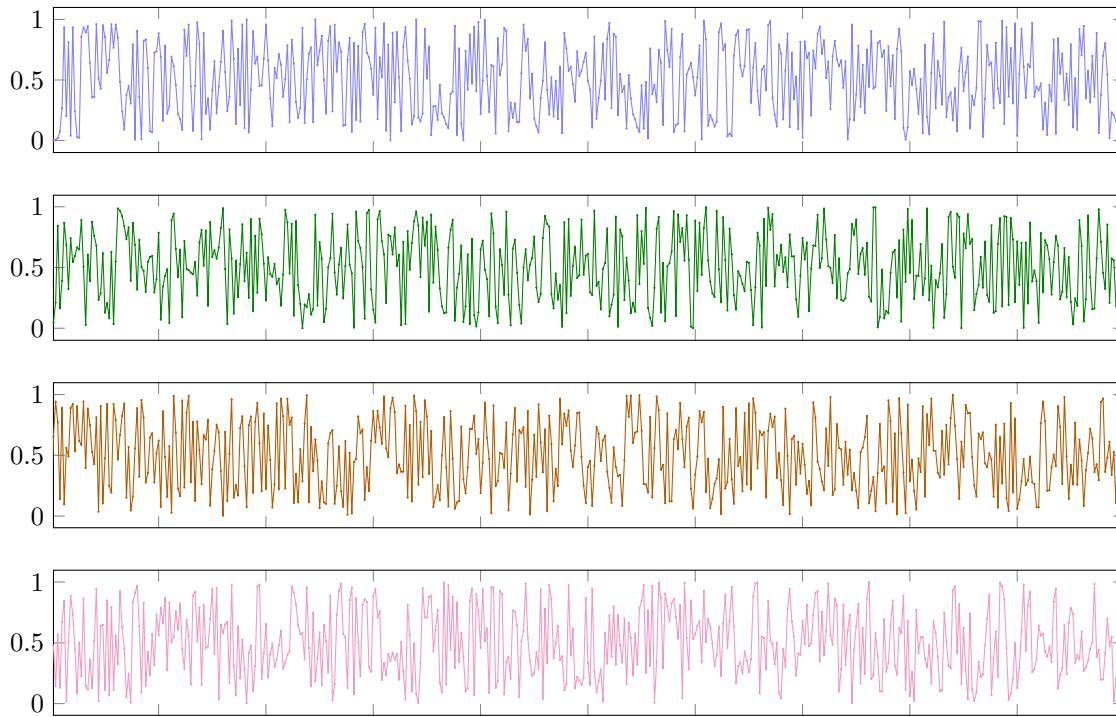
## Náhoda a pravdepodobnosť

```
4 struct Nahoda {
5     Int x;
6
7     Nahoda(Int seed) { x = seed; }
8
9     int operator()() {
10        x = (a * x + c) % m;
11        return x % (1 << 30);
12    }
13}
```

Ked teraz napíšem

```
1 Nahoda<7, 3, 23> r(3);
2 for (int i = 0; i < 20; i++) cout << r() << " ";
3 cout << endl;
```

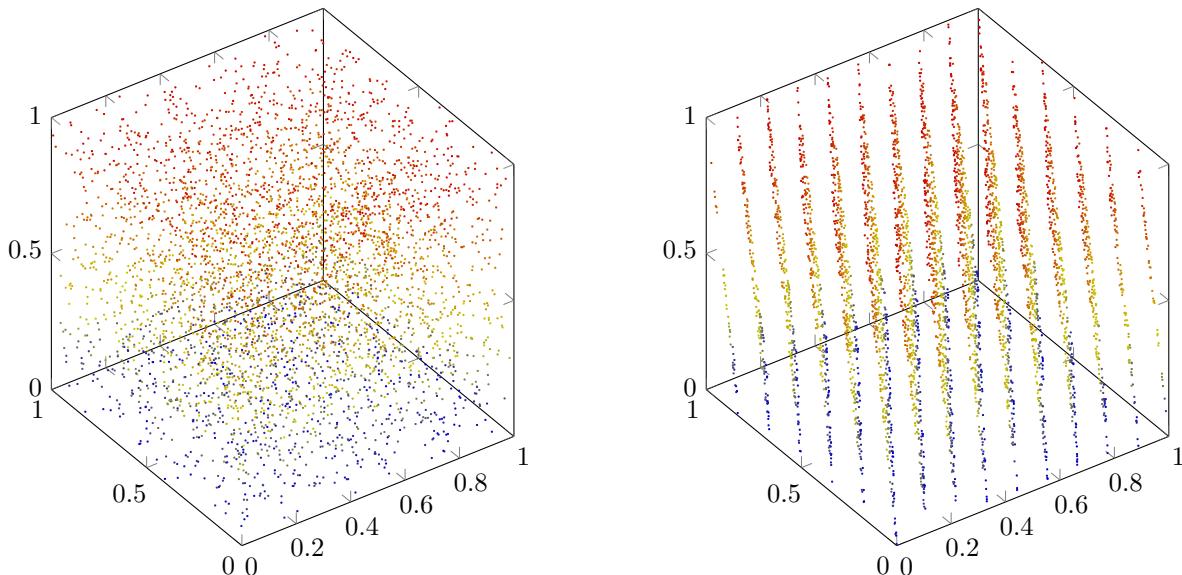
vypíše sa 1 10 4 8 13 2 17 7 6 22 19 21 12 18 14 9 20 5 15 16. To na prvý pohľad vyzerá náhodne, ale ak pokračujem vo vypisovaní a namiesto 20 vypíšem 60 čísel, dostanem 1 10 4 8 13 2 17 7 6 22 19 21 12 18 14 9 20 5 15 16 0 3 1 10 4 8 13 2 17 7 6 22 19 21 21 12 18 14 9 20 5 15 16 0 3 1 10 4 8 13 2 17 7 6 22 19 21 12 18 14 9 – čísla sa po chvíli začnú opakovať. Ked sa nad tým zamyslíš, tak ak mám v tomto generátore parameter  $m$ , tak najneskôr po  $m$  krokoach sa číslo zopakuje a potom sa už bude opakovať stále. Môžeme to skúsiť napraviť tak, že zvolíme veľmi veľké  $m$ , napr.<sup>5</sup> Nahoda<65539, 0, 2147483648> Teraz sa už čísla tak skoro neopakujú. Skúsil som 4 rôzne seedy a výstup som naškáloval do rozsahu 0 . . . 1:



Teraz vyzerá byť všetko v poriadku, ale nie je. Ak si budem z generátora brať trojice čísel (naškálovaných na rozsah 0 . . . 1) a predstavím si ich ako body v 3D priestore, očakával by som náhodné pozície ako na obrázku vľavo. Ale `randu` dá body, ktoré všetky ležia na 15 rovinách ako na obrázku vpravo. Ak si chceš napr. len vytvoriť "náhodné" testovacie vstupy pre tvoj program, tak to asi nijak príliš nevadí, ale ak by si náhodné čísla

<sup>5</sup>Tento pseudonáhodný generátor použila v minulosti IBM na svojich mainframoch pod menom `randu`. Parameter  $a = 2^{16} + 3$ , a  $m = 2^{31}$ .

chcel použiť napr. na bezpečné šifrovanie, tak podobné chyby môžu spôsobiť, že sa tvoj program bude dať hacknúť. Náhoda je náročná vec.



Pri lepšej volbe parametrov môže ale aj takáto jednoduchá procedúra dobre fungovať na generovanie pseudonáhodných čísel. Napr.  $\langle 1103515245, 12345, 2^{31} \rangle$ ,  $\langle 25214903917, 11, 2^{48} \rangle$  alebo  $\langle 6364136223846793005, 1442695040888963407, 2^{64} \rangle$ , sú populárne volby. Samozrejme, existuje aj veľa všeličajkých iných prístupov. Skús sa zamyslieť, ako by si generoval pseudonáhodné čísla po svojom, prípadne ako by si testoval, či čísla, ktoré vidíš, sú náhodné alebo generované nejakým programom.

V C++ sú dva spôsoby, ako môžeš získať (pseudo)náhodné čísla. Prvým, starším, je funkcia `random()`. Je definovaná v knižnici `cstdlib`, ktorú ale `iostream` používa tak či tak. Takže na použitie `random` potrebujete bud `#include<cstdlib>` alebo `#include<iostream>`. Pred použitím treba inicializovať seed volaním `random(seed)` a potom každé volanie `random()` vráti číslo z rozsahu  $0, \dots, 2^{31} - 1$ . Napr.

```

1 #include <iostream>
2 using namespace std;
3 int main() {
4     srand(42);
5     for (int i = 0; i < 10; i++) cout << random() % 100 << endl;
6 }
```

vypíše 10 pseudonáhodných čísel z rozsahu  $0, \dots, 99$ . Pretože tento program má fixný seed, tak vždy, keď ho spustíš, vypíše rovnaké čísla (čo niekedy chceš, napr. pri ladení programu). Otázka ale zníe, ako získať seed, ktorý by bol pri každom spustení rôzny. To sa dá napríklad tak, že za seed dás čas, keď sa program spustil (napr. počet milisekund od nejakého fixného dátumu). Ako to urobiť, si ukážeme v nasledujúcej kapitole.

Druhý, modernejší spôsob, je knižnica `random`. Jej základom je trieda `random_device`. Čo presne sa v nej deje, závisí od konkrétnej implementácie, ale v princípe sa snaží získať "skutočné" náhodné čísla (napr. v Linuxe spravidla číta z `/dev/urandom`). Dá sa použiť podobne, ako `random()`, akurát jej netreba dávať začiatoký seed:

```

1 #include <iostream>
2 #include <random>
3 using namespace std;
4
5 int main() {
6     random_device rd;
7     for (int i = 0; i < 10; i++) cout << rd() << endl;
```

8 }

Problém tohto použitia je, že skutočné náhodné čísla sa môžu minúť a volanie `rd()` zhavaruje. Preto sa `random_device` zvyčajne nepoužíva priamo, ale ako seed do nejakého náhodného generátora. Tých je v knižnici `random` niekoľko, často sa používa tzv. 32-bit Mersenne Twister (*Matsumoto and Nishimura, 1998*) `mt19937`, resp. jeho 64-bit verzia `mt19937_64`. Náhodný generátor dostane seed v konštruktore, napr.

```
1 #include <iostream>
2 #include <random>
3 using namespace std;
4
5 int main() {
6     mt19937 rnd(random_device{}());
7     for (int i = 0; i < 10; i++) cout << rnd() << endl;
8 }
```

Zápis `mt19937 rnd(random_device{}());` je skrátená verzia `random_device rd; mt19937 rnd(rd());` Zátvorky {} sú *brace initializer*, treba zavolať konštruktor `random_device` bez parametrov a potom zavolať jeho `operator()`. Ak už mám generátor vyrobený, môžem ho nastaviť metódou `seed`, napr. `rnd.seed(42)`.

Okrem náhodných generátorov sú v knižnici `random` naprogramované aj rôzne pravdepodobnostné rozdelenia. Tiež sa im tu príliš nebudeme venovať, ale napr. ak chcem dostávať rovnomerne rozdelené náhodné čísla z intervalu  $(0, 1)$ , môžem napísat

```
1 #include <iostream>
2 #include <random>
3 using namespace std;
4
5 int main() {
6     mt19937 rnd(random_device{}());
7     uniform_real_distribution<> dis(0.0, 1.0);
8     for (int i = 0; i < 10; i++) cout << dis(rnd) << endl;
9 }
```

## Meranie času, statické a privátne metódy 30

Na meranie času slúži knižnica `chrono`. Môže pôsobiť trochu komplikované, ale my z nej budeme používať iba niečo. Základom je, že sú rôzne druhy hodiniek (`clock`), napr. `system_clock` alebo `high_resolution_clock`. Každé hodinky majú nejaký začiatočný čas a od neho si pamätajú počet tiknutí (každé hodinky môžu mať rôznu rýchlosť). Zvyšné triedy spracovávajú tiknutia hodiniek tak, aby sa dali použiť na meranie času. Prvou triedou je `duration`, v ktorej sa ukladá počet tiknutí nejakej rýchlosťi. Je to šablóna s dvoma parametrami – prvý je typ, v ktorom sa počet ukladá, a druhý typ, ktorý udáva, kolko sekúnd trvá jedno tiknutie. Na dĺžku tiknutia sa používa typ `ratio`, čo je šablóna reprezentujúca zlomok. Napr. `duration<long, ratio<1, 10>>` uchováva počet tiknutí v type `long` a jedno tiknutie trvá desatinu sekundy. Treba zdôrazniť, že dĺžka tiknutia je parameter šablóny, pre každú dĺžku sa teda vyrobí separátny typ. Pre typické časy sú vyrobené pomocné typy, napr. `milliseconds` je `duration<long, ratio<1, 1000>>`, `hours` je `duration<int, ratio<3600, 1>>` a pod.

Počet tiknutí sa dá nastaviť v konštruktore a vrátiť funkciou `count()`. Premena jednotiek sa dá robiť pomocou šablóny `duration_cast` takto:

```
1 #include <chrono>
2 #include <iostream>
3
4 using namespace std;
5 using namespace chrono;
6 using Dvojsekundy = duration<double, ratio<2, 1>>;
7
8 int main() {
9     milliseconds m(1000);
10    Dvojsekundy ds = duration_cast<Dvojsekundy>(m);
11    cout << m.count() << " " << ds.count() << endl;
12 }
```

Tento program vypíše `1000 0.5`. Premenná `m` ráta v milisekundách a obsahuje 1000 tikov. Premenná `ds` ráta v dvojsekundových intervaloch: 1000ms je 1s a to je 0.5 dvojsekundového intervalu.

Druhou pomocnou triedou je `time_point`. Je to šablóna, ktorej parametrom sú hodinky, a pamäta si počet tiknutí od začiatku hodiniek. Každé hodinky majú funkciu `now()`, ktorá vráti aktuálny `time_point`, napr. `time_point<system_clock> vcul = system_clock::now();`

Tento zápis s dvoma dvojbodkami je nový. V kapitole 24 sme hovorili, že dve dvojbodky znamenajú “*patriaci do*”: vnorený typ, metóda triedy, funkcia v `namespace` a pod. V tomto prípade to vyzerá, ako keby sme volali funkciu `now()` z `namespace system_clock`, ale `system_clock` nie je `namespace`, ale typ. Takže `now()` by mala byť metóda toho typu. Lenže metódu z nejakého typu vieme zavolať iba na premennú toho typu (metóda má prvý neviditeľný parameter `this`), niečo ako `budik.now()`. Takýto zápis označuje volanie tzv. *statickej metódy*. Podme si to pozrieť detailnejšie. Doteraz sme hovorili, že vlastný typ vyrobený pomocou `struct` má svoje premenné. Zoberme napr. takýto typ `Mamut`

```
1 struct Mamut {
2     int ID;
3     float chobot;
4     vector<float> chlp;
5 };
```

v ktorom je o mamutovi uložený jeho identifikátor, dĺžka chobota a dĺžka každého chlpu. Každá premenná typu `Mamut` má vyhradenú pamäť na všetky tri položky. Povedzme, že keď chceme pridať nového mamuta, chceme mu dať nový identifikátor tak, aby všetky mamuty mali rôzne identifikátory. Na to si môžem urobiť globálnu

## Meranie času, statické a privátne metódy

---

premennú `volne`, v ktorej si budem pamätať prvý voľný identifikátor. To ale nie je veľmi pekné, lebo po čase (napr. keď budem chcieť typ `Mamut` použiť v inom programe) ľahko zabudnem, že k typu `Mamut` patrí aj globálna premenná `volne`. A navyše sa mi ľahko môže stať, že si `volne` kdesi v programe omylom niečím prepíšem. Premennú `volne` ale môžem definovať ako statickú premennú typu `Mamut`:

```
1 struct Mamut {
2     int ID;
3     float chobot;
4     vector<float> chlp;
5
6     Mamut() {
7         ID = volne;
8         volne++;
9     }
10    int pocet_chlpov() { return chlp.size(); }
11    int get_ID() { return volne; }
12
13    static int volne;
14 };
15
16 int Mamut::volne = 1;
17
18 int main() {
19     Mamut m;
20     cout << m.ID << endl;
21     cout << Mamut::volne << endl;
22 }
```

a vyrobiť globálnu premennú `Mamut::volne`, ktorá logicky patrí k typu `Mamut`, ale nepatrí žiadnej konkrétnej premennej: každá premenná typu `Mamut` má stále v pamäti iba tri položky `ID`, `chobot` a `chlp`. Zápis `static int volne;` iba hovorí *očakávam, že v programe bude definovaná globálna premenná Mamut::volne* t.j. pred začiatkom programu musíš mať definíciu `int Mamut::volne;` Program by vypísal 1 a 2.

Podobne je to s metódami. Hovorili sme, že každá metóda nejakého typu má neviditeľný prvý parameter `this`. Ak metódu označíš ako `static`, tento parameter mať nebude a bude to obyčajná funkcia, ktorá ale logicky patrí k typu. Napr. môžem označiť `static int get_ID(){...}` a bude to obyčajná funkcia, ktorá vráti `Mamut::volne`, ale `pocet_chlpov()` nemôže byť `static`, lebo v nej používam `this->chlp.size()`.

Stále mi ale ostáva problém, že `Mamut::volne` môžem omylom kdekoľvek prepísat. Ja by som pritom chcel zaručiť, aby sa používala iba v konštruktore typu `Mamut`. Na to slúži privátna časť typu. Ak v definícii typu napíšem `private:`, všetky nasledujúce premenné a metódy budú privátne. Naspať do verejnej časti sa prepнем kľúčovým slovom `public`:

```
1 struct A {
2     int x; // verejné
3     private:
4     int y; // privátne
5     public:
6     int z; // verejné
7     private:
8     int u; // privátne
9 };
```

K privátnym premenným sa dá pristupovať iba z metód príslušnej triedy. V mojom prípade môžem premennú `volne` urobiť v type `Mamut` privátnou:

```

1 struct Mamut {
2     int ID;
3     float chobot;
4     vector<float> chlp;
5
6     Mamut() {
7         ID = volne; // toto je v poriadku, lebo k premennej volne
8         volne++;    // pristupujem z metódy (konštruktora) Mamut
9     }
10
11    int pocet_chlpov() { return chlp.size(); }
12    static int get_ID() { return volne; }
13
14    private:
15        static int volne;
16    };
17
18    int Mamut::volne = 1; // toto je v poriadku,
19                      // inicializácia má výnimku
20
21    int main() {
22        Mamut m;
23        cout << m.ID << endl;
24        cout << Mamut::volne << endl;    // toto je chyba, Mamut::volne
25                                // je privátne v type Mamut
26        cout << Mamut::get_ID() << endl; // toto je v poriadku
27    }

```

Teraz nikde v programe nemôžem k premennej `Mamut::volne` pristupovať, takže mám zaručené, že si ju omylom neprepíšem. Podobne to funguje aj s metódami: privátne metódy sa dajú volať iba z iných metód typu, ktorému patria. Napr.

```

1 struct A {
2     int x;
3
4     private:
5     int y;
6     void hrr(int a) { y += a; }
7
8     public:
9     int frr(int a) { // k privátnym premenným a metódam
10        y = x;           // môžem pristupovať
11        hrr(a);
12        return y;
13    }
14};
15
16 int main() {
17     A a;
18     a.x = 1;             // ok, x je public
19     a.y = 2;             // <- chyba
20     a.hrr();            // <- chyba
21     cout << a.frr(4) << endl; // ok, vypíše 5
22 }

```

Privátne časti typov umožňujú ľahšie udržiavať poriadok vo väčších programoch: napr. typ `set` v C++ má v `public` časti všetky metódy ako `insert()`, `erase()`, `find()`, `size()`, ... a v `private` časti implementáciu

red-black stromu. Keby sa niekedy autori rozhodli vymeniť red-black strom za AA-strom, môžu to urobiť bez strachu, že nejaké programy prestanú fungovať. Len pripomienim, čo by malo byť jasné: privátne a statické premenné/metódy sú rôzne veci: statický znamená, že je to normálna globálna premenná alebo funkcia, ale logicky patrí k danému typu (a tým pádom môže byť privátnej). Privátne premenné/metódy sú také, ku ktorým sa dá pristupovať len z metód typu, ktorému patria. Na záver tejto vsuvky posledná poznámka: okrem kľúčového slova `struct` sa na definovanie nových typov dá použiť aj `class`. Robia presne to isté, iba `struct` má default časť `public` a `class` má default `private`.

Podme naspäť k meraniu času. Skončili sme pri tom, že každý typ hodiniek má statickú metódu `now()`, ktorá vráti aktuálny čas ako `time_point`, napr.

```
1 time_point<system_clock> vcul = system_clock::now();
```

Ešte treba vedieť, že `time_point` má metódu `time_since_epoch()`, ktorá vráti `duration` od začiatku príslušných hodiniek a že dve premenné typu `time_point` môžem odčítať a dostať som príslušný `duration`.

**Úloha 114.** Naprogramuj funkciu `template<typename F> int meraj(F f){...}`, ktorá má ako parameter funkciu /lambdu bez parametrov a odmeria, kolko milisekund trvá jej spustenie.

S riešením predchádzajúcej úlohy môžeš skúsiť pozorovať, ako je kompilátor schopný veci optimalizovať. Zober si napr. program

```
1 int main() {
2     cout << meraj([]() {
3         int x;
4         for (int i = 0; i < 1000000000; i++) x++;
5     }) << endl;
6 }
```

A skompiluj ho raz `g++ test.cc -o test` a raz s prepínačom `-O3` napr. `g++ -O3 test.cc -o test_opt`. Pri optimalizácii kompilátor pochopil, že v tom cykle nič zaujímať nerobiš a dá sa to zrátať priamo.

**Úloha 115.** S pomocou predchádzajúcej úlohy napiš program, ktorý odmeria rýchlosť triedenia takto: pre každé  $n$  z rozsahu 10000, 20000, 30000, ..., 1000000 sa  $10 \times$  vygeneruje náhodný vektor s  $n$  prvками, odmeria sa čas, ktorý treba na jeho utriedenie funkicou `sort` z knižnice `algorithm` a vypíše sa priemer z časov v milisekundách.

## Projekt: mapy náhodných ostrovov

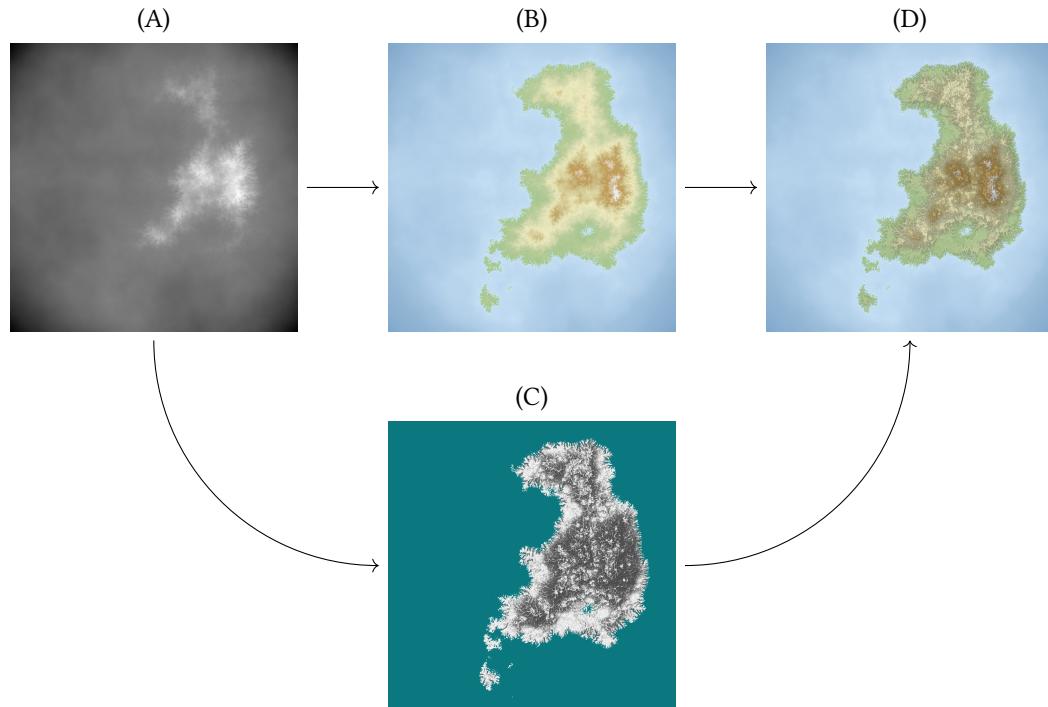
31

Náhodné čísla majú veľa rôznych použití. Môžu pomôcť niečo efektívnejšie naprogramovať, ako sme videli pri porovnávaní súborov. Môžu sa s nimi vyrábať testovacie vstupy pri ladení programov. A dajú sa dobre využiť aj pri generovaní rôznych vecí, ktoré chceme, aby zakaždým vyzerali inak. Tu sa hodí, že máme pseudonádondé čísla: na rozdiel od náhodných totiž môžeme použiť seed na to, aby sme “náhodné” veci presne zopakovali, ak treba. Určite poznáš rôzne hry, ktoré vedia generovať náhodné svety. Predpokladajme, že dej hry sa odohráva na hornatom ostrove a chceme, aby sa zakaždým vygeneroval iný. V tomto projekte chceme vygenerovať náhodný ostrov a vykresliť ho v štýle starých tieňovaných máp nejak takto:



Úlohu si rozdelíme na menšie časti a o každej z nich si niečo povieme. Najprv si vyrobíme výškovú mapu (*heightmap*): tabuľku `double` s číslami z rozsahu  $0 \dots 1$ , kde  $0$  znamená najmenšiu a  $1$  najväčšiu výšku (obrázok A). Z nej potom urobíme dva obrázky: jeden, ktorý bude určovať farbu získame tak, že každému pixelu priradíme farbu z nejakého gradientu podľa jeho výšky (obrázok B) a druhý, ktorý bude určovať tieňe, vyrobíme podľa sklonu terénu (obrázok C). Napokon oba obrázky spojíme do výsledku (obrázok D).

## Projekt: mapy náhodných ostrovov



Začneme s prípravnou prácou: na zapisovanie finálneho výsledku môžeme použiť náš starý súbor `obrazok.h`<sup>1</sup>. Okrem toho budeme veľa robiť s tabuľkami, kde budeme mať buď reálne čísla alebo farby, tak je dobré sa na to pripraviť.

**Úloha 116.** Uprav triedu `Tabulka` z kapitoly 22 tak, aby sa v nej dali ukladať akokoľvek typy (pomocou šablóny).

Prvý pokus o náhodnú výškovú mapu môže vyzerať nejak takto<sup>2</sup>

```

1 #include <algorithm>
2 #include <fstream>
3 #include <iostream>
4 #include <random>
5 #include <sstream>
6 #include <vector>
7
8 #include "obrazok.h"
9 #include "tabulka.h"
10
11 using namespace std;
12 mt19937 rnd(random_device{}());
13 uniform_real_distribution<> dis(0.0, 1.0);
14
15 void zapis(Tabulka<double>& T, const char* fname) {
16     vector<unsigned char> data(T.m * T.n * 4);
17     for (int i = 0; i < T.m; i++) {
18         for (int j = 0; j < T.n; j++) {
19             int offs = 4 * ((T.n - j - 1) * T.n + i);
20             for (int k = 0; k < 3; k++)

```

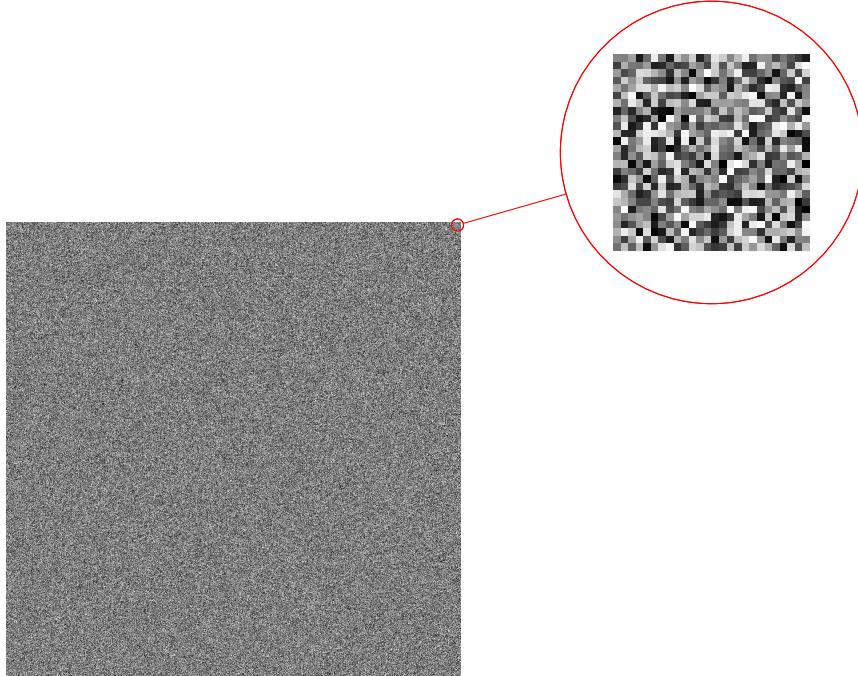
<sup>2</sup>Tu idem použiť zápis, ktorý sme doteraz nemali a zaslúži si komentár. Ak mám napr. premennú `int i`, tak vieš, že `i++` je príkaz, ktorý priráta k `i` jednotku. Doteraz som ti nepovedal, že `i++` sa dá použiť aj ako výraz. Keby som mal napr. `vector<int> a`, tak `int x = a[i++]` znamená *Do x ulož hodnotu a[i] a potom zvýš i o jednotku*. Je to teda to isté, ako keby som napísal `x = a[i]; i = i+1`. Niekoľko si možno zbadal aj zápis `++i`, ktorý sám osebe robí to isté, t.j. pripočítava k `i` jednotku, ale urobí to na začiatku. Preto `int y = a[++i]` je to isté ako `i = i+1; y = a[i]`;

```

21     // min a max sú z knižnice <algorithm> a vrátia menšie a väčšie z dvojice čísel
22     data[offs++] = min(255 * max(T(i, j), 0.0), 255.0);
23     data[offs++] = 255;
24 }
25 zapis_rgba_png(T.n, T.m, data.data(), fname);
26 }
27
28 int main(int argc, char** argv) {
29     if (argc > 1) {
30         unsigned long seed;
31         stringstream ss(argv[1]);
32         ss >> seed;
33         rnd.seed(seed);
34     }
35     int n = 1024;
36     Tabulka<double> T(n, n);
37     for (int i = 0; i < n; i++)
38         for (int j = 0; j < n; j++) T(i, j) = dis(rnd);
39
40     zapis(T, "sum.png");
41 }
```

V tomto programe som si spravil pomocnú funkciu `zapis()`, ktorá zoberie tabuľku s výškovou mapou a zapíše z nej obrázok do súboru tak, že nula je čierna a jednotka je biela. Pri zapisovaní obrázka potrebujem na jeden pixel štyri byty `r, g, b, a`. Keďže chcem vyrobiť odtiene sivej, hodnoty `r, g, b` nastavím všetky rovnako, a to na hodnotu `255 * T(i, j)`.

V hlavnom si vyróbím náhodný generátor a pozriem sa, či nie je na príkazovom riadku parameter. Ak je, zoberiem ho ako seed. Ak budem chcieť zopakovať ten istý ostrov, stačí zadať rovnaký seed. Spustím program a výsledok je

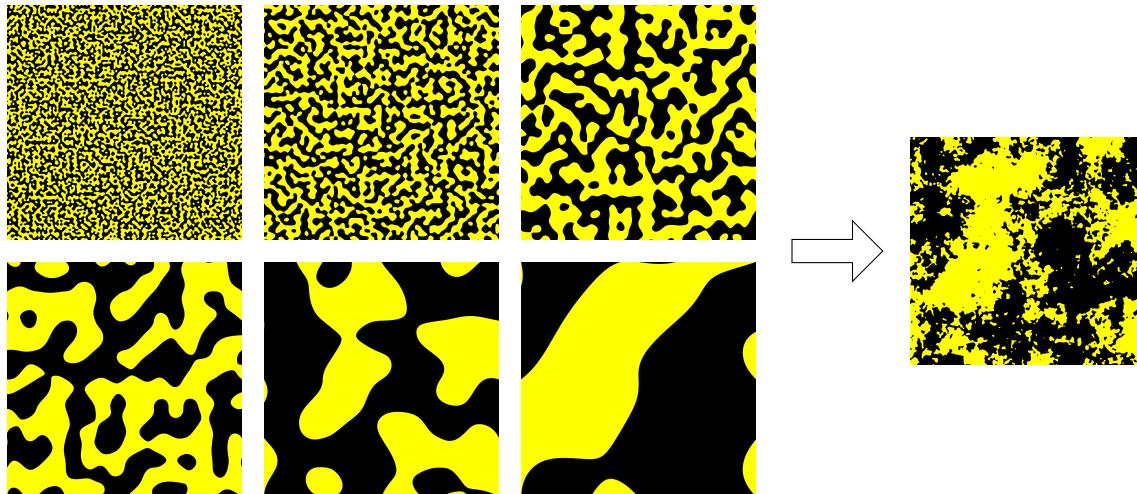


Ugh. Zväčšený roh vpravo ukazuje, prečo to nefunguje: keďže každý pixel je nezávislé náhodné číslo, nie je medzi nimi žiadnen vzťah. Hodnoty medzi jednotlivými pixelmi skáču tak, že výsledkom, keď sa naňho pozerám trochu z diaľky, je sivá plocha. Potrebujem niečo, čo by síce bolo náhodné, ale zasa nie až tak úplne. Ukážem ti

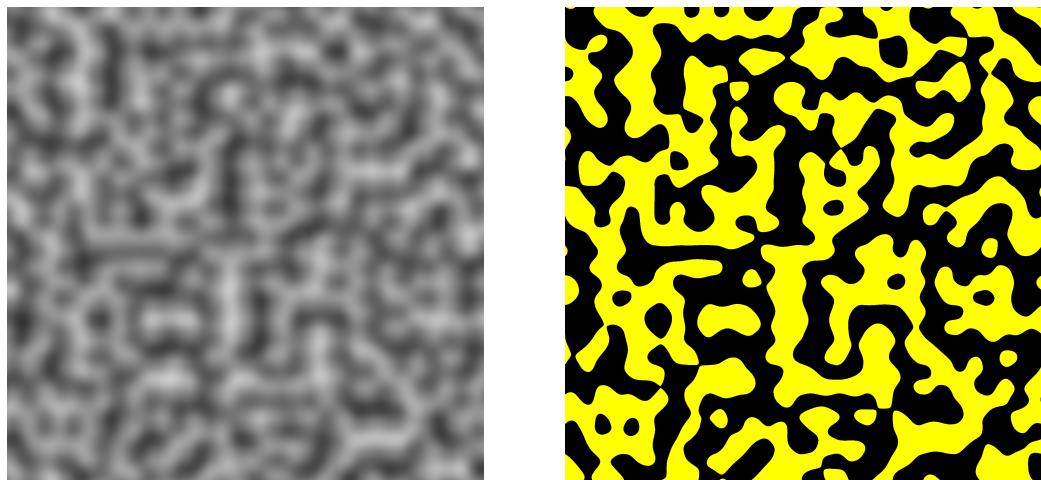
## Projekt: mapy náhodných ostrovov

---

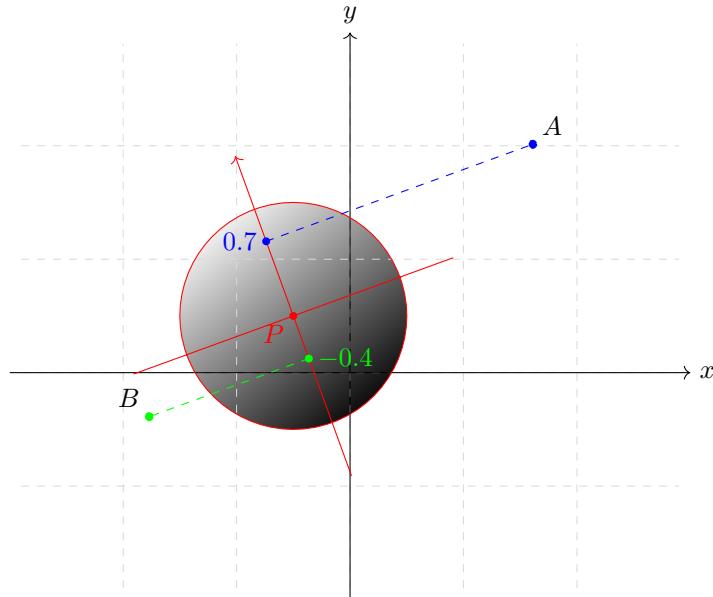
jednu z možností, ako náhodnosť regulovať. Volá sa podľa chlapíka, čo s tým prvý prišiel, *Perlin noise*. S ním sa dajú generovať rôzne veľké náhodné vzory, ktorých kombináciou vznikajú fraktálovité náhodné útvary.



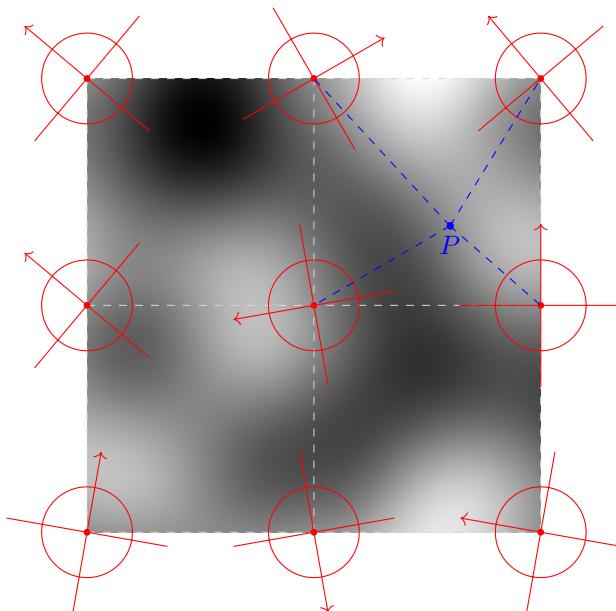
Perlin noise najprv popíšeme matematicky a potom sa pozrieme na to, ako ho naprogramovať. Zoberme si rovinu. Každému bodu chceme priradiť číslo tak, že čísla menšie ako  $-1$  znamenajú čiernu, čísla väčšie ako  $1$  bielu a medzi nimi je spojitý prechod. Budeme tak dostávať obrázky ako ten nasledujúci ako vľavo. Keby sme chceli vzorku ako tá vpravo, môžeme každý bod zaokrúhliť: ak je kladný, pixel bude žltý, ak je záporný, pixel bude čierny.



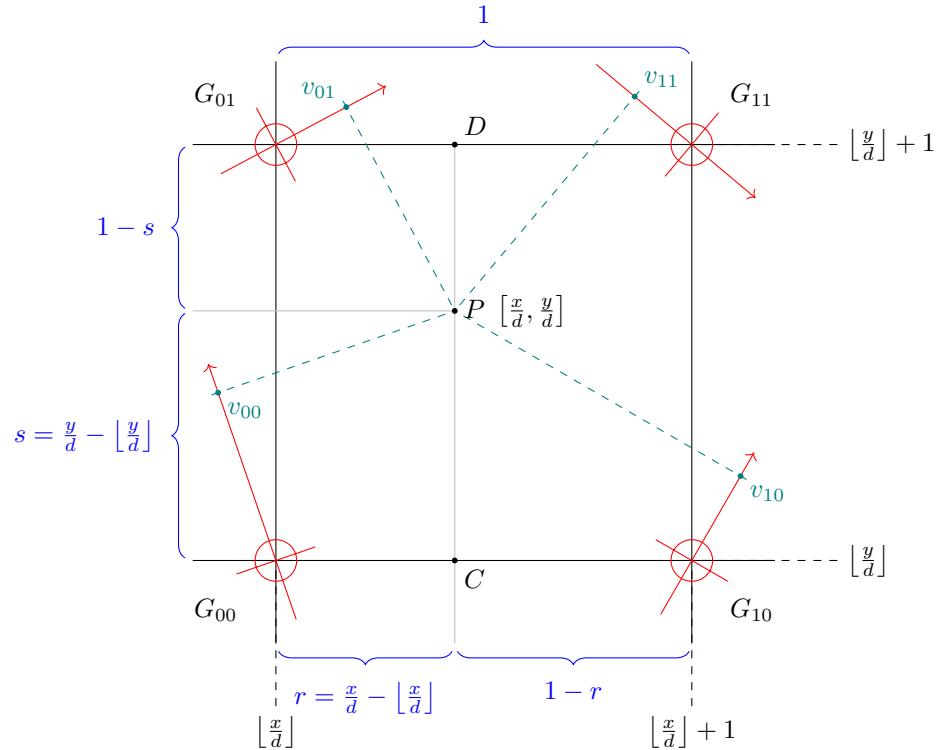
Základom pre vyrobenie obrázka vľavo je "natočený gradient". Zoberiem si nejaký bod  $P$  a umiestním doňho natočenú šípku. Tento gradient každému bodu v rovine priradí hodnotu podľa jeho pozicie v smere šípky. Napr. ak mám šípku umiestnenú v bode  $P = [-0.5, 0.5]$ , tak bodu  $A$  sa priradí číslo  $0.7$  a bodu  $B$  číslo  $-0.4$ .



Výsledný obrázok získame tak, že si v rovine spravíme pravidelnú mriežku, v ktorej umiestnime náhodne otočené gradienty. Každý bod roviny leží v nejakom štvorci mriežky a jeho výsledná farba sa určí kombináciou farieb zo štyroch susedných gradientov.

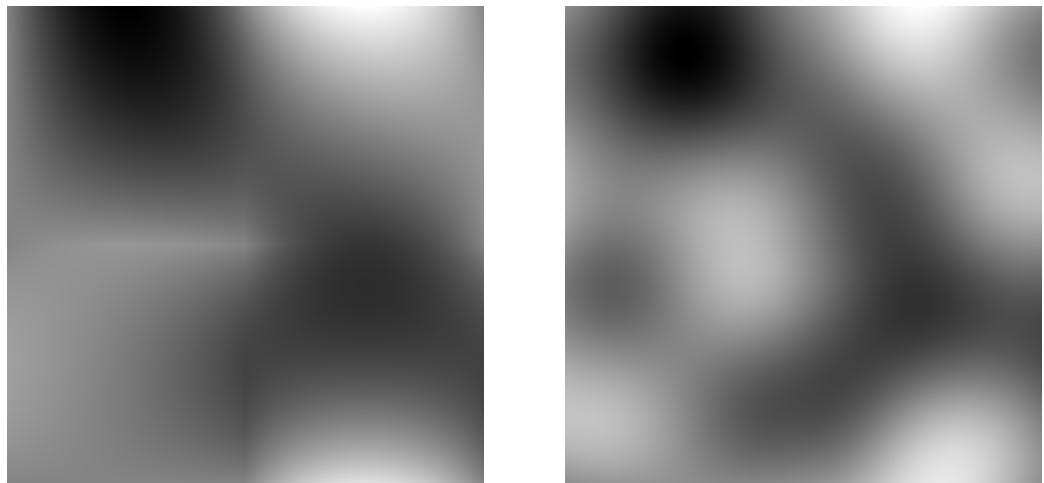


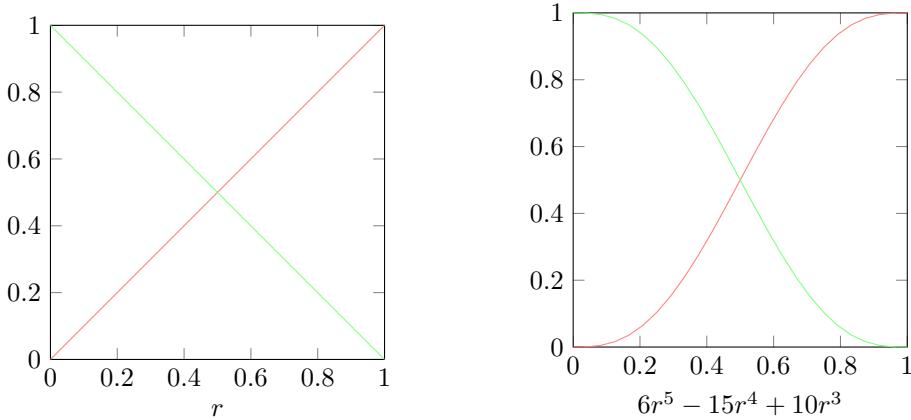
Kombináciu hodnôt urobíme podobne, ako sme robili interpoláciu farieb vo farebnom gradiente v projekte o Mandelbrotovej množine, až na to, že teraz potrebujeme interpolovať v dvoch rozmeroch (preto sa tomu hovorí *bilineárna interpolácia*). Predstav si, že máme mriežku gradientov s krokom  $d$  a chceme určiť farbu pre bod  $P$  so súradnicami  $[x, y]$ . Keď si všetky súradnice vydelíme  $d$ kom, mriežka bude mať krok 1.  $P$  sa nachádza v mriežke medzi štyrmi bodmi  $G_{00}$ ,  $G_{10}$ ,  $G_{01}$  a  $G_{11}$ . Pri pomeníme, že  $\lfloor x \rfloor$  označuje celú časť z  $x$ , preto  $G_{00}$  má súradnice  $\lfloor \frac{x}{d} \rfloor, \lfloor \frac{y}{d} \rfloor$  a ostatné 4 body sú vždy o 1 ďalej.



Predpokladajme, že sme už vyrátali hodnoty, ktoré prislúchajú bodu  $P$  v jednotlivých gradientoch a označili sme si ich  $v_{00}$ ,  $v_{10}$ ,  $v_{01}$  a  $v_{11}$ . Ako z nich skombinovať výslednú hodnotu v  $P$ ? Desatinná časť po vydenení  $x/d$  je  $r = \frac{x}{d} - \lfloor \frac{x}{d} \rfloor$ . Keby si bod  $P$  posúval v štvorci zľava doprava, hodnota  $r$  sa bude plynule meniť medzi 0 a 1. Preto hodnotu v bode  $C$  vieme interpolovať ako  $v_C = v_{00}(1 - r) + v_{10}r$ . Podobne hodnotu v bode  $D$  interpolujeme  $v_D = v_{01}(1 - r) + v_{11}r$ . Napokon urobíme rovnakým spôsobom interpoláciu medzi  $v_C$  a  $v_D$  v smere osi  $y$ , takže výsledok bude  $v_C(1 - s) + v_Ds$ .

Skoro dobré, ale je tam stále drobný problém: pretože interpolujeme lineárne iba v rámci jedného štvorca, vznikajú nám nepekné zlomy ako vľavo, kým my potrebujeme hladký prechod ako vpravo





Hladký prechod sa dá dosiahnuť tak, že namiesto interpolácie s parametrami  $r$  a  $1 - r$ , ktoré sú pri koncoch špicaté, budeme interpolovať s parametrami  $f(r)$  a  $1 - f(r)$  pre nejakú vhodnú funkciu  $f$ , ktorá je na koncoch hladká. Perlin odporúča zobrať  $f(r) = 6r^5 - 15r^4 + 10r^3$ . Môžeme aj tak, prečo nie.

Teraz môžeme napísť takmer celý program na Perlin noise. Najprv si urobíme pomocnú funkciu na lineárnu interpoláciu `lerp`, ktorá sa bude hodíť vo všeobecnejšej podobe

```

1 template <typename T>
2 T lerp(const T& v0, const T& v1, double t) {
3     return (1 - t) * v0 + t * v1;
4 }
```

Teraz vyrobíme triedu `Perlin`, ktorá bude mať dva parametre: `n` veľkosť obrázka (pre jednoduchosť budem robiť štvorcové obrázky rozmerov  $n \times n$ ) a krok mriežky `d`. Zavolenie konštruktora vyrobí tabuľku `H`, v ktorej bude výsledná výšková mapa. Na natočené šípky budem mať triedu `Vec` o ktorej si poviem viac o chvíli, nateraz si ju predstav ako

```

1 struct Vec {
2     double x, y;
3 };
```

Natočené šípky budú uložené v tabuľke  $G$  rozmerov  $m \times m$ , kde  $m = 1 + \lfloor \frac{n}{d} \rfloor$ . V konštruktore najprv vygenerujeme náhodné šípky tak, že zvolíme náhodný uhol `theta` ( $\theta$ ) a šípka bude ukazovať do bodu so súradnicami  $[\cos(\theta), \sin(\theta)]$  (o funkciach sin a cos pozri odbočku v kapitole 17). Potom už len vyrátam pre každý bod  $P = [i, j]$  jeho farbu. To znamená, že keď vyrobíme premennú typu `Perlin`, konštruktor vyrobí tabuľku `H`, ku ktorej potom môžem pristupovať.

```

1 struct Perlin {
2     int n, d, m;
3     Tabulka<double> H;
4     Tabulka<Vec> G;
5
6     // konštruktor.
7     // všimni si zápis n{_n} - niekedy je čitateľnejšie volať copy konštruktor
8     // pre int za dvojbodkou ako písanie int _d
9     Perlin(int _n, int _d) : n{_n}, d{_d}, m{1 + n / d},
10           H(n, n), G(m, m) {
11         for (int i = 0; i < m; i++)
12             for (int j = 0; j < m; j++) {
13                 double theta = dis(rnd) * M_PI * 2.0;
14                 G(i, j) = {cos(theta), sin(theta)};
15             }
16     }
```

```

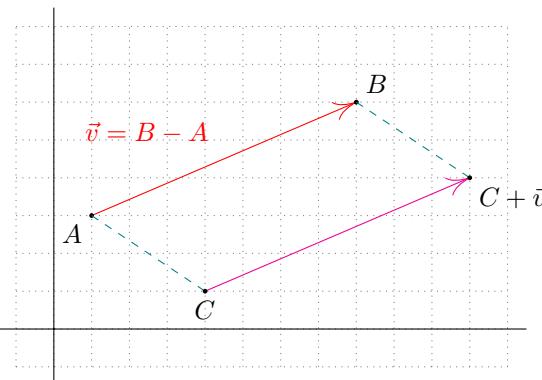
17   for (int i = 0; i < n; i++) {
18     for (int j = 0; j < n; j++) {
19       // bod P so súradnicami [i,j]
20       Vec p{ (double)i / (double)d, (double)j / (double)d};
21
22       // súradnice bodu G00
23       int o[2] = {(int)p.x, (int)p.y};
24
25       // hodnoty v00, v10, v01 a v11
26       double v[4];
27
28       ... // tu ich treba nejak vyrátať
29
30       double r = fade(p.x - o[0]), s = fade(p.y - o[1]);
31       double vc = lerp(v[0], v[1], r), vd = lerp(v[2], v[3], r);
32       H(i, j) = lerp(vc, vd, s);
33     }
34   }
35
36   // fade funkcia, ako ju odporúčal Perlin
37   double fade(double t) {
38     return t * t * t * t * (10 + t * (6 * t - 15));
39   }
40 };

```

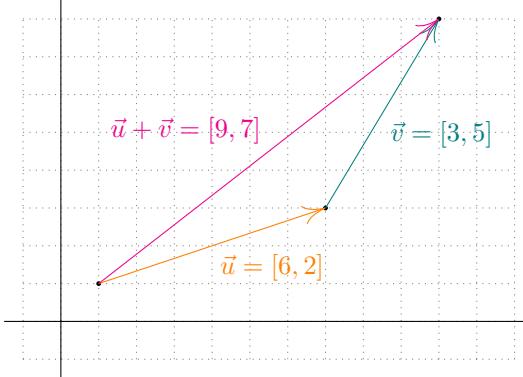
Ostáva doplniť posledná vec, a to ako vyrátať hodnoty  $v_{00}, \dots, v_{11}$ , teda ako pre daný bod zrátať jeho výšku v nejakom natočenom gradiente. Predtým sa ale hodí

### Matematické intermezzo: o bodoch a vektoroch

Bod v rovine je vec, ktorá má dve súradnice,  $x$  a  $y$ . Ak mám dva body, napr.  $A[x, y]$ ,  $B[x', y']$ , tak najkratšia cesta z bodu  $A$  do bodu  $B$  je úsečka, ktorá sa v smere osi  $x$  pohne o  $x' - x$  a v smere osi  $y$  o  $y' - y$ . Takýto úsečka sa volá prenášač (vektor) a dá sa zapísat dvoma číslami pre posun v  $x$ -ovom a  $y$ -ovom smere, takže vyzerá rovnako ako bod so súradnicami  $[x' - x, y' - y]$ . Je preto prirodzené si to predstaviť tak, že ak odčítam od seba dva body, dostanem vektor. Ak treba body a vektor rozlíšiť, nad vektory sa niekedy zvykne písať šípkou, napr. vektor  $v$  je  $\vec{v} = B - A$ . Podobne keď k nejakému bodu  $C$  prirátame vektor  $\vec{v}$ , dostaneme nový bod (ten, do ktorého nás vektor prenesie).



Vektoru môžeme sčítovať prirodzeným spôsobom po zložkách (t.j.  $[x, y] + [x', y'] = [x + x', y + y']$ ), čo sa dá nakresliť ako skladanie šípok: na konci jednej nakreslíme druhú:



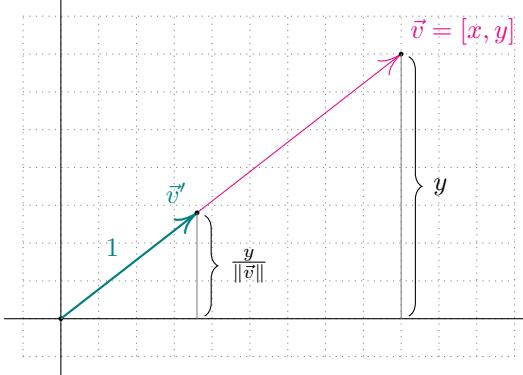
Rovnako dobre dáva zmysel, aby sme vektor vynásobili číslom: napr. ak  $\vec{v} = [x, y]$ , tak  $4.2 \cdot \vec{v} = [4.2x, 4.2y]$  je šípka v rovnakom smere ako  $\vec{v}$ , iba 4.2-krát dlhšia. Podobne  $-0.5 \cdot \vec{v}$  je šípka polovičnej dĺžky a v opačnom smere.

Ak mám bod  $A[x, y]$ , tak si viem povedať, že to je vlastne vektor  $A - [0, 0]$ , t.j. šípka, ktorá ukazuje zo začiatku súradnicovej sústavy do  $A$ . Teraz je bod a vektor naozaj to isté. Premysli si to: napr.  $[9, 7] - [3, 5] = [6, 2]$  môžem interpretovať tak, že mám bod so súradnicami  $[9, 7]$  a bod so súradnicami  $[3, 5]$ , tak vektor medzi nimi má súradnice  $[6, 2]$ . Ale rovnako môžem povedať, že ak mám vektor  $[3, 5]$ , tak  $-[3, 5]$  je rovnako dlhý vektor v opačnom smere a  $[9, 7] + (-[3, 5])$  je skladanie vektorov, ktorého výsledkom je vektor  $[6, 2]$ .

Z Pythagorovej vety viem, že ak  $\vec{v}$  je vektor so súradnicami  $[x, y]$ , tak jeho dĺžka je  $\|\vec{v}\| = \sqrt{x^2 + y^2}$ . Ak obe súradnice vydelím dĺžkou, dostanem vektor  $\vec{v}' = \left[ \frac{x}{\|\vec{v}\|}, \frac{y}{\|\vec{v}\|} \right]$ , ktorého dĺžka je

$$\|\vec{v}'\| = \sqrt{\left( \frac{x}{\|\vec{v}\|} \right)^2 + \left( \frac{y}{\|\vec{v}\|} \right)^2} = \sqrt{\frac{x^2}{\|\vec{v}\|^2} + \frac{y^2}{\|\vec{v}\|^2}} = \sqrt{\frac{1}{\|\vec{v}\|^2} (x^2 + y^2)} = \sqrt{\frac{1}{\|\vec{v}\|^2} \sqrt{x^2 + y^2}} = \frac{1}{\|\vec{v}\|} \|\vec{v}\| = 1$$

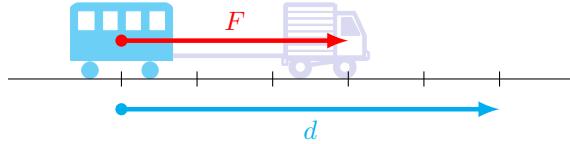
Hovoríme, že  $\vec{v}'$  je *normovaný* (resp. *normalizovaný*) vektor.



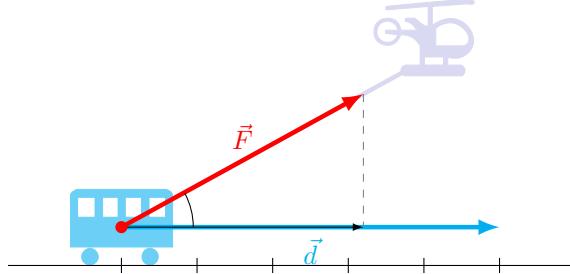
Máme teda vektory, ktoré vieme sčítovať, odčítovať a násobiť reálnym číslom. Samozrejme, toto všetko by rovnako dobre fungovalo v troch (a viacerých) rozmeroch. Môžeme ale nejak rozumne vynásobiť dva vektory medzi sebou? Tu to už nie je také jednoznačné. Môžeme vymyslieť rôzne operácie, ktoré nazveme súčin, a ktoré budú mať rôzne vlastnosti podľa toho, čo práve potrebujeme počítať. Jeden spôsob je uvedomiť si, že vektor v rovine so súradnicami  $[x, y]$  je vlastne komplexné číslo  $x + iy$ , takže môžeme používať násobenie ako pri komplexných číslach. Ukážem ti ešte dva iné spôsoby, ktoré sa nám budú hodíť, a to tzv. *skalárny* a *vektorový* súčin.

Skalárny súčin je spôsob násobenia, ktorý je veľmi prirozený napr. vo fyzike. Možno vieš, že fyzikálna veličina *práca* je súčin sily a dráhy, t.j. ak pôsobí na telo sila  $1\text{N}$  a posunie ho tým o  $1\text{m}$ , vykoná pri tom prácu  $1\text{J}$ . Zvyčajne sa to zapisuje  $W = Fd$ .

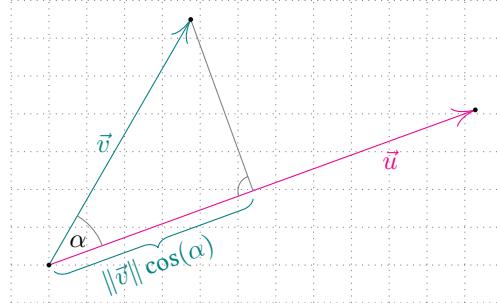
## Projekt: mapy náhodných ostrovov



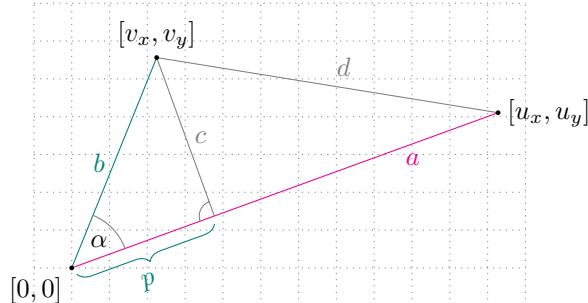
Tu sa ale potichu predpokladá, že sila aj dráha sú čísla, a preto ich môžem násobiť. Lenže v skutočnosti sila aj dráha sú vektorov a môžu pôsobiť rôzny smereom



V tomto prípade sa časť sily, ktorou pôsobí vrtuľník, vyplytvá na nadlahčovanie vagóna a prácu robí iba tá časť, ktorá je rovnobežná s dráhou, takže na vyrávanie práce násobím dĺžku vektora  $\vec{d}$  a priemetu  $\vec{F}$  (čierna šípka na obrázku). Aby sa zachoval rovnaký zápis, chcel by som násobiť vektorov tak, aby som mohol napísat  $W = \vec{F} \cdot \vec{d}$ . Ako sa také násobenie dá urobiť? Výsledkom násobenia dvoch vektorov musí byť číslo, ktoré vznikne vynásobením dĺžky jedného a priemetu druhého:



Výsledok by teda mal byť  $\vec{u} \cdot \vec{v} = \|\vec{u}\| \cdot \|\vec{v}\| \cdot \cos(\alpha)$ , kde  $\alpha$  označuje uhol, ktorý tie dva vektorov zvierajú. Teraz idem trochu počítať a na záver mi vyjde, že takýto skalárny súčin sa dá veľmi jednoducho vyrátať. Tak podme na to. Zoberme si vektorov  $\vec{u}, \vec{v}$  ako na predchádzajúcom obrázku a označme si  $a = \|\vec{u}\|$ ,  $b = \|\vec{v}\|$ ,  $p = b \cos(\alpha)$ . Situácia teda vyzerá takto:



Potrebuje vyrátať skalárny súčin  $\|\vec{u}\| \cdot \|\vec{v}\| \cdot \cos(\alpha)$ , t.j. v našom označení  $ap$ . Všimneme si dva pravouhlé trojuholníky: jeden so stranami  $b, p, c$  a druhý so stranami  $d, a - p, c$ . Z Pythagorovej vety dostaneme pre prvý z nich  $p^2 = b^2 - c^2$  a pre druhý  $c^2 = d^2 - (a - p)^2$ . Keď to dáme dokopy, dostaneme  $p^2 = b^2 - d^2 + (a - p)^2$ . Teraz roznásobíme  $(a - p)^2 = (a - p) \cdot (a - p) = a^2 - 2ap + p^2$ , takže máme

$$p^2 = b^2 - d^2 + a^2 - 2ap + p^2$$

a teda

$$2ap = b^2 + a^2 - d^2$$

Teraz si dosadím  $b^2 = \|\vec{v}\|^2 = v_x^2 + v_y^2$ ,  $a^2 = \|\vec{u}\|^2 = u_x^2 + u_y^2$  a  $d^2 = (u_x - v_x)^2 + (u_y - v_y)^2$  a dostanem

$$2ap = (v_x^2 + v_y^2) + (u_x^2 + u_y^2) - (u_x - v_x)^2 - (u_y - v_y)^2$$

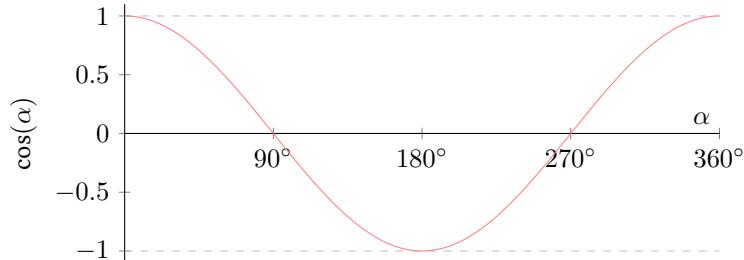
Ked' si roznásobím  $(u_x - v_x)^2 = u_x^2 - 2u_xv_x + v_x^2$  a  $(u_y - v_y)^2 = u_y^2 - 2u_yv_y + v_y^2$  a dosadím, dostanem

$$ap = u_xu_y + v_xv_y$$

Takže si to zhrňme: ak mám dva vektory  $\vec{u} = [u_x, u_y]$  a  $\vec{v} = [v_x, v_y]$ , tak

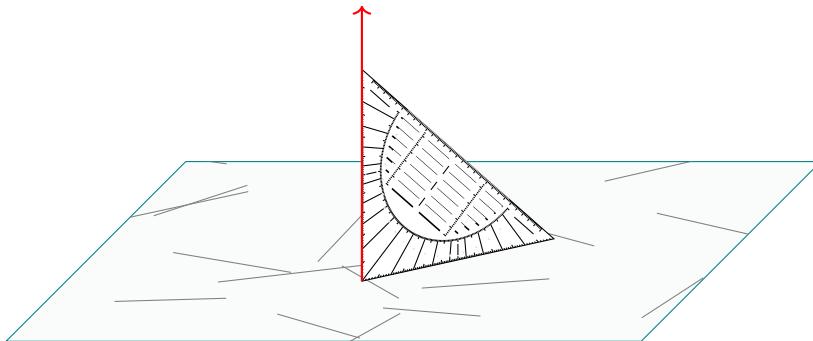
$$\vec{u} \cdot \vec{v} = \|\vec{u}\| \cdot \|\vec{v}\| \cdot \cos(\alpha) = u_xu_y + v_xv_y$$

Toto sa často hodí, lebo  $u_xu_y + v_xv_y$  sa dá zrátať ľahko a rýchlo a ak sú  $\vec{u}$  a  $\vec{v}$  normované, čiže ich dĺžka je 1, tak priamo vyrátam kosínus uhla medzi nimi. Ked' si nakreslím, ako závisí kosínus od velkosti uhla, dostanem známy obrázok



Špeciálne, viem rýchlo zistiť, či sú dva vektory kolmé: stačí overiť, či ich skalárny súčin je rovný nule. A, samozrejme, sice sme to všetko rátali v rovine, ale pre trojrozmerné vektory by to fungovalo úplne rovnako.

Iný spôsob súčinu, ktorý sa ale dá použiť iba pre trojrozmerné vektory, je tzv. *vektorový súčin*. Ak máš rovinu v 3D priestore (napr. list papiera na stole) a v nej nakreslené hocjaké úsečky, tak existuje jeden smer, ktorý je kolmý na každú z nich:



Takýto smer sa volá *normála* danej roviny. Pri vektorovom súčine, ako už názov napovedá, budeme počítať vektor. Predpokladajme, že máme dva (3D) vektory  $\vec{u} = [u_x, u_y, u_z]$  a  $\vec{v} = [v_x, v_y, v_z]$ . Ich vynásobením dostaneme výsledný (3D) vektor  $\vec{w} = \vec{u} \times \vec{v}$ . Budeme chcieť, aby výsledok  $\vec{w}$  mal smer normály na rovinu, v ktorej ležia  $\vec{u}$  a  $\vec{v}$ . Okrem smeru si treba povedať, či má výsledok ísť "hore", alebo "dolu". Budeme chcieť, aby platilo *pravidlo pravej ruky*: ak prstami pravej ruky ukážeš na  $\vec{u}$  a vystretným palcom na  $\vec{v}$ , tak vrch dlane<sup>3</sup> smeruje v smere  $\vec{u} \times \vec{v}$ :

<sup>3</sup>Všimni si, že tu na rozdiel od násobenia čísel alebo skalárneho súčinu záleží na poradí:  $\vec{u} \times \vec{v} = -(\vec{v} \times \vec{u})$ .

## Projekt: mapy náhodných ostrovov



Pri označovaní osí sa zvyčajne používa tzv. pravoručý systém, takže ak si nazvem  $\vec{x} = [1, 0, 0]$ ,  $\vec{y} = [0, 1, 0]$  a  $\vec{z} = [0, 0, 1]$ , tak chcem, aby  $\vec{x} \times \vec{y} = \vec{z}$  a  $\vec{y} \times \vec{x} = -\vec{z}$ . Ak mám vektor  $\vec{u} = [u_x, u_y, u_z]$ , tak  $\vec{u} = u_x \vec{x} + u_y \vec{y} + u_z \vec{z}$  a podobne  $\vec{v} = [v_x, v_y, v_z] = v_x \vec{x} + v_y \vec{y} + v_z \vec{z}$ . V z-ovej súradnici výsledku by preto mohlo byť  $u_x v_y$  s kladným znamienkom a  $u_y v_x$  so záporným. A rovnako v ostatných dimenziách. To nás nabáda k definícii

$$[u_x, u_y, u_z] \times [v_x, v_y, v_z] = [u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x]$$

Teraz si vieme ľahko overiť, že takto vymyslený súčin je to, čo chceme, totiž že  $\vec{u} \times \vec{v}$  je kolmý na  $\vec{u}$  aj  $\vec{v}$ . Stačí použiť, čo sme si ukázali pred chvíľou, totiž že kolmé vektorov majú skalárny súčin 0. Preto rátajme

$$\begin{aligned} (\vec{u} \times \vec{v}) \cdot \vec{u} &= [u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x] \cdot [u_x, u_y, u_z] = \\ &= (u_y v_z - u_z v_y) u_x + (u_z v_x - u_x v_z) u_y + (u_x v_y - u_y v_x) u_z = \\ &= \textcolor{teal}{(u_y v_z u_x)} - \textcolor{blue}{(u_z v_y u_x)} + \textcolor{red}{(u_z v_x u_y)} - \textcolor{blue}{(u_x v_z u_y)} + \textcolor{red}{(u_x v_y u_z)} - \textcolor{blue}{(u_y v_x u_z)} = \\ &= 0 \end{aligned}$$

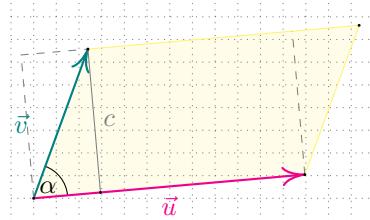
Preto  $\vec{u} \times \vec{v}$  je kolmý na  $\vec{u}$  a rovnako by sme vyrátali, že je kolmý na  $\vec{v}$ .

Vektorový súčin sa dá, okrem počítania normálne v 3D, použiť napr. na počítanie orientácie vektorov v 2D. Zoberme vektorov  $\vec{u} = [u_x, u_y]$  a  $\vec{v} = [v_x, v_y]$ . Doplníme ich do troch rozmerov, ako keby ležali v rovine so  $z = 0$ :  $[u_x, u_y, 0]$  a  $[v_x, v_y, 0]$  a vyrátam vektorový súčin  $[0, 0, u_x v_y - u_y v_x]$ . Ak je  $z$ -ová súradnica kladná, uhol, ktorý vektorov zvierajú je orientovaný proti smeru hodinových ručičiek, ak je záporná, tak v smere.

Posledná užitočná vlastnosť vektorového súčinu, ktorú tu spomeniem, je jeho veľkosť. Podobnými počtami, ako pri skalárnom súčine<sup>4</sup> sa dá vidieť, že

$$\|\vec{u} \times \vec{v}\| = \|\vec{u}\| \cdot \|\vec{v}\| \cdot \sin(\alpha)$$

Ak mám v priestore dva vektorov, ktoré nie sú rovnobežné, definujú mi rovnobežník takto:



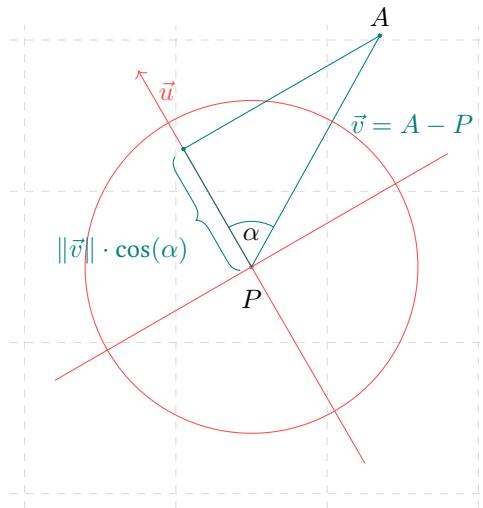
Lahko vidíš, že  $c = \|\vec{v}\| \cdot \sin(\alpha)$ , a že presunutím odstávajúceho trojuholníka vpravo dostaneme obdĺžnik s obsahom  $\|\vec{u}\| \cdot \|\vec{v}\| \cdot \sin(\alpha)$ . Preto  $\|\vec{u} \times \vec{v}\|$  je obsah rovnobežníka, ktorý je ohraničený vektorom  $\vec{u}$  a  $\vec{v}$ .

**Úloha 117.** Naprogramuj triedy `Vec` a `Vec3` pre dvoj- a trojrozmerné vektorov, s operáciami sčítania, odčítania, násobenia číslom, skalárneho a vektorového súčinu (pre vektorový súčin môžeš použiť napr. operator`^`).

<sup>4</sup>iba trochu dlhšími, preto ich tu nebudem písat

## Naspäť k Perlinovi

Predchádzajúce matematické intermezzo sme začali v situácii, že posledná vec, ktorú sme potrebovali na rátanie Perlinovho šumu, bolo vyrátať výšku bodu v gradiente. Dajme tomu, že v bode  $P$  máme umiestnený gradient, ktorý má smer vektora  $\vec{u}$ , pričom  $\|\vec{u}\| = 1$ . Chcem vyrátať výšku bodu  $A$  v tomto gradiente. Ak si označím vektor  $\vec{v} = A - P$ , tak výška, ktorú potrebujem, je  $\|\vec{v}\| \cdot \cos(\alpha)$ :



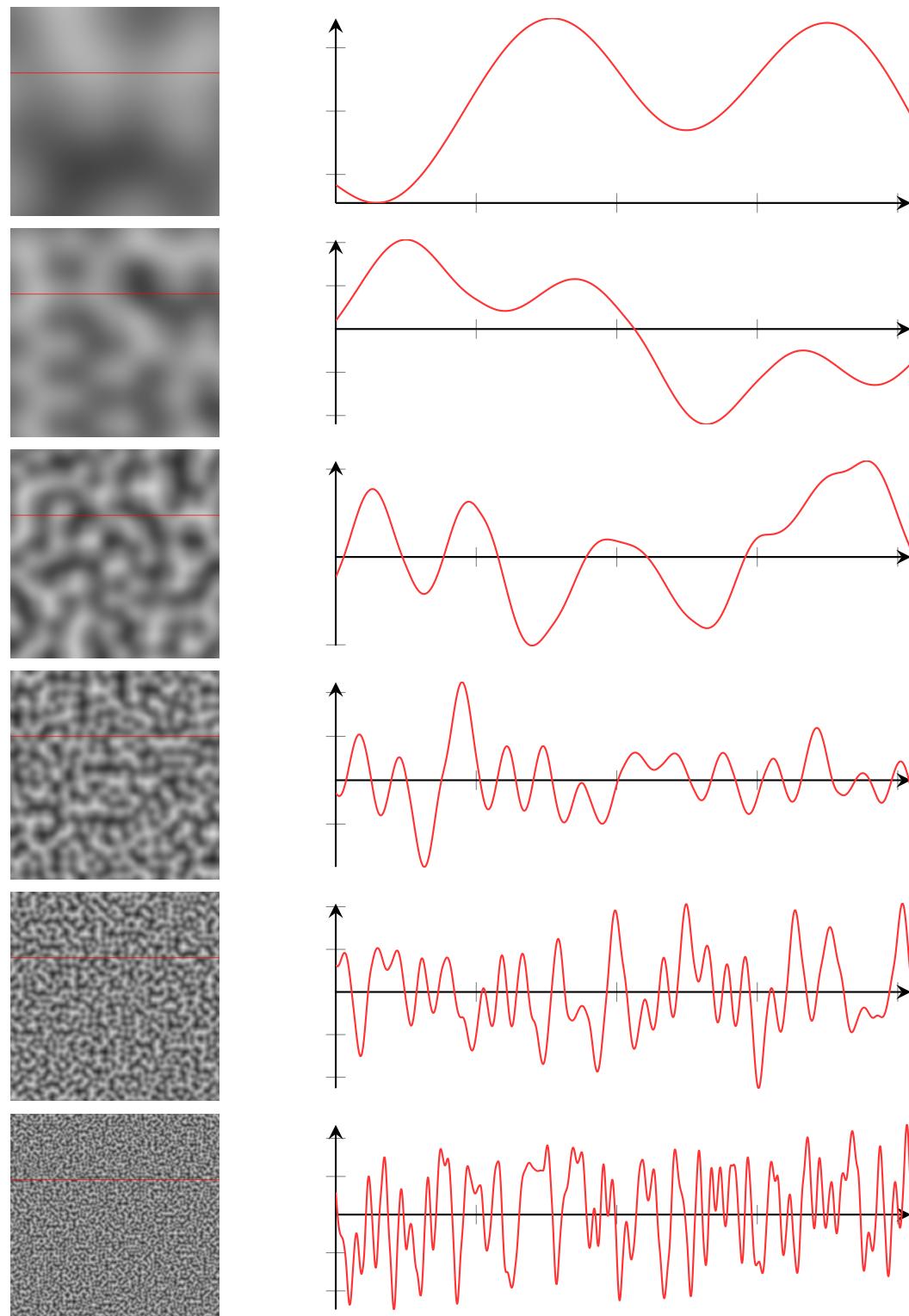
Viem, že skalárny súčin  $\vec{u} \cdot \vec{v} = \|\vec{u}\| \cdot \|\vec{v}\| \cdot \cos(\alpha)$ . Pretože  $\|\vec{u}\| = 1$ , stačí mi zrátat  $\vec{u} \cdot \vec{v}$  a mám hľadanú výšku.

**Úloha 118.** Dokonči triedu `Perlin`.

Keď už vieme robiť základný Perlin noise, môžme z neho vytvárať rôzne kombinácie. Rôzne rozostupy mriežky s gradientami vyrábajú rôzne hustý šum. Prvých 6 veľkostí by mohlo vyzerať ako nasledujúci obrázok (vľavo je celý šum, vpravo je priebeh výšok na reze v červenej čiare):

Projekt: mapy náhodných ostrovov

---

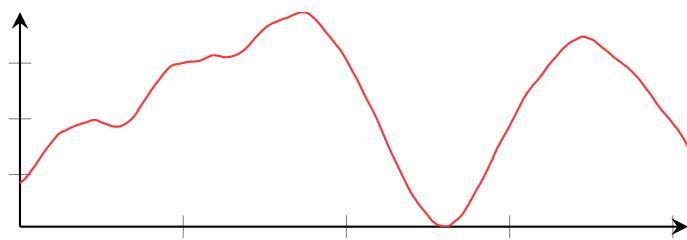
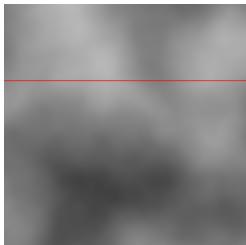


Vo výsledku chceme, aby hustý šum bol nízky a naopak riedký šum bol vysoký. Dosiahneme to napr. tak, že začneme s najhustejším šumom a postupujeme zakaždým k dvojnásobne redším. V každej iterácii najprv výsledok zmenšíme tak, že ho prenásobíme nejakým koeficientom, napr. v mojom prípade 0.6, potom vyrobíme novú premennú typu [Perlin](#) s redším šumom a jej tabuľku  $H$  (ktorá vznikla v konštruktore) pripočítame k výsledku:

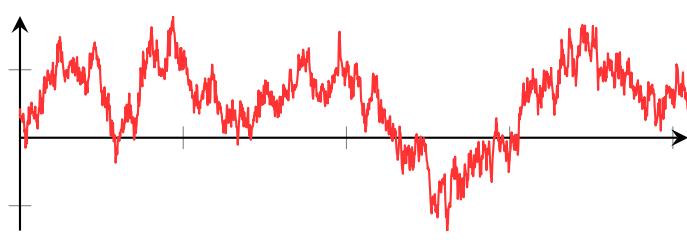
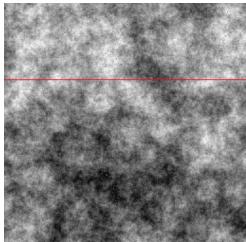
```

1 Tabulka<double> P(n, n);
2 for (int t = 2; t < n; t *= 2) {
3     P *= coeff;
4     P += Perlin(n, t).H;
5 }
```

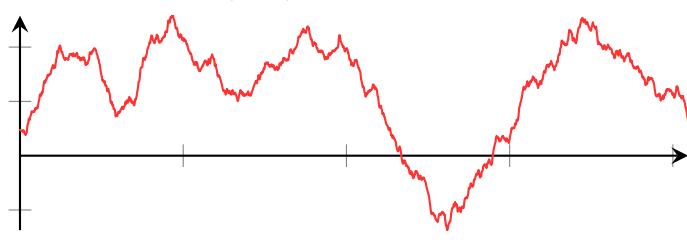
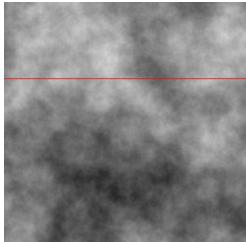
Čím je koeficient `coeff` väčší, tým rovnomernejšie sú vo výsledku zastúpené všetky veľkosti šumu, a tým viac sa blížime k náhodnému šumu. Pre `coeff = 0.3` by som dostať



a pre `coeff = 0.8`



V mojom obrázku som zobraľ `coeff = 0.6` a dostať som takýto výsledok:



Toto bude nateraz naša základná výšková mapa. Ofarbiť ju je ľahké: zoberiem si nejaký gradient, podobný ako pri Mandelbrotovej množine a podľa výšky priradím farbu. To by si nemal mať problém naprogramovať. Predsaden ale niekoľko poznámok: pretože o chvíľu budeme ešte s farbami pracovať, je lepšie namiesto triedy

`struct Farba { unsigned char r, g, b, a };`; používať pre farebné zložky reálne čísla od 0 do 1, napr. ako `Vec3` a do celých čísel ich prekonvertovať až pri zápisе do súboru, napr. takto:

```

1 void zapis(Tabulka<Vec3>& T, const char* fname) {
2     vector<unsigned char> data(T.m * T.n * 4);
3     for (int i = 0; i < T.m; i++) {
4         for (int j = 0; j < T.n; j++) {
5             int offs = 4 * ((T.n - j - 1) * T.n + i);
6             data[offs++] = min(255 * T(i, j).x, 255.0);
7             data[offs++] = min(255 * T(i, j).y, 255.0);
8             data[offs++] = min(255 * T(i, j).z, 255.0);
9             data[offs++] = 255;
10        }
11    zapis_rgba_png(T.n, T.m, data.data(), fname);
12 }
```

## Projekt: mapy náhodných ostrovov

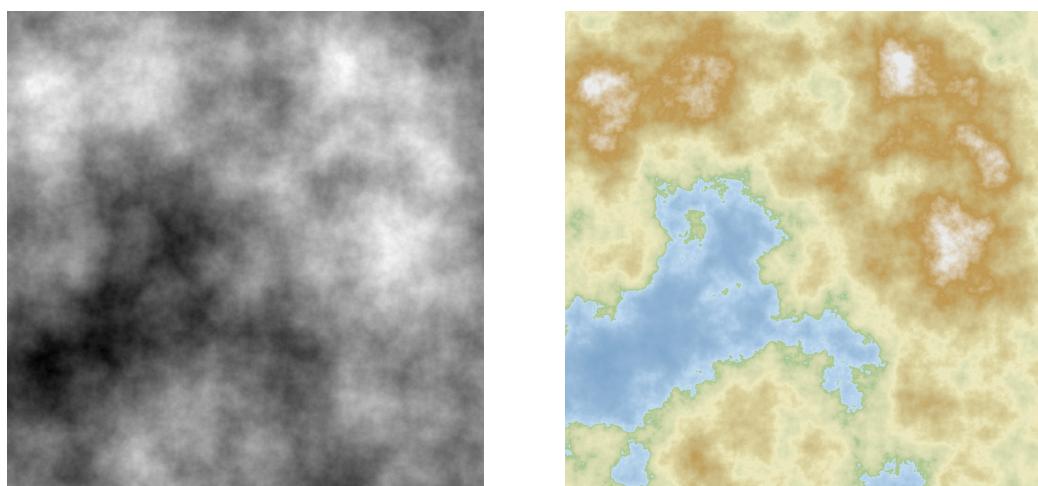
Pre pekné gradienty som sa inšpiroval zo stránky [cpt-city](http://soliton.vm.bytemark.co.uk/pub/cpt-city/)<sup>5</sup>. Je ich tam veľa a väčšina z nich sa dá voľne použiť<sup>6</sup>. Formát súboru **gpf** je jednoduchý: v každom riadku sú 4 čísla: prvé udáva pozíciu v gradiente (od 0 do 1) a potom nasledujú tri zložky pre **r g b** (každé od 0 do 1). Riadok, ktorý sa začína znakom '#' je komentár. Takže môžeš experimentovať s tým, že si nejaký gradient stiahneš a upraviš si ho, aby sa ti hodil.

**Úloha 119.** Naprogramuj triedu **Gradient**, ktorá funguje podobne ako gradient z Mandelbrotovej množiny, ale vie čítať **.gpf** súbory a pracuje s farbou uloženou vo **Vec3**.

Pri ofarbovaní je dobré mať dva gradienty – **gWater** na vodu a **gLand** na pevninu<sup>7</sup>. Ak mám v premennej **Tabuľka<double> H(n, n)** uloženú výškovú mapu, tak tabuľku **Tabuľka<Vec3> F(n, n)** s farbami vyrobím tak, že si poviem výšku hladiny **wl** (tu som si zobraľ **-0.22**), všetko pod ňou farbím vodným gradientom a všetko nad ňou pevninovým. Keďže gradient má rozsah od 0 do 1, nájdeme si maximálnu a minimálnu výšku a pri ofarbovaní si rozdiel skutočnej výšky a vodnej hladiny predelím rozdielom od maxima, resp. minima tak, aby som dostal<sup>8</sup> číslo od 0 do 1:

```
1 minl = 1e50, maxl = -1e50;
2 for (int i = 0; i < H.m; i++)
3     for (int j = 0; j < H.n; j++)
4         if (H(i, j) < minl)
5             minl = H(i, j);
6         else if (H(i, j) > maxl)
7             maxl = H(i, j);
8
9 for (int i = 0; i < H.m; i++)
10    for (int j = 0; j < H.n; j++)
11        if (H(i, j) > wl)
12            F(i, j) = gLand((H(i, j) - wl) / (maxl - wl));
13        else
14            F(i, j) = gWater((H(i, j) - minl) / (wl - minl));
```

Z výškovej mapy vľavo som dostal ofarbený obrázok vpravo:



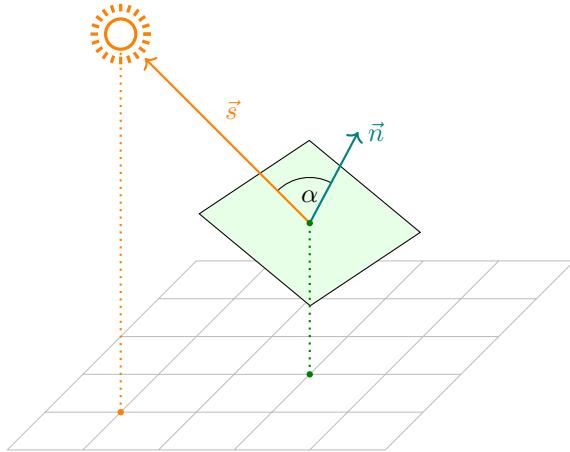
<sup>5</sup><http://soliton.vm.bytemark.co.uk/pub/cpt-city/>

<sup>6</sup>treba si ale prečítať licenciu

<sup>7</sup>V mojich obrázkoch som použil gradient afrikakarte-bath a afrikakarte-topo od používateľa Lilleskut: <http://soliton.vm.bytemark.co.uk/pub/cpt-city/wkp/lilleskut/index.html>

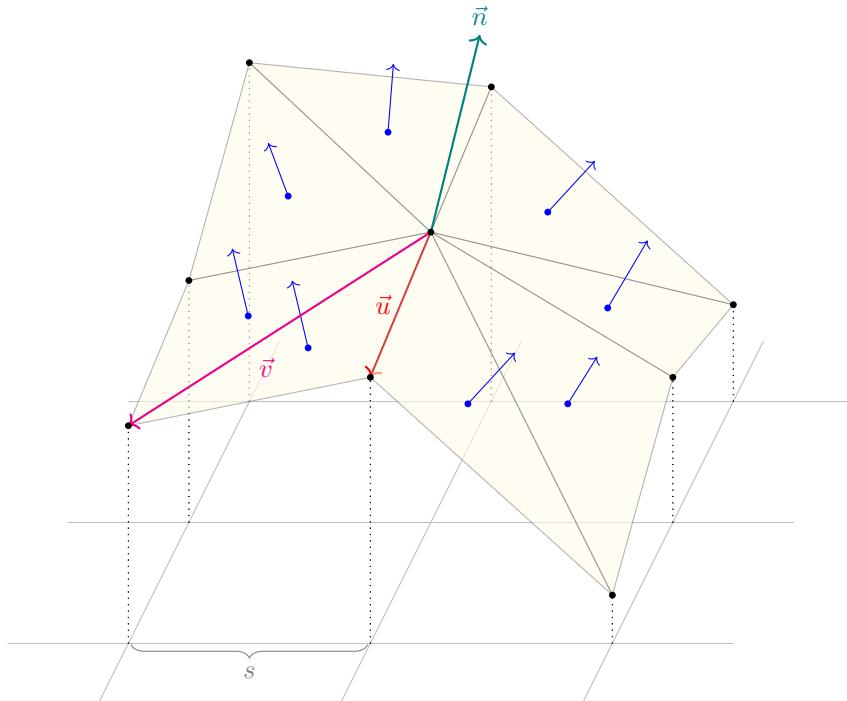
<sup>8</sup>Napríklad ak **H(i, j)** je na pevnine, tak **H(i, j)-wl** udáva, ako vysoko je pixel nad vodnou hladinou. Najvyšší bod je **maxl-wl** nad vodnou hladinou, preto **(H(i, j)-wl) / (maxl-wl)** nadobúda hodnoty od 0 keď **H(i, j)** je presne na hladine po 1, ak **H(i, j)** je najvyššie možné.

Teraz bude treba zafarbený obrázok vytieňovať. Na to použijeme jednoduchý osvetľovací model (tzv. Lambertov). Predstav si, že máš malú plôšku, ktorá má normálový vektor  $\vec{n}$  a nejakú základnú farbu. Čím na ňu dopadá menej svetla, tým je tmavšia. Ako viem zistiť, kolko svetla na ňu dopadá? Ak svetlo prichádza z nejakého smeru, pozriem sa na uhol medzi zdrojom svetla s normálovým vektorom:



Kosínus tohto uha mi vyjadruje množstvo dopadajúceho svetla: ak plôška smeruje presne na zdroj svetla, uhol je  $0^\circ$ ,  $\cos(0) = 1$ , takže farba je najsilnejšia. Ako uhol rastie,  $\cos(\alpha)$  postupne klesá, až keď je uhol  $90^\circ$ , tak nedopadá žiadne svetlo  $\cos(90^\circ) = 0$ . Ak  $\vec{s}$  je vektor smerom k svetelnému zdroju, stačí mi znormálizovať ho a vyrátať skalárny súčin s  $\vec{n}$  a mám  $\cos(\alpha)$ , ktorý potrebujem.

Posledná otvorená otázka je, ako vyrátať normálový vektor pre nejaký pixel vo výškovej mape. Môžem to spraviť napríklad tak, že sa pozriem na 8 okolitých pixelov. Každý z nich mi dáva trojuholník, ktorému viem vypočítať normálový vektor pomocou vektorového súčinu. Výsledný vektor potom dostonem ako priemer normálových vektorov trojuholníkov:



Naprogramovať sa to dá napríklad takto:

## Projekt: mapy náhodných ostrovov

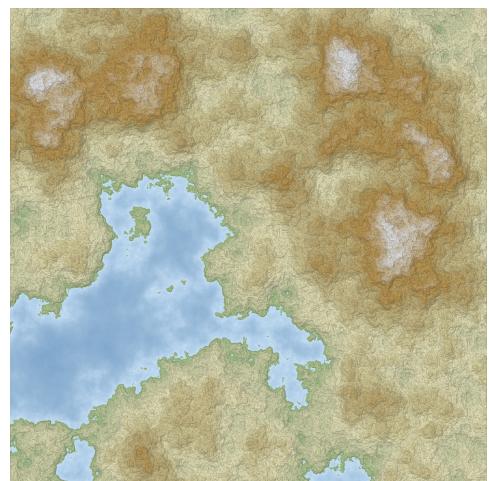
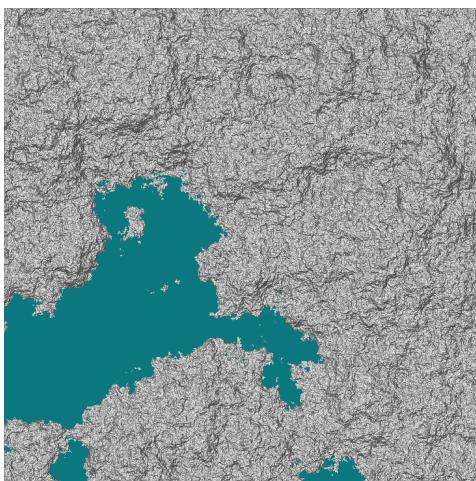
```
1 Vec3 normala(Tabulka<double> &H, int i, int j, double s = 1) {
2     Vec3 res;
3     vector<Vec> dir{{0, 1}, {1, 1}, {1, 0}, {1, -1}, {0, -1},
4                     {-1, -1}, {-1, 0}, {-1, 1}, {0, 1}};
5     for (int t = 0; t < dir.size() - 1; t++) {
6         int i0 = i + dir[t].x, j0 = j + dir[t].y,
7             i1 = i + dir[t + 1].x, j1 = j + dir[t + 1].y;
8         res +=
9             -1.0 *
10            (
11                (Vec3{s * dir[t].x, s * dir[t].y, H(i0, j0)}
12                 - Vec3{0, 0, H(i, j)})
13                 ^ // vektorový súčin
14                (Vec3{s * dir[t + 1].x, s * dir[t + 1].y, H(i1, j1)} -
15                 Vec3{0, 0, H(i, j)}));
16    }
17 }
18 return res.normalize();
19 }
```

Táto funkcia dostane ako parameter tabuľku  $H$  a pozíciu pixela  $i, j$ . Navyše dostanem parameter  $s$ , ktorý hovorí, ako ďaleko sú od seba jednotlivé pixely (t.j. výškovú mapu si predstavujem ako sieť s krokom  $s$ ).

Pre obrázok s rozmermi  $2048 \times 2048$  som si nastavil smer svetla na  $\text{Vec3 light}\{-1, 1, 12\}$ ; a  $\text{double } s=0.004$ ; a z výškovej mapy  $H$  som si vyrátal tabuľku  $\text{Tabulka<double>} N$  takto:

```
1 light.normalize();
2 for (int i = 0; i < H.m; i++)
3     for (int j = 0; j < H.n; j++)
4         if (H(i, j) <= w1)
5             N(i, j) = 1;
6         else
7             N(i, j) = light * normala(H, i, j, s);
```

Výsledok je na obrázku vľavo. Obrázok vpravo vznikol tak, že som každú farbu pixela na ofarbenej mape prenásobil koeficientom<sup>9</sup>  $0.6 + 0.4 * \max(0.0, N(i, j))$ ;

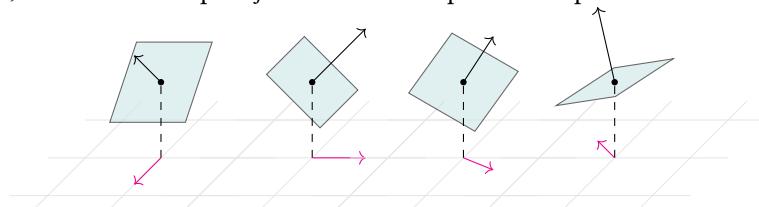


<sup>9</sup>t.j. zobraľ som 60% pôvodnej farby a 40% som rozdelil od čiernej po pôvodnú farbu podľa normály. Nie je to úplne ideálne, ale môže nám to postačiť.

Toto už vyzerá ako mapa terénu, aj keď zrovna ostrov to nie je. To sa dá ale ľahko napraviť pri generovaní výškovej mapy: okrem Perlinovho šumu sa navyše každý bod posunie dole úmerne jeho vzdialosti od stredu obrázka. Okraje klesnú a v strede ostane ostrov. S trochou hrania sa s klesaním okrajov a zvýraznením hôr som dospel k takému výsledku:



Úplne posledná vec, ktorá tomu chýba, sú doliny riek a fjordy. Rozhodol som sa preto spraviť jednoduchú fyzikálnu simuláciu vodnej erózie. Urobiť realistickú virtuálnu eróziu je téma sama osobe, ale v [bakalárskej práci Hansa Beyera<sup>10</sup>](#) je veľmi zjednodušený a ľahko naprogramovateľný prístup, ktorý si tu môžeme vyskúšať. Hlavná idea je, že na povrch našej výškovej mapy budeme púštať kvapky<sup>11</sup>. Kvapka má nejaký objem vody a v nej je rozpustený nejaký objem bahna. Ak je bahna málo a kvapka ide rýchlo, vymye kúsok z povrchu: výšková mapa sa trochu zníží a v kvapke bude viac bahna. Naopak, ak je bahna veľa a kvapka ide pomaly, bahno sa bude usadzovať: v kvapke ho ostane menej a povrch sa trochu zvýši. Ako zistíme, ktorým smerom sa bude kvapka pohybovať? Krátka odpoveď je, že v smere, ktorým ukazuje normálový vektor. Dalo by sa to pomerne jednoducho spočítať, ale nateraz sa uspokojme s intuitívной predstavou podľa obrázka



Predstav si malú plôšku, z ktorej trčí normálový vektor ako klinec. Ktorým smerom klincom pohneš, tým smerom sa plôška nakloní, a tade bude aj kvapka padať. Takže stačí nám zrátať normálový vektor v 3D a

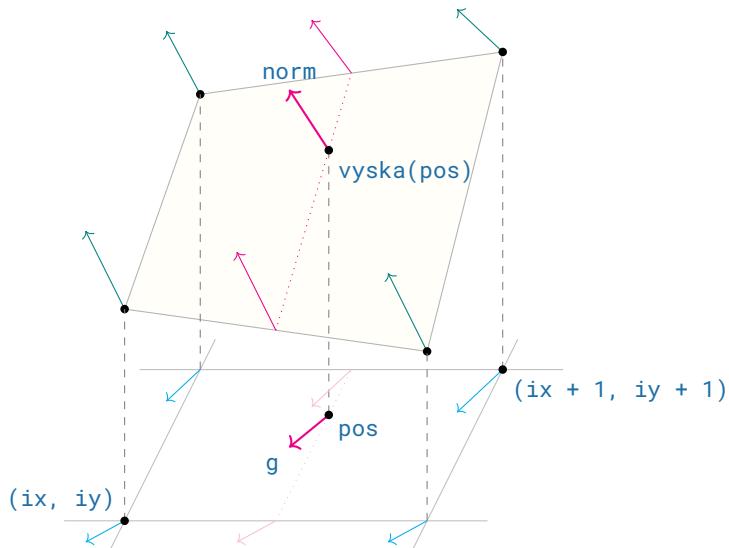
<sup>10</sup><https://www.firespark.de/resources/downloads/implementation%20of%20a%20methode%20for%20hydraulic%20erosion.pdf>

<sup>11</sup>Vzhľadom na mierku, ktorú máme, to bude skôr ako futbalový štadión plný vody, ale volajme to kvapka.

## Projekt: mapy náhodných ostrovov

pozrieť sa na jeho priemet, t.j. na zložky v smere  $x$  a  $y$ . Výsledný 2D vektor sa volá *gradient* a udáva smer, ktorým plôška klesá.

Pre účely simulácie si budeme pozíciu kvapky pamätať v reálnych číslach ako premennú `Vec pos`. Výšková mapa mi bude udávať výšku v bodoch s celočíselnými súradnicami (tieto budem volať *mrežové body*). Pre danú pozíciu si skutočnú výšku vyrátame bilineárhou interpoláciou zo štyroch susedných mrežových bodov. Normálmu vyrátam podobne – najprv vyrátam normálové vektory v štyroch susedných mrežových bodoch a výsledok zrátam bilineárhou interpoláciou.



Simulácia jednej kvapky bude vyzerať takto. Kvapka sa na začiatku zajví na náhodnej pozícii

```
1 Vec pos{dis(rnd) * n, dis(rnd) * n};
```

Budeme simulať niekolko krokov, na to budeme mať konštantu `lifetim`. V každom kroku vyrátame gradient `Vec g`; tak, ako sme to práve opísali. Nový smer kvapky `dir` získame tak, že interpolujeme gradient a predchádzajúci smer s parametrom `zotrvacnost`, t.j. budeme robiť `dir = lerp(g, dir, zotrvacnost)`. Výsledný smer normalizujeme tak, aby sa kvapka pohla o vzdialenosť jedna. Vyrátame pôvodnú výšku a novú výšku a ich rozdiel označíme `delta`.

O kvapke si navýše pamätáme rýchlosť<sup>12</sup>, objem vody (číslo od 0 do 1) a množstvo rozpusteného bahna. Kolko bahna sa z kvapky môže usadiť závisí od sklonu, rýchlosťi a množstva vody:

```
double dasausadit = max(-delta, min_sklon) * rychlos * voda * rozpustnos;
```

Pričom `min_sklon` a `rozpustnos` sú konštanty, ktoré si na začiatku nejak nastavíme. Teraz môžu nastať dva prípady: ak kvapka teče dokopca, alebo nesie viac bahna, ako `dasausadit`, bahno sa bude usádzať. Ak teče dokopca (t.j. `delta > 0`) bahno sa snaží vyplniť rozdiel výšok, t.j. usadí sa `min(delta, bahno)`. V opačnom prípade sa usadí (`bahno - dasausadit`) \* `usadzanie`; kde `usadzanie` je opäť ďalší parameter z rozsahu 0 až 1. Usadené bahno odrátame z bahna v kvapke a prirátame ho k výškovej mape. Pri tom použijeme zase bilineárnu interpoláciu, t.j. ak množstvo usadeného mahna máme v premennej `usadit`, tak do výškovej mapy `H` rozdelíme usadeninu podľa pozície kvapky medzi meržovými bodmi tak, že bližšie mrežové body dostanú viac a vzdialenejšie menej<sup>13</sup>:

<sup>12</sup>Rýchlosť používam iba pri výpočte erózie. Pri určovaní smeru sa vždy pohnem o jednotku. To je kvôli tomu, aby rýchle kvapky nepreskakovali mrežové body a pomalé kvapky sa zbytočne nesimulovali veľakrát na tom istom mieste.

<sup>13</sup>Nasledovný program je možno trochu neprehľadný, ale vyskúšaj si, čo znamená pre konkrétné mrežové body. Napr. pre `dx=0, dy=0` vyplynie, že `H(ix, iy)` dostane časť  $(1 - p.x) * (1 - p.y)$ . Ak napr. `p` je na pozícii `(ix, iy)`, tak `p.x` aj `p.y` je 0 a všetok materiál ide do `H(ix, iy)`. Ak by `pos` bol na opačnom konci na pozícii `(ix+1, iy+1)`, tak do `H(ix, iy)` nejde nič.

```

1 int ix = pos.x, iy = pos.y;      // mrežový bod po zaokrúhlení pozicie
2 Vec p{pos.x - ix, pos.y - iy};  // pozícia v rámci gridu
3
4 for (int dx = 0; dx < 2; dx++)
5   for (int dy = 0; dy < 2; dy++) {
6     H(ix + dx, iy + dy) += usadit * (dx * p.x + (1 - dx) * (1 - p.x) *
7                                         (dy * p.y + (1 - dy) * (1 - p.y)));
8   }

```

Ak sa bahno neusádzia, tak nastáva erózia: časť materiálu z výškovej mapy sa presunie do kvapky. Aká veľká časť sa uberie, kontroluje ďalší parameter [erózia](#). Materiál budeme uberať nielen zo susedných mrežových bodov, ale zo širšieho okolia, aby nevznikali príliš úzke a strmé kaňony:

```

1 double urvat = min((dasausadit - bahno) * erozia, -delta);
2
3 // odoberie zo širšieho okolia podľa toho, ako sú nastavené váhy
4 for (int dx = 0; dx < 2 * r + 1; dx++)
5   for (int dy = 0; dy < 2 * r + 1; dy++) {
6     int i = ix - r + dx, j = iy - r + dy;
7     double kusok = urvat * vahy[dx][dy];
8     H(i, j) -= kusok;
9     bahno += kusok;
10  }

```

Pole [vahy](#) bude ako "štetec" rozmerov  $2r + 1 \times 2r + 1$ , ktorý si doredu pripravíme tak, aby pokrýval okolie do vzdialenosťi  $r$ , v strede mal väčšie hodnoty ako na krajoch a dokopy mal súčet 1 takto (číslo v poličku je váha v percentách):

		0.4	0.7	0.8	0.7	0.4		
	0.6	1.1	1.4	1.5	1.4	1.1	0.6	
0.4	1.1	1.7	2.1	2.3	2.1	1.7	1.1	0.4
0.7	1.4	2.1	2.8	3.1	2.8	2.1	1.4	0.7
0.8	1.5	2.3	3.1	3.8	3.1	2.3	1.5	0.8
0.7	1.4	2.1	2.8	3.1	2.8	2.1	1.4	0.7
0.4	1.1	1.7	2.1	2.3	2.1	1.7	1.1	0.4
	0.6	1.1	1.4	1.5	1.4	1.1	0.6	
		0.4	0.7	0.8	0.7	0.4		

To sa dá naprogramovať napr. takto:

```

1 double sum = 0;
2 for (int dx = 0; dx < 2 * r + 1; dx++)
3   for (int dy = 0; dy < 2 * r + 1; dy++) {
4     // vzdialenosť od stredu
5     double dist = sqrt((r - dx) * (r - dx) + (r - dy) * (r - dy)) / (r + 1);
6     double w = 0;
7     if (dist < 1) w = (1 - dist);
8     vahy[dx][dy] = w;
9     sum += w;
10  }

```

## Projekt: mapy náhodných ostrovov

```
11 for (int dx = 0; dx < 2 * r + 1; dx++)  
12     for (int dy = 0; dy < 2 * r + 1; dy++) vahy[dx][dy] /= sum;
```

Pozícia  $[r][r]$  je presne v strede pola, preto  $\sqrt{(r - dx)^2 + (r - dy)^2}$  je vzdialenosť políčka  $[dx][dy]$  od stredu. Vzdialenosť predelíme  $r + 1$ , aby sme dostali kruh s polomerom 1. Pozície mimo kruhu budú mať hodnotu 0, pozície v kruhu hodnotu úmernú vzdialnosti od stredu. Nakoniec každú hodnotu vydelíme súčtom<sup>14</sup>.

Nakoniec upravíme rýchlosť podľa fyzikou inšpirovaného vzťahu

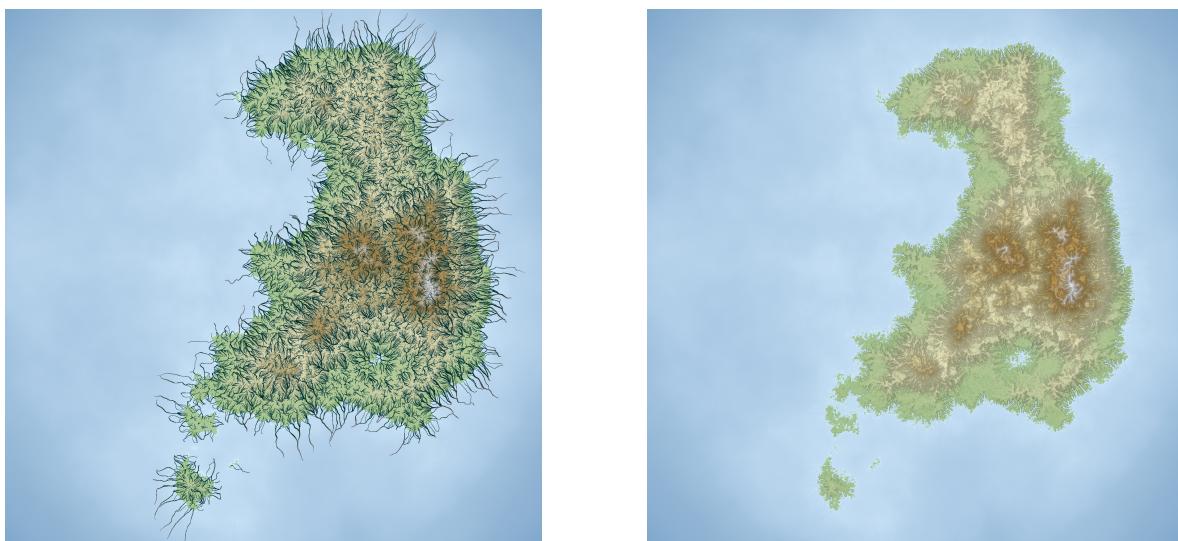
```
1 if (delta < 0) delta *= -1;  
2 rychlosť = sqrt(rychlosť * rychlosť + delta * gravitacia);
```

kde `gravitacia` je parameter, a odparíme časť vody `voda *= (1 - odparovanie)` kde `odparovanie` je ešte iný parameter.

**Úloha 120.** Naprogramuj triedu `Eroder`, ktorá v konštruktore dostane referenciu na výškovú mapu a má funkciu `kvap`, ktorá na nej odsimuluje jednu kvapku.

Výhodou tejto metódy je, že je jednoduchá, nevýhodou je, že má strašne veľa parametrov, ktoré treba správne nastaviť, ak má výsledok vyzeráť dobre<sup>15</sup>.

Po troche hrania sa s parametrami som dostať výsledok, s ktorým som bol viac-menej spokoný. Na obrázku vľavo sú dráhy kvapiek, vpravo je ostrov po erózii.



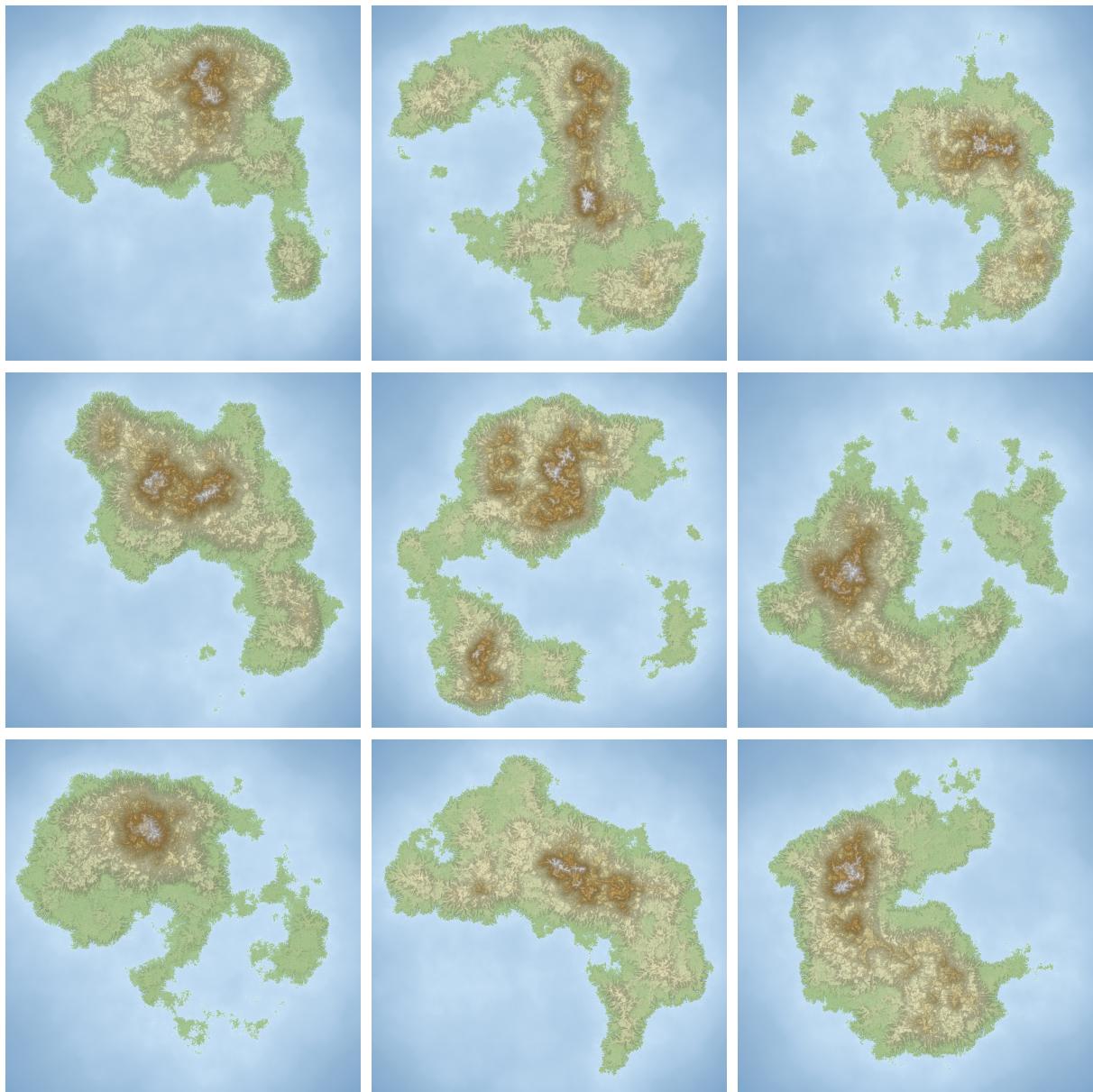
**Úloha 121.** Daj to celé dokopy a naprogramuj generovanie náhodných ostrovov.

Tu je niekoľko ďalších ostrovov, ktoré sa vygenerovali s rôznymi seedmi:

<sup>14</sup>To je typická finta ako zaručiť, že celkový súčet je 1. Ak mám čísla  $a_1, a_2, \dots, a_n$  a označím  $s = a_1 + a_2 + \dots + a_n$ , tak keď zoberiem čísla  $b_1 = \frac{a_1}{s}, b_2 = \frac{a_2}{s}, \dots, b_n = \frac{a_n}{s}$ , tak ich vzájomné relativne veľkosti sa nezmenia, lebo som všetky vydelil rovnakým číslom, a ich súčet bude  $b_1 + \dots + b_n = \frac{a_1}{s} + \dots + \frac{a_n}{s} = \frac{1}{s}(a_1 + \dots + a_n) = \frac{s}{s} = 1$ .

<sup>15</sup>V pôvodnej práci je ešte jedno odporúčanie: keďže pri úpravách výškovej mapy sú zmeny pri jednej kvapke veľmi malé, kvôli reprezentácii reálnych čísel môže písť k podtečeniu: číslo, ktoré prirátavame je tak malé, že sa v mantise neprejaví. Preto je lepšie udržovať si zmeny v separátnej tabuľke až nakoniec ich prirátať k výškovej mape.

Projekt: mapy náhodných ostrovov

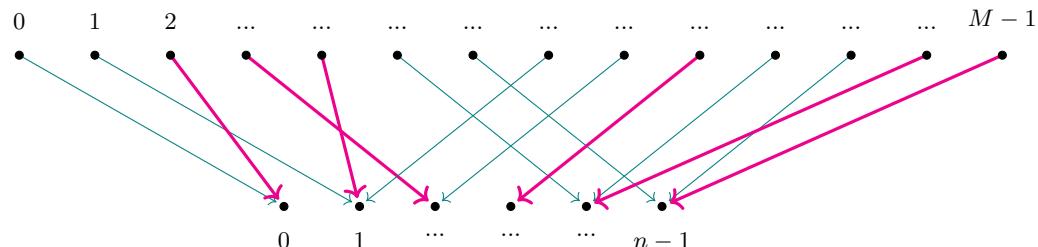


Samozrejme, v tomto projekte by sa dalo pokračovať k realistickejším výsledkom, ale chcel som ti len ukázať, ako sa náhodné čísla dajú použiť na vytváranie vecí, ktoré sú "len trochu" náhodné.

## 32 Hešovanie: `<unordered_set>` a `<unordered_map>` v STL

Vráťme sa k úlohe 98. Opäť budeme mať žiarovky, ktoré budú mať výrobné čísla `unsigned long`, takže budú z rozsahu 0 až  $M - 1$ , kde  $M = 2^{64}$ . Zároveň pre jednoduchší začiatok predpokladajme, že vieme, že budeme mať presne  $n$  žiaroviek. Na vstupe budú opäť chodiť príkazy na prepnutie žiarovky a otázky, či nejaká žiarovka svieti. V kapitole 27 sme to riešili použitím vyhľadávacích stromov. Teraz si ukážeme iný spôsob riešenia, ktorému sa hovorí *hešovanie*.

Keby sme z nejakého dôvodu vedeli, že nám budú v nejakom poradí chodiť žiarovky s číslami 42, 43, 44, ..., 41 +  $n$ , stačilo by rezervovať si pole `bool A[n]` a stav žiarovky s číslom  $z$  by sme si mohli pamätať v premennej `A[z-42]`. Podobne keby sme vedeli, že budú chodiť žiarovky 2, 4, ...,  $2n$ , tiež by stačilo rezervovať pole `bool A[n]` a žiarovku s číslom  $z$  by sme mali v premennej `A[z/2-1]`. V oboch prípadoch by sme mali nejakú funkciu `int h(unsigned long z) { ... }`, ktorá pre číslo žiarovky vyráta jej umiestnenie v poli, takže žiarovka  $z$  je umiestnená v `A[h(z)]`. Ideálne by bolo, keby som mohol mať jednu takú funkciu `h`, ktorá by fungovala pre viacero (ideálne pre všetky) možné vstupy. Každú funkciu `h`, bez ohľadu na to, ako sa vypočíta, si viem predstaviť ako kartičku, kde pre každý z  $M$  bodov v hornom riadku (všetky možné výrobné čísla) ide šípka do nejakého bodu v spodnom riadku (všetky pozície v poli).



Ak používam túto konkrétnu funkciu `h`, tak pri každom spustení sa na vstupe ocitne  $n$  rôznych výrobných čísel (fialové šípky) a o týchto chceme, aby sa zobrazili na rôzne miesta. Keď sa nad tým chvíľu zamyslíš, tak je zrejmé, že to nemôže fungovať: nech sú šípky zoradené hocijako, máme  $M$  šípok a  $n$  cieľov, preto musí byť cieľ, do ktorého smeruje aspoň  $M/n$  šípok. No a keďže  $M$  je veľmi veľké, dá sa nájsť vstup, v ktorom všetkých  $n$  vybratých šípok ukazuje do tej istej pozície.

Napriek tomu, že je jasné, že to nebude fungovať, skúsme byť chvíľu tvrdohlaví a pokračovať v úvahách. Dajme tomu, že by sme mali nejakú takúto *hešovaciu* funkciu `h`, teda nejak vybraté modré šípky. Podľa toho, ktoré výrobné čísla sa objavia na vstupe, niektoré šípky budú fialové. Nedúfame, že naša funkcia bude úplne dokonalá, a tak sa nám možno môže stať, že niekoľko fialových šípok bude smerovať do rovnakého cieľa. Ak ich ale nebude veľa, ešte to nie je stratené. Takéto tzv. *kolízie* môžeme vyriešiť napr. tak, že v našom poli `A` si nebudeme na pozícii `h(i)` ukladať `bool` hodnotu, či je žiarovka s číslom `h(i)` zapnutá, ale spájaný zoznam zapnutých žiaroviek so všetkými výrobnými číslami  $x$ , pre ktoré `h(x) = h(i)`. Môžeme to naprogramovať napr. takto:

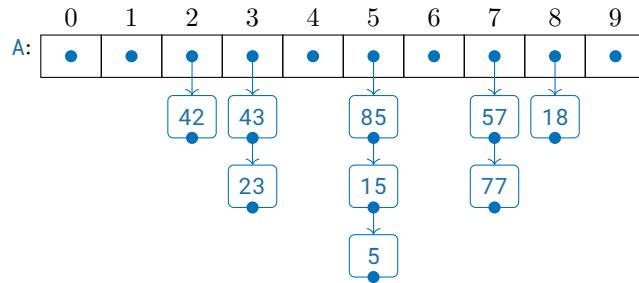
```
1 using Key = unsigned long;
2 using HashFunct = function<int(Key)>;
3
4 struct HashMap {
5     vector<forward_list<Key>> A;
6     HashFunct h;
7
8     HashMap(int n, HashFunct _h) : h{_h} { A.resize(n); }
9
10    void insert(Key x) {
11        auto &list = A[h(x)];
12        if (find(list.begin(), list.end(), x) == list.end())
13            list.push_front(x);
14    }
15 }
```

```

16 void erase(Key x) {
17     auto &list = A[h(x)];
18     auto it = find(list.begin(), list.end(), x);
19     if (it != list.end()) {
20         *it = *list.begin();
21         list.pop_front();
22     }
23 }
24
25 bool contains(Key x) {
26     auto &list = A[h(x)];
27     return find(list.begin(), list.end(), x) != list.end();
28 }
29 };

```

V tomto programe je hešovacia funkcia typu HashFunct, t.j. dostane parameter  $x$  a vráti hodnotu  $h(x)$  z rozsahu  $0, \dots, n - 1$ . Premenná typu `HashMap` dostane v konštruktore dĺžku poľa  $n$  a hešovanciu funkciu, a potom má metódy `insert`, `erase` a `contains`, ktoré sú zjavné. Mohli by sme teda napísat napr. `HashMap h(10, [](Key x) { return x % 10; })`; a po tom, čo postupne zavoláme `insert` s hodnotami `77 5 23 18 43 57 15 43 42 85` to bude v pamäti vyzerať takto:



Akú by mal nás program zložitosť? Každá z funkcií `insert`, `erase`, `contains` najprv vyráta hešovaciu funkciu (nateraz predpokladajme, že to vieme urobiť rýchlo) a potom v najhoršom prípade prehľadá celý spájaný zoznam  $A[h(x)]$ . Takže zložitosť každej operácie je úmerná dĺžke zoznamu  $A[h(x)]$ . V zozname  $A[h(x)]$  skončia všetky žiarovky zo vstupu, ktorých výrobné číslo má v našej hešovacej funkcií kolízii s výrobným číslom  $x$ , t.j. žiarovky s číslom  $z$ , pre ktoré platí  $h(x) = h(z)$ , preto zložitosť nášho programu bude závisieť od počtu kolízii našej hešovacej funkcie.

A sme zase tam, kde sme boli na začiatku: nech si zvolíme hocjakú hešovaciu funkciu, vždy bude existovať vstup, na ktorom bude mať veľa kolízií. Opäť nám ale príde na pomoc náhoda. Na začiatku síce nevieme, aké výrobné čísla budú na vstupe, ale ako funkciu  $h$  si vyberieme náhodnú funkciu. Čo to znamená náhodná funkcia? Funkcia je obrázok, kde z každého z  $M$  výrobných čísel ide modrá šípka do niektorého z  $n$  miest. Kolko je takých rôznych obrázkov? Na obrázku je  $M$  šípok a každá z nich má  $n$  možných cieľov, teda dokopy je  $n^M$  možností. Napíšeme si ich na papieriky a vložíme do vrecka. Potom náhodne vytiahneme jeden obrázok a máme náhodnú funkciu. Kontrolná otázka:

**Úloha 122.** Vyberme si náhodnú funkciu  $h$ . Aká je pravdepodobnosť, že  $h(42) = 1$ ?

Vyšlo ti  $1/n$ ? Dobre. Jedna možnosť, ako sa o tom presvedčiť, je pozrieť sa na všetkých  $n^M$  papierikov a ofarbiť ich podľa toho, aká je hodnota  $h(42)$ : ak  $h(42) = 0$ , papierik bude modrý, ak  $h(42) = 1$ , bude červený atď. Vo vrecku teda budem mať papieriky  $n$  farieb. Koľko je tam papierikov jednej farby? Bez ohľadu na to, kam smeruje šípka  $h(42)$ , ostatné šípky sú umiestené hocjakako, a teda mám  $n^{M-1}$  možností. Pôvodnú úlohu sa teraz viem opýtať takto: vo vrecku mám  $n^M$  papierikov ofarbených  $n$  farbami, pričom z každej farby mám rovnako veľa, t.j.  $n^{M-1}$  papierikov. Aká je pravdepodobnosť, že vytiahnem červený papierik? Tu je ale odpoveď jasná: červených papierikov je  $n^{M-1}$ , preto pravdepodobnosť je  $n^{M-1}/n^M = 1/n$ .

Kolko kolízií so vstupom má náhodná funkcia? To, samozrejme, závisí od toho, aký papierik si vytiahneme. Môžeme mať šťastie a vytiahneme si takú funkciu, čo nebude mať žiadne kolízie, môžeme mať smolu a bude ich mať veľa. Toto je v istom zmysle typická situácia, ktorú sme zažili aj pri porovnávaní súborov: náhoda nám môže pomôcť urobiť veci efektívne, ale zaplatíme za to tým, že s malou pravdepodobnosťou nám to nevyjde. Preto aj teraz nás nebude zaujímať najhorší prípad (ktorý, dúfame, bude nastávať s malou pravdepodobnosťou), ale *očakávaný prípad*.

### Ďalšia odbočka, tentokrát o očakávanej hodnote.

V časti 29 sme hovorili o tom, že môžeme mať nejaký náhodný proces, napr. že hodíme oranžovou a modrou kockou. Výsledkom je nejaký jav, napr. , ktorý má nejakú pravdepodobnosť (v tomto prípade 1/36). Niečo, čo viem vypočítať z výsledku takého náhodného procesu, sa v matematike volá *náhodná premenná*. V príklade s dvoma kockami môžem mať napr. náhodnú premennú  $X$ , ktorá vyráta, kolko bodiek je na tej kocke, na ktorej padlo väčšie číslo, takže napr.  $X(\text{orange dot}) = 5$ ,  $X(\text{green dot}) = 4$ . Pretože  $X$  závisí iba od výsledku náhodného procesu (hodu), môžeme sa napr. pýtať, aká je pravdepodobnosť, že  $X = 3$  (čo budeme značiť  $\Pr[X = 3]$ ). Možnosti, pri ktorých je na väčšej kocke číslo 3, sú , , , a , čiže 5 možností. Všetkých možností je 36, preto  $\Pr[X = 3] = 5/36 \approx 0.14$ . Všetky pravdepodobnosti si môžeme napísať do tabuľky:

$X$												
	1	2	3	4	5	6						
	2	2	3	4	5	6						
	3	3	3	4	5	6						
	4	4	4	4	5	6						
	5	5	5	5	5	6						
	6	6	6	6	6	6						

$i$	1	2	3	4	5	6
$\Pr[X = i]$	$\frac{1}{36}$	$\frac{3}{36}$	$\frac{5}{36}$	$\frac{7}{36}$	$\frac{9}{36}$	$\frac{11}{36}$

Akú hodnotu  $X$  môžeme očakávať, keď hodíme dvoma kockami? S najväčšou pravdepodobnosťou to bude 6, ale aj 5 a 4 budú dosť časte. Preto nie je rozumné povedať, že očakávame ten výsledok, ktorý má najväčšiu pravdepodobnosť. Radšej si predstavme, že by som hádzal veľakrát a zrátajme, aký bude priemer hodnôt  $X$ , ktoré dostanem. Z časti 29 vieme, že ak  $n$ -krát zopakujeme náhodný proces, v ktorom je nejaký jav s pravdepodobnosťou  $p$ , tak nás jav nastane zhruba  $pn$ -krát<sup>1</sup>. Preto keď hodíme  $n$ -krát kockami,  $\frac{n}{36}$ -krát bude hodnota  $X$  jedna,  $n \frac{3}{36}$ -krát dva a tak ďalej. Priemer zo všetkých hodov preto bude

$$\frac{1}{n} \left( \frac{n}{36} + 2n \frac{3}{36} + 3n \frac{5}{36} + 4n \frac{7}{36} + 5n \frac{9}{36} + 6n \frac{11}{36} \right) = \frac{1}{36}(1 + 6 + 15 + 28 + 45 + 66) = 161/36 \approx 4.47.$$

Očakávaná hodnota<sup>2</sup> náhodnej premennej  $X$  je teda 161/36, čo sa zvykne zapísat  $E[X] = 161/36$ .

Predchádzajúce počty sa často opakujú, tak si ich podme zapísavť všeobecne. Majme náhodnú premennú  $X$ , ktorá má hodnoty prirodzené čísla. Aby som si zjednodušil zápis, budem používať  $\Pr[X = i]$  pre hocjaké  $i$  s tým, že pre niektoré  $i$  je  $\Pr[X = i] = 0$  (napr. v našom prípade pre všetky  $i > 6$ ). Po  $n$  opakovaniach budem mať výsledok s hodnotou  $i$  zhruba  $\Pr[X = i]$ -krát, preto priemerná hodnota bude

$$E[X] = \frac{1}{n} (n \cdot \Pr[X = 1] + 2n \cdot \Pr[X = 2] + 3n \cdot \Pr[X = 3] + \dots)$$

Číslo  $n$  sa pri úpravách stratí, preto výsledok viem elegantne zapísat<sup>3</sup> ako

<sup>1</sup>Keby nastal príliš viackrát alebo príliš menejkrát, vedeli by sme urobiť prediktor a už by to nebolo náhodný proces.

<sup>2</sup>V angličtine *expected value*, v slovenčine sa tomu správne hovorí *stredná hodnota*. Treba si dať pozor, že to nie je hodnota, ktorú najčastejšie dostanem (napr. na kocke mi ľahko padne číslo 4.47), ale priemer hodnôt cez veľa pokusov.

<sup>3</sup> Tu som použil matematickú verziu `for` cyklu, ktorá sa značí velkým gréckym písmenom  $\sum$  (*sigma*, čosi ako naše  $s$  v slove *suma*). Podobne ako pri `for` cykle, aj v sume mám premennú, ktorá postupne nadobúda hodnoty od jedného čísla po druhé. Za znakom  $\sum$

$$E[X] = \sum_{i=1}^{\infty} i \cdot \Pr[X = i]$$

Urobme si podobnú náhodnú premennú  $Y$ , ktorá namiesto väčšieho bude vracať menšie číslo z dvoch kociek.

$Y$	⚀⚀	⚀⚁	⚁⚀	⚁⚁	⚂⚀	⚂⚁
⚀	1	1	1	1	1	1
⚁	1	2	2	2	2	2
⚂	1	2	3	3	3	3
⚃	1	2	3	4	4	4
⚄	1	2	3	4	5	5
⚅	1	2	3	4	5	6

$i$	1	2	3	4	5	6
$\Pr[Y = i]$	$\frac{1}{36}$	$\frac{9}{36}$	$\frac{7}{36}$	$\frac{5}{36}$	$\frac{3}{36}$	$\frac{1}{36}$

Strednú hodnotu vypočítame  $E[Y] = \frac{11}{36} + 2 \frac{9}{36} + 3 \frac{7}{36} + 4 \frac{5}{36} + 5 \frac{3}{36} + 6 \frac{1}{36} = \frac{91}{36} \approx 2.53$ . To znamená, že ak budeme hádzať dvoma kockami, tak priemerná hodnota na menšej z nich bude 2.39. Na rátanie strednej hodnoty sa dá pozrieť ešte inak: v tabuľke vľavo sú hodnoty  $Y$  pre všetky možné výsledky. Keď rátame napr.  $\Pr[Y = 2]$  tak sa pozrieme do tabuľky a spočítame, kolkokrát je výsledok 2; v našom prípade dostaneme  $\Pr[Y = 2] = \frac{9}{36}$ . Keď rátame strednú hodnotu, tak číslo  $i = 2$  násobíme  $\Pr[Y = 2]$ , t.j. máme  $2 \frac{9}{36}$ . To je ale to isté, ako keby sme pre každé poličko tabuľky, kde je dvojka (takých je 9) zarátali hodnotu  $2 \frac{1}{36}$ . To platí samozrejme nielen pre dvojkou, takže môžme napísat

$$E[Y] = \frac{1}{36} \cdot \sum_{a=\text{⚀}} \sum_{b=\text{⚁}} Y(a, b)$$

Týmto sme vlastne vyrátali priemerné číslo z tabuľky vľavo. Takže (za predpokladu, že každý výsledok má rovnakú pravdepodobnosť) očakávaná hodnota náhodnej premennej je priemerná hodnota cez všetky možné výsledky. To dáva zmysel: každý výsledok má rovnakú pravdepodobnosť, takže ak veľakrát opakujem hod, každý výsledok stretnem rovnaký počet krát.

Náhodné premenné sú funkcie, ktoré z výsledku náhodného procesu rátajú nejaké číslo. Preto ich môžem napríklad aj sčítať. Takže môžem mať náhodnú premennú  $Z = X + Y$ , pre ktorú napr.  $Z(\text{⚁⚁}) = X(\text{⚁⚁}) + Y(\text{⚁⚁}) = 5 + 3 = 8$ .  $Z$  je vlastne súčet oboch kociek ( $X$  je väčšia a  $Y$  menšia), a teda vyzerá takto:

$Z$	⚀⚀	⚀⚁	⚁⚀	⚁⚁	⚂⚀	⚂⚁
⚀	2	3	4	5	6	7
⚁	3	4	5	6	7	8
⚂	4	5	6	7	8	9
⚃	5	6	7	8	9	10
⚄	6	7	8	9	10	11
⚅	7	8	9	10	11	12

$i$	1	2	3	4	5	6	7	8	9	10	11	12
$\Pr[Z = i]$	$\frac{0}{36}$	$\frac{1}{36}$	$\frac{2}{36}$	$\frac{3}{36}$	$\frac{4}{36}$	$\frac{5}{36}$	$\frac{6}{36}$	$\frac{5}{36}$	$\frac{4}{36}$	$\frac{3}{36}$	$\frac{2}{36}$	$\frac{1}{36}$

môžem mať hocjaký výraz s mojou premennou a výsledky sa sčítajú. Napr.  $1 + 2 + 3 + 4 + 5$  by som mohol napísat  $\sum_{x=1}^5 x$ . Podobne  $1 \cdot 2 + 2 \cdot 3 + 3 \cdot 4 + \dots + n(n+1)$  by som napísal  $\sum_{x=1}^n x(x+1)$ . Často sa hodí si nejaké premenné očíslovať a potom v sume používať index podobne ako v poli. Napr. ak mám čísla  $a_1, a_2, \dots, a_n$ , tak ich priemer je  $\frac{1}{n} \sum_{i=1}^n a_i$ .

## Hešovanie: `<unordered_set>` a `<unordered_map>` v STL

Stredná hodnota je (po troche rátania)  $E[Z] = \sum_{i=1}^{12} i \cdot \Pr[Z = i] = \dots = 7$ , čo je presne  $E[X] + E[Y]$ . Nech by  $X$  a  $Y$  boli hocjaké, môžem si napísať

$$\begin{aligned} E[Z] &= \frac{1}{36} \cdot \sum_{a=\square}^{\square} \sum_{b=\square}^{\square} Z(a, b) = \frac{1}{36} \cdot \sum_{a=\square}^{\square} \sum_{b=\square}^{\square} (X(a, b) + Y(a, b)) = \\ &= \frac{1}{36} \cdot \sum_{a=\square}^{\square} \sum_{b=\square}^{\square} X(a, b) + \frac{1}{36} \cdot \sum_{a=\square}^{\square} \sum_{b=\square}^{\square} Y(a, b) = E[X] + E[Y] \end{aligned}$$

Tu sme to počítali pre hod dvoma kockami, ale asi je jasné, že to rovnako zafunguje pre hocjaké náhodné premenné. Preto pre hocjaké dve náhodné premenné platí

$$E[X + Y] = E[X] + E[Y]$$

Tento vzťah je ozaj užitočný<sup>4</sup> a často používaný, ako hned ukážeme.

Koniec odbočky, naspäť k hešovaniu. Zaujímala nás zložitosť operácií `insert`, `erase`, `contains`, ak budeme predpokladať, že máme náhodnú hešovaciu funkciu `h`. Každá z tých operácií, ak sa pýta na prvok `x`, musí v najhoršom prípade prehľadať zoznam `A[h(x)]`, v ktorom sú všetky prvky zo vstupu, ktoré sa zobrazia na rovnaké miesto ako `x` (t.j. `h(x)==h(i)`). Povedzme, že na vstupe sú žiarovky  $z_1, z_2, \dots, z_n$ . To budú teraz pre nás nejaké fixné čísla a pýtame sa, aký je očakávaný počet počet žiaroviek  $z_i$ , ktoré majú kolíziu s danou žiarovkou  $x$ , ak zoberieme náhodnú hešovaciu funkciu. Označme si  $Z$  náhodnú premennú, ktorá počíta počet kolízií s  $x$ .  $Z$  je teda funkcia, ktorá dostane hešovaciu funkciu a vyráta, koľko žiaroviek spomedzi  $z_1, \dots, z_n$  má kolíziu s  $x$ . A nás zaujíma stredná hodnota  $E[Z]$ . Mohli by sme sa ju pokúsiť vyrátať ako v predchádzajúcej odbočke: zrátať priemerný počet kolízií s  $x$  cez všetky hešovacie funkcie. To vyzerá ako riadna divočina, tak to skúsme spraviť inak. Pre každú žiarovku  $z_i$  si sprawme náhodnú premennú  $Z_i$ . Bude to funkcia, ktorá dostane hešovaciu funkciu a zistí, či konkrétna žiarovka  $z_i$  má kolíziu s  $x$ . Ak áno,  $Z_i$  vráti jednotku, inak nulu<sup>5</sup>. Je jasné, že  $Z = Z_1 + Z_2 + \dots + Z_n$  (resp. napísané kratšie  $Z = \sum_{i=1}^n Z_i$ ). No a z predchádzajúcej odbočky viem, že

$$E[Z] = E\left[\sum_{i=1}^n Z_i\right] = \sum_{i=1}^n E[Z_i].$$

Takže namiesto toho, aby som rátal divokú strednú hodnotu zložitej náhodnej premennej  $Z$ , stačí mi vyrátať stredné hodnoty jednoduchších náhodných premenných  $Z_i$  a výsledky sčítať. Pozriem sa na nejakú  $Z_i$ . Môže nadobúdať hodnoty 0 alebo 1, preto

$$E[Z_i] = 0 \cdot \Pr[Z_i = 0] + 1 \cdot \Pr[Z_i = 1] = \Pr[Z_i = 1].$$

Sú dve možnosti. Ak  $z_i = x$ , tak, samozrejme, nech zoberiem hocjakú hešovaciu funkciu, tak  $h(z_i) = h(x)$ . Preto  $\Pr[Z_i = 1] = 1$ . Druhá možnosť, ak  $z_i \neq x$  je zaujímatejšia. Rovnako, ako sme riešili úlohu 122, sa dá ukázať, že  $\Pr[Z_i = 1] = \frac{1}{n}$ . Takže dokopy mám

$$E[Z] = \sum_{i=1}^n E[Z_i] = \sum_{i=1}^n \Pr[Z_i = 1] = 1 + (n-1)\frac{1}{n} < 2.$$

Skvelé! Ehm, čo sme to vlastne zrátali?  $E[Z]$  je očakávaná dĺžka spájaného zoznamu v mieste `h(z)` pre nejakú konkrétnu žiarovku `z`. Keby som teda riešil úlohu so žiarovkami pomocou triedy `HashMap` zo strany 180, do

<sup>4</sup> Ako vráví známy dialóg zo starého filmu *Adéla ještě nevečeřela*:

- Jak primitívni...

- Ale jak účinné!

<sup>5</sup> Takýmto náhodným premenným sa zvykne hovoriť *indikátorové* alebo *Bernoulliho*.

ktoréj by som na začiatku dal ako parameter náhodnú hešovaciu funkciu, na každú operáciu by mi stačilo v očakávanom prípade prehľadať najviac dvojprvkový zoznam `A[h(x)]`. To vyzerá viac ako nádejne.

Lenže na druhý pohľad zistím, že náhodnú funkciu urobíť neviem. Totiž aj keby som ju mal, ako by som si ju zapamätať? Keď na vstupe príde číslo `x`, musím nejak vedieť zistiť, aká hodnota mu prislúcha a to je v podstate rovnaká úloha, ako tá, čo riešim so žiarovkami. Naďťastie sa ukáže, že nepotrebujem zvolať hešovaciu funkciu náhodne spomedzi všetkých funkcií, ale existujú rôzne sady funkcií, ktoré fungujú rovnako: ak si vyberiem náhodnú funkciu z tej sady, v očakávanom prípade budem mať málo kolízií. Jednu z nich ti ukážem, nech to nevyzerá, že je v tom nejaká mágia.

Predpokladajme teraz, že  $n$  aj  $M$  sú mocniny dvojky, takže  $n = 2^b$  a  $M = 2^m$ . Napr. ak je výrobné číslo `unsigned long` a  $n = 1024$ , tak  $m = 64$  a  $b = 10$ . Hešovacia funkcia `h` bude popísaná tabuľkou  $T$  zloženou z núl a jendotiek, ktorá má  $b$  riadkov a  $m$  stĺpcov. Ak mám vyrátať hodnotu  $h(x)$ , napišen si  $x$  v dvojkovej sústave, bude mať  $m$  cifier. Predstavím si ich ako masku: ak je na nejakom mieste jednotka, vyberiem príslušný stĺpec. Vybrané stĺpce skombinujem operáciou `XOR`, t.j. na pozícii, kde je párný počet jednotiek, bude nula, a na pozícii, kde je nepárný počet jednotiek, bude jednotka. Výsledný stĺpec prečítam ako  $b$ -bitové číslo v dvojkovej sústave. Pre tabuľku  $T$  a číslo  $x$  budem túto operáciu značiť  $T \cdot x$ , ako keby to bolo násobenie<sup>6</sup>. Pre nasledovnú tabuľku  $T$  je  $T \cdot 26 = 12$ :

	32	16	8	4	2	1
26	0	1	1	0	1	0
$T$	1	0	0	0	1	1
	0	1	0	1	0	0
	0	0	1	1	1	0
	0	0	1	1	1	0
	0	0	1	1	1	0

→      →      →      →      →      →

1	8
1	4
0	2
0	1

12

**Úloha 123.** Naprogramuj triedu `HashFunc`, ktorá ako parameter v konštruktore dostane číslo  $b$  a `seed` a vytvorí si náhodnú tabuľku  $T$ . Potom bude mať `int operator()(unsigned long x)` ktorý pre dané `x` vráti  $T \cdot x$ .

Keď sme počítali počet kolízií pri hešovaní s náhodnou funkciu, dostali sme sa to takejto situácie: mali sme pevne daný prvok `x` a prvok  $z_i \neq x$  a pýtali sme sa, aká je pravdepodobnosť (ak zoberieme úplne náhodnú hešovaciu funkciu), že  $z_i$  sa zahešuje na rovnakú pozíciu ako `x`. Vyšlo nám, že je to  $1/n$  a potom všetko dobre dopadlo. Preto si teraz predstavme rovnakú situáciu, iba sa pýtame, aká je pravdepodobnosť, že  $h(z_i) = h(x)$ , ak ako hešovaciu funkciu vybereime náhodnú<sup>7</sup> tabuľku  $T$ . Kedže  $x \neq z_i$ , je nejaká pozícia, kde sa ich bitové zápisy líšia. Dajme tomu, že  $i$ -ty bit  $x$  je 0 a  $i$ -ty bit  $z_i$  je 1. To znamená, že ak mám hocjakú tabuľku  $T$ , tak v  $T \cdot x$  sa  $i$ -ty stĺpec nevyberie, čiže hodnoty v  $i$ -tom stĺpci výsledok  $T \cdot x$  nijak neovplyvňujú. Rozdeľme si teraz všetky možné tabuľky  $T$  do skupín: v jednej skupine budú všetky tabuľky, ktoré sa líšia iba v  $i$ -tom stĺpci a všetko ostatné majú rovnaké. V každej skupine je preto  $n = 2^b$  tabuľiek<sup>8</sup>. My sa pýtame, kolko je tabuľiek, pre ktoré  $T \cdot x = T \cdot z_i$ . V každej skupine je hodnota  $T \cdot x$  rovnaká pre všetky tabuľky z tej skupiny, ale hodnoty  $T \cdot y$  sú všetky rôzne. To preto, lebo každá zmena jedného bitu v  $i$ -tom stĺpci  $T$  zmení príslušný bit vo výsledku  $T \cdot z_i$ :

<sup>6</sup>Ono to vlastne aj násobenie je, ale to tu teraz nemusíme rozoberať.

<sup>7</sup>opäť si predstavme, že všetky možné tabuľky nahádzeme do vrecka a jednu si vytiahneme

<sup>8</sup>tolko je rôznych kombinácií núl a jendotiek v  $i$ -tom stĺpci

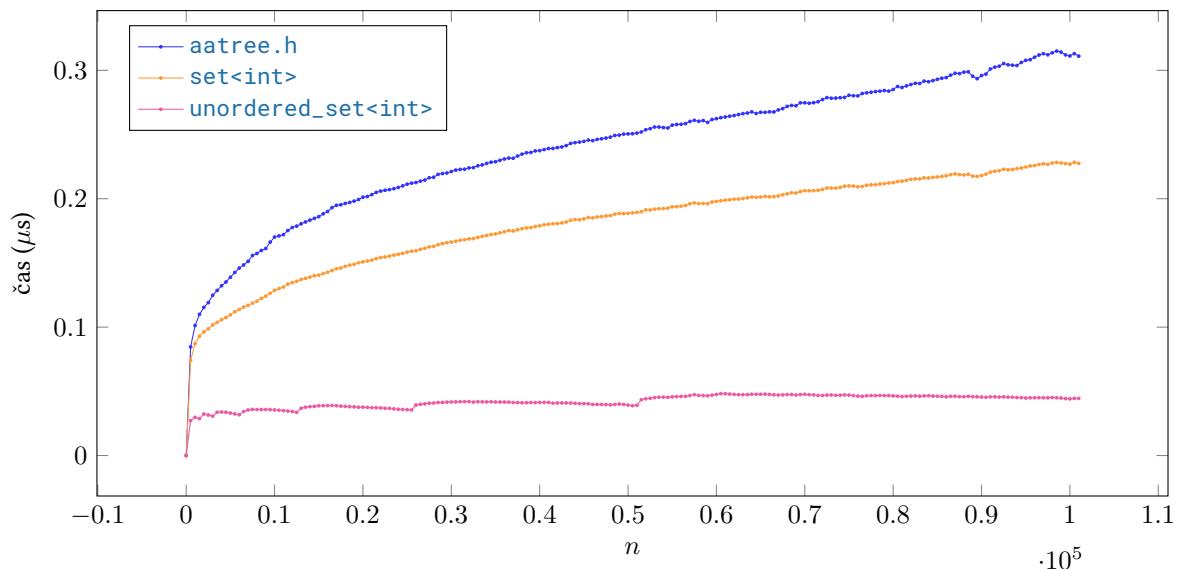
## Hešovanie: `<unordered_set>` a `<unordered_map>` v STL


Takže z každej skupiny je iba jedna tabuľka  $T$ , pre ktorú  $T \cdot z_i = M \cdot x$ , a preto náhodná tabuľka má pravdepodobnosť  $1/n$ , že spôsobí kolíziu  $z_i$  a  $x$ . Zvyšok výpočtu prejde rovnako ako pre úplne náhodné funkcie.

Preto ak spojíš triedu `HashMap` s výsledkom úlohy 123, dostaneš riešenie úlohy so žiarovkami, kde v priemernom prípade každá operácia prezrie iba konštantne dlhý zoznam. Oprávnené ale môžeš namietať, že na vyrávanie hešovacej funkcie, t.j. na zránie  $T \cdot x$  treba zhruba  $\log M \cdot \log n$  operácií, takže sme si oproti vyhľadávaciemu stromu zase až tak nepomohli. To je pravda, ale dajú sa nájsť podobné sady hešovacích funkcií, ktoré majú rovnaké vlastnosti, ale dajú sa zrátať rýchlejšie. Napríklad ak zoberieme nejaké prvočíslo  $p > M$  a náhodnú hešovaciu funkciu urobíme takto: zvolíme sí náhodné čísla  $a, b$  tak, že  $1 \leq a < p$  a  $0 \leq b < p$  a  $h(x)$  bude  $((ax + b) \bmod p) \bmod n$ . Veľmi podobnými, len trochu dlhšími, výpočtami sa dá prísť na to, že aj táto sada hešovacích funkcií funguje, a navyše sa dá aj rýchlo vyrátať.

Posledná vec, čo chýba k úplnosti, je znalosť  $n$ . V úlohe so žiarovkami som zámerne povedal, že je dopredu známe, kolko rôznych výrobných čísel sa objaví. V iných úlohách to tak nie je. Dá sa to však vyriešiť podobne ako v prípade triedy `vector`: budem si sledovať, ako veľmi je moja hešovacia tabuľka zaplnená, a keď sa zaplní privela, zväčším ju a všetky prvky v nej zahešujem nanovo. Tým sa aj zabezpečí, že ak som mal smolu<sup>9</sup> a niektorý zoznam je príliš dlhý, po prehešovaní očakávam, že sa prvky rozdelia rovnomerne.

Môže to celé fungovať? V STL je trieda `unordered_set`, ktorá sa správa skoro rovnako ako nás starý známy `set`, ale namiesto vyhľadávacieho stromu má hešovaciu tabuľku. To spôsobí, že iterátor nedáva prvky v utriedenom poradí, a je iba jednosmerný. Ale zase prístup do `unordered_set` je o dosť rýchlejší:



Záverečné poučenie z tejto časti je, že ak nepotrebuješ mať prvky utriedené, je spravidla niekoľkonásobne rýchlejšie použiť `unordered_set`, resp. obdobnú `unordered_map`.

<sup>9</sup>Alebo mi niekto zámerne zostrojil vstup, v ktorom má moja funkcia veľa kolízií, čo je typický prípad, ak napr. programuješ algoritmy na šifrovanie

Ako všetko v STL, aj `unorederd_set` je šablóna, takže môžeš mať napr. `unordered_set<string>`. Ako to ale funguje, keď doposiaľ všetko okolo hešovania sa dialo na celých číslach? V STL je funkcia, ktorá zo stringu vyráta číslo, a to potom slúži ako kľúč do hešovacej funkcie. Táto funkcia musí byť tiež starostivo vybraná, lebo každé dva stringy, ktoré v nej budú mať rovnaký kľúč, skončia vždy v rovnakom zozname v hešovacej tabuľke, nech by už hešovacia funkcia bola akokoľvek rafinovaná. Takže ak používaš napr. `unordered_set<string>`, treba si uvedomiť, že každý prístup znamená prejsť daný string a zrátať jeho kľúč.

`unordered_set<string>` je v poriadku, ale keď napíšeš napr.

```

1 #include <iostream>
2 #include <string>
3 #include <unordered_set>
4 using namespace std;
5
6 struct Clovek {
7     string meno;
8     int vek;
9 };
10
11 int main() { unordered_set<Clovek> ludia; }
```

kompilátor vypíše dlhý zoznam chýb, kde sa bude asi medzi iným hovoriť niečo ako  
`call to implicitly-deleted default constructor of 'unordered_set<Clovek>'`  
 a možno aj `use of deleted function 'std::hash<Clovek>::hash()'`

Tým ti chce povedať, že pre triedu `Clovek` mu chýba práve tá funkcia, ktorá z premennej typu `Clovek` urobí kľúč do hešovacej tabuľky. Aby si mohol `unordered_set<Clovek>` používať, treba takú funkciu dodať. Ukážem ti ako, ale na to treba povedať, čo je to tzv. *specializácia šablóny*.

Dajme tomu, že mám takúto šablónu

```

1 template <typename T>
2 struct Vec {
3     T x;
4     void daj() { cout << "mam<vec, ale neviem, co<sonou" << endl; }
5 }
```

V programe môžem napísat

```

1 Vec<int> a;
2 Vec<string> b;
3 a.x = 42;
4 b.x = "dvaastyridsat";
5 a.daj();
6 b.daj();
```

Tým hovoríme komplilátoru, aby si zo šablóny vytvoril dva (z pohľadu ďalšieho programu úplne nezávislé) typy `Vec<int>` a `Vec<string>`, vytvoril príslušné premenné atď, a keď to spustíme, dvakrát sa vypíše tá istá veta. Niekedy sa ale hodí urobiť výnimku a povedať komplilátoru napr. *"Všetky typy `Vec<string>`, `Vec<bool>`, atď. rob podľa šablóny, ale typ `Vec<int>` urob inak"*. Takáto výnimka sa volá špecializácia a viem ju napísat takto:

```

1 template <typename T>
2 struct Vec {
3     T x;
4     void daj() { cout << "mam<vec, ale neviem, co<sonou" << endl; }
5 };
```

## Hešovanie: `<unordered_set>` a `<unordered_map>` v STL

```
6 template <>
7 struct Vec<int> {
8     int a;
9     Vec(int x, int y) { a = x + y; }
10    void daj() { cout << "mam int" << a << endl; }
11 };
12 }
```

Zátvorky v `template` ostanú prázdne a za meno typu dosadím parameter, ktorý špecializujem. Takto vyrobený typ nemá s pôvodným nič spoločné, môže mať iné premenné aj metódy. Často je ale rozumné mať (aj) metódy, čo sa volajú rovnako. Teraz by som mohol napísať

```
1 Vec<int> a(5,6);
2 Vec<string> b;
3 b.x = "dvaastyridsat";
4 a.daj();
5 b.daj();
```

a program by vypísal:

```
mam int 11
mam vec, ale neviem, co s nou
```

Niekedy chcem pri špecializácii zmeniť len zopár detailov a chcem, aby väčšina vecí ostala rovnaká, ako v šablóne. V tom prípade môžem špecializovať iba jednotlivé metódy takto:

```
1 template <>
2 void Vec<int>::daj() {cout << "mam int" << x << endl; }
```

Opäť začínam prázdnym `template <>` a potom nasleduje funkcia, ktorú chcem zmeniť. Musím ale povedať, komu patrí, a to robím dvoma dvojbodkami. V tomto prípade hovoríme "*Typ Vec<int> urob podľa šablóny, ale jeho metódu daj() urob takto.*"

Funkcia, ktorá pre rôzne typy hovorí, ako vyrobiť klúč do hešovacej funkcie je v STL urobená cez triedu `hash`. Je to jednoduchá šablóna, ktorá vyzerá zhruba takto

```
1 template <typename Key>
2 struct hash {
3     unsigned long operator()(const Key &x) const { ... }
4 };
```

To znamená, že premenná `hash<T> h` má operátor `h(x)`, ktorý zoberie premennú typu `T` a vráti jej hešovací klúč. Môžeš si to vyskúšať,

```
1 hash<string> h;
2 cout << h("hrom do toho") << endl;
```

vypíše napr. [14868293708368628758](#). Funkcie v `unordered_map` a `unordered_set` používajú príslušný typ `hash` na to, aby vyrobili hešovacie klúče. Samotná šablóna `hash` je naprogramovaná tak, aby spôsobila chybu, všetku prácu musia robiť jej výnimky-špecializácie. Keď chcem používať `unordered_set<Clovek>`, musím<sup>10</sup> preto dodať špecializáciu `hash<Clovek>`. To môžem spraviť napr. takto:

<sup>10</sup>no nemusím, sú aj iné možnosti, ale o nich tu teraz nehovorím

```

1 template <>
2 struct std::hash<Clovek> {
3     unsigned long operator()(const Clovek& c) const {
4         return 42;
5     }
6 };

```

Toto by malo byť jasné, jediná vec, čo stojí za pozornosť je, že nakoľko hash patrí do `namespace std`, tak pri špecializácii musím napísat celé meno `std::hash` aj keď som si na začiatku "privlastnil" všetko zo `std` pomocou `using namespace std;`.

Hešovacia tabuľka v knižnici `<unordered_map>` potrebuje okrem rátania kľúča objekty porovnávať, preto do triedy `Clovek` treba pridať operátor `==`

```

1 struct Clovek {
2     string meno;
3     int vek;
4     bool operator==(const Clovek& c) const {
5         return meno == c.meno && vek == c.vek;
6     }
7 };

```

Teraz sa program, ktorý používa `unordered_set<Clovek>` bude dať skompilovať aj spustiť, len namiesto hešovacej tabuľky budem mať len spájaný zoznam: všetky premenné dostanú rovnaký kľúč 42, a tak sa vždy zahešujú na to isté miesto, kde vznikne dlhý zoznam. Ako môžem získať rozumný kľúč? To je celkom na dlhú diskusiu, ale na bežné účely poslúži aj jednoduchý prístup: ak `string` a `int` už majú dobré špecializácie hash, tak ja len vrátim ich `XOR`

```

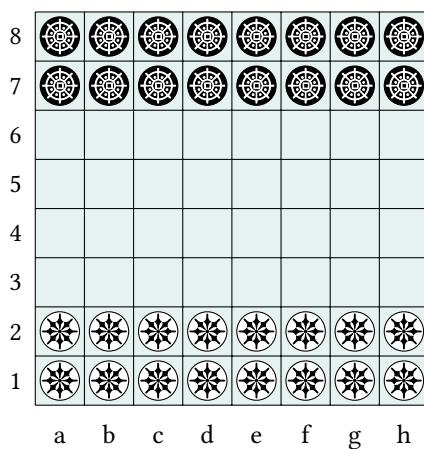
1 template <>
2 struct std::hash<Clovek> {
3     unsigned long operator()(const Clovek& c) const {
4         return hash<string>()(c.meno) ^ hash<int>()(c.vek);
5     }
6 };

```

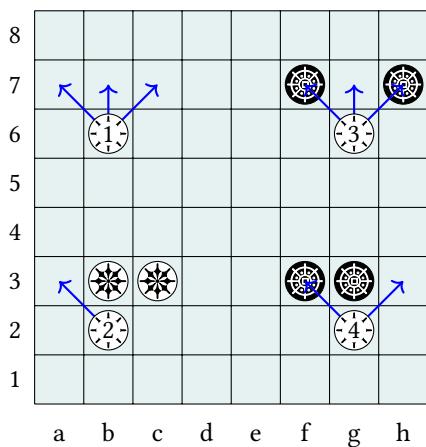
Čo je to za zápis? `hash<string>()` je volanie konštruktora, t.j. vyrobí premennú typu `hash<string>`. Vzápäť zavolám jej `operator()(const string &x)` s parametrom `c.meno` a dostanem nejaké číslo typu `unsigned long`. Dočasnú premennú typu `hash<string>` potom zahodím. Rovnako sa to urobí pre `hash<int>`. Takže nakoniec mám dve čísla, z ktorých urobím bitový `XOR (^)`.

## 33 Projekt Breakthrough

V nasledujúcich častiach ti ukážem niekoľko ďalších vecí v C++, ktoré je ale najlepšie vidieť na trochu väčších programoch. Začneme preto s projektom, ktorý nás bude sprevádzať dlhší čas: našim cieľom bude napísat program, ktorý hrá hru Breakthrough<sup>1</sup>. Hru Breakthrough vymyslel r. 2000 Dan Troyka. Jej výhodou je, že má veľmi jednoduché pravidlá (v porovnaní napr. so šachom), takže sa ľahko programuje, ale zároveň vôbec nie je jednoduchá na hranie. Hrá sa na šachovnici  $8 \times 8$ . Hrajú dvaja hráči (biely a čierny) a na začiatku má každý hráč 16 kameňov umiestnených takto:



Hráči sa striedajú na ťahoch (začína biely), pričom v každom ťahu si hráč vyberie nejaký svoj kameň a pohnie ním. Cieľom je dostať niektorý svoj kameň na opačný koniec šachovnice. Komu sa to podarí prvému, vyhral (ak hráč pride o všetky svoje kamene, automaticky prehral). Hráč môže pohnúť svoj kameň o jedno poličko rovno alebo šikmo, ak je cieľové poličko voľné. Navyše, ak je na cieľovom poličku súperov kameň, hráč ho môže vyhodiť, ale iba šikmo (ako pešiak v šachu).



Príklad pravidiel: biely kameň ① môže ísť z b6 na a7, b7, alebo c7, lebo všetky sú voľné. Biely kameň ② z b2 môže ísť iba na a3, lebo b3 a c3 sú blokované. Biely kameň ③ z g6 môže ísť na f7, g7, alebo h7, pričom ak pôjde na f7 alebo h7 tak zoberie súperov kameň. Napokon kameň ④ z g2 môže buď zobrať súpera na f3, alebo prejsť na h3, ale nemôže ísť na g3.

<sup>1</sup>Pravidlá som prevzal z [https://trmph.com/bin/Basic\\_Introduction\\_to\\_Breakthrough.pdf](https://trmph.com/bin/Basic_Introduction_to_Breakthrough.pdf)

V tejto časti budem hovoriť o usporiadaní väčšieho projektu. Prostredia ako [Code::Blocks](#)<sup>1</sup>, či [MS Visual Studio](#)<sup>2</sup> majú spravidla možnosť vytvoriť projekt, ktorý sa skladá z viacerých súborov. Ja tu budem ukazovať príklady v prostredí linuxového terminálu: jednako dávajú väčšiu flexibilitu, nie sú závislé od konkrétneho prostredia a dá sa na nich najlepšie pochopiť, ako veci interne fungujú. Pre Windows sa dá nainštalovať napr. [MSYS2](#)<sup>3</sup>, ktorý poskytuje viac-menej rovnaké prostredie (terminál, príkazy, ...).

Čím väčší program píšeš (a niektoré programy môžu byť veľmi veľké), začína byť čoraz dôležitejšie udržovať si v programe poriadok. Ak dopredu vieš, že program, ktorý ideš písat, môže byť časom dlhší, je dobré si v ňom začať dodržovať poriadok už hneď od začiatku. V kapitole 20 sme hovorili o tom, že časti programu, ktoré spolu súvisia (napr. definíciu triedy a jej metód) je dobré dať do samostatného súboru a do hlavného programu ho pridať pomocou direktív `#include`. Dobré je aj dať si na začiatok takého súboru dlhší komentár, kde si napišeš názvy metód, čo robia ako ako sa používajú: keď budeš po čase potrebovať zistíť, ako sa tá trieda vlastne používa, nebudeš musieť hľadať hlboko v programe.

Toto ale nerieši iný problém, a tým je čas komplilácie. Isto si si všimol, že hlavne ak použiješ optimalizáciu (napr. prepínač `-O3`), program sa kompliluje pomerne dlho. Ak máš aj program rozdelený do viacerých súborov, preprocesor ich pred začiatkom komplilácie zlepí do jedného, takže sa vždy kompliluje všetko. Aby sme si ukázali, ako sa to dá zrýchliť, treba vedieť, že komplilátor vyrába program v dvoch fázach: najprv vyrobí tzv. *object code*, kde sú pre všetky funkcie vyrobené príkazy pre procesor, ale ešte nemajú presne určené adresy v pamäti. Object code potom spracuje tzv. *linker*, ktorý zoberie kusy z object code a zlepí ich do výsledného spustiteľného súboru. To, čo trvá dlho, je vyrobiť object code, následné linkovanie je krátke. Za normálnych okolností object code ani nevidiš, ale komplilátoru sa dá (pre `g++` je to prepínač `-c`) povedať, aby ho napísal do súboru a nelinkoval. Dajme tomu, že v súbore `main.cc` máš nasledovný program:

```

1 int cibulka = 4;
2 int lopata = 5;
3
4 int tulipany() {
5     int res = 1;
6     while (cibulka-- > 0) res <= 1;
7     return res;
8 }
9
10 int jama(int x) { return x * lopata; }
11
12 int main() { int x = jama(tulipany()); }
```

Kvôli jednoduchosti v ňom nepoužívam `cout`, takže po spustení len do premennej `x` priradi<sup>4</sup> 80 a skončí. Keď v termináli napišeš `g++ main.cc -o main` komplilátor vyrobí súbor `main`, ktorý môžeš spustiť `./main` (a nič sa nestane). Ak ale napišeš `g++ -c main.cc -o main.o` komplilátor vyrobí súbor `main.o`, ktorý obsahuje object code a spustiť nedá. Potom môžeš napísať `g++ main.o -o main` a object code z `main.o` sa zlinkuje do súboru `main`, ktorý sa už spustiť dá. Na súbor `main.o` sa dá pozrieť napr. príkazom `objdump -Ct main.o`, ktorý vypíše napr. niečo takéto:

<sup>1</sup><https://www.codeblocks.org/>

<sup>2</sup><https://visualstudio.microsoft.com>

<sup>3</sup><https://www.msys2.org/>

<sup>4</sup>Funkcia `tulipany` nastaví premennú `cibulka` na nulu a pritom ráta hodnotu  $2^{cibulka}$ . Keďže `cibulka` je na začiatku 4, tak `tulipany` vráti 16, čo následne `jama` vynásobí piatimi.

## Ako udržať poriadok vo veľkých projektoch

---

```
main.o:      file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 1    df *ABS*      0000000000000000 main.cc
0000000000000000 1    d .text       0000000000000000 .text
0000000000000000 g    0 .data       0000000000000004 cibulka
0000000000000004 g    0 .data       0000000000000004 lopata
0000000000000000 g    F .text       0000000000000002d tulipany()
0000000000000002d g   F .text       0000000000000013 jama(int)
0000000000000040 g   F .text       0000000000000001e main
```

Bez toho, aby sme to rozoberali do detailov vidno, že súbor má dva segmenty: `.data` a `.text` a v nich obsahuje všetky globálne premenné a skompilované funkcie uložené za sebou (číslo v prvom stĺpci je adresa). Náš nápad na zrýchlenie kompliacie by preto bol rozdeliť program do viacerých súborov, každý z nich prekomplilovať do object code a zlinkovať dokopy. Keď sa potom niečo zmení, bude stačiť prekomplilovať ten zmenený súbor a zlinkovať dokopy s ostatnými, nezmenenými, object súbormi (linkovanie je rýchle).

Spravme si preto súbor `tulipany.cc`, v ktorom bude:

```
1 int cibulka = 4;
2
3 int tulipany() {
4     int res = 1;
5     while (cibulka-- > 0) res <= 1;
6     return res;
7 }
```

a súbor `jama.cc`, v ktorom bude

```
1 int lopata = 5;
2
3 int jama(int x) {
4     return x * lopata;
5 }
```

Teraz môžem napísaf `g++ -c tulipany.cc -o tulipany.o` a vyrobí sa súbor `tulipany.o`, ktorý si viem skontrolovať `objdump -Ct tulipany.o`:

```
tulipany.o:      file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 1    df *ABS*      0000000000000000 tulipany.cc
0000000000000000 1    d .text       0000000000000000 .text
0000000000000000 g    0 .data       0000000000000004 cibulka
0000000000000000 g    F .text       0000000000000002d tulipany()
```

Rovnako to viem urobiť aj so súborom `jama.cc`. V súbore `main.cc` ale nechcem dávať `#include "tulipany.cc"`, to by som bol tam, kde na začiatku, takže mi v ňom ostane

```
1 int main() {
2     int x = jama(tulipany());
3 }
```

Ked teraz ale napíšem `g++ -c main.cc -o main.o`, kompilátor nevyrobí `main.o`, ale začne sa, celkom oprávnené, sťažovať, že jamu ani tulipány nepozná. Toto by bol inak dosť neprekonateľný problém, našťastie C++ má mechanizmus tzv. *deklarácií*. Čo to je? Ak napíšeš súbor `slon.cc`

```
1 int slon(int x) {
2     return x * 5;
3 }
4
5 int main() {
6     int z = slon(3);
7 }
```

je v názve *definícia* funkcie `slon`. Do programu ale môžeš napísať hlavičku funkcie bez tela (s bodkočiarkou na konci), čím hovoríš kompilátoru *Časom ti definujem takúto funkciu. Ak ju niekde treba použiť, bud' v pohode, je to pod kontrolou*. Tomuto sľubu sa hovorí *deklarácia*. Samozrejme, sľub treba dodržať a funkciu časom definovať, ale to sa prejaví až pri linkovaní. Zoberme súbor `slon.cc`

```
1 int slon(int x);
2
3 int main() {
4     int z = slon(3);
5 }
```

Ak napíšem `g++ -c slon.cc -o slon.o`, kompilátor vytvorí súbor `slon.o`, v ktorom je

```
slon.o:      file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 1    df *ABS*          0000000000000000 slon.cc
0000000000000000 1    d   .text          0000000000000000 .text
0000000000000000 g    F   .text          0000000000000001c main
0000000000000000             *UND*          0000000000000000 slon(int)
```

Všimni si **\*UND\*** pri symbolu `slon(int)`: kompilátor ti uveril, že `slon` nakoniec niekde bude, a do object code si zaznačil, že zatiaľ ho nenašiel, ale nerobí kvôli tomu paniku. Keby si chcel súbor (aj) zlinkovať (napr. `g++ slon.o -o slon`), vypíše sa chyba:

```
/usr/bin/ld: /tmp/cc4uVwt3.o: in function 'main':
slon.cc:(.text+0xe): undefined reference to 'slon(int)'
collect2: error: ld returned 1 exit status
```

Teraz linker hovorí, že chcel zlepíť dokopy výsledného súboru, ale slona v žiadnom vstupnom súbore nenašiel. Definičiu slona môžeš dodať kamkoľvek, napr. ak spravíš súbor `slon.cc` takto:

```
1 int slon(int x); // ok, deklarácia
2 int slon(int x); // druhá deklarácia, nie je problém ak je rovnaká
3
4 int main() {
5     int z = slon(3);
6 }
7
8 int slon(int x) { // definícia
9     return x * 2;
10 }
```

## Ako udržať poriadok vo veľkých projektoch

---

Všetko bude v poriadku. Záver z tohto pozorovania je teda takýto:

1. funkcia môže mať veľa deklarácií (musia byť rovnaké)
2. funkcia musí mať práve jednu definíciu (inak sa vyrobí object code, ale nezlinkuje sa)
3. použitiu funkcie v programe musí predchádzať definícia alebo deklarácia

Naspäť k príkladu s jamou a tulipánom (zmaž všetky súbory `.o`). Máme súbory `jama.cc`, `tulipany.cc` a `main.cc`. Ten zmeníme tak, že pridáme deklarácie príslušných funkcií:

```
1 int jama(int);
2 int tulipany();
3
4 int main() {
5     int x = jama(tulipany());
6 }
```

Teraz treba napísť príkazy:

```
g++ -c jama.cc -o jama.o
g++ -c tulipany.cc -o tulipany.o
g++ -c main.cc -o main.o
g++ main.o jama.o tulipany.o -o main
```

a všetko prejde, ako má. Čo ak by som ale chcel v hlavnom programe modifikovať premennú `lopata` (ktorá je v súbore `jama.cc`)? Nemôžem napísť:

```
1 int jama(int);
2 int tulipany();
3
4 int main() {
5     lopata = 10;
6     int x = jama(tulipany());
7 }
```

lebo pri komplilovaní `main.cc` a vyhlási chybu, že nepozná premennú `lopata`. Nemôžem napísť ani

```
1 int jama(int);
2 int tulipany();
3
4 int lopata;
5
6 int main() {
7     lopata = 10;
8     int x = jama(tulipany());
9 }
```

Tým totiž vyrábim (inú) premennú `lopata` v súbore `main.o`. Súbor `main.cc` sa skompiluje, ale linker sa bude sťažovať na `multiple definition of 'lopata'`. Analógiu deklarácie funkcie je slovíčko `extern` pre premennú: hovorí komplilátoru aby pre ňu nerobil novú adresu, ale spoľahlol sa na to, že časom taká premenná bude definovaná. Výsledný súbor `main.cc` by teda mohol vyzerať takto:

```

1 // odkazy na súbor jama.cc
2 extern int lopata;
3 int jama(int);
4
5 // odkazy na súbor tulipany.cc
6 extern int cibulka;
7 int tulipany();
8
9 // hlavný program
10 int main() {
11     lopata = 10;
12     cibulka = 7;
13     int x = jama(tulipany());
14 }
```

Aby sa udržal poriadok, je zvykom deklarácie z nejakého súboru (napr. [jama.cc](#)) dať do rovnako sa volajúceho súboru s príponou [.h](#) ako *header* (napr. [jama.h](#)) a ten potom vložiť pomocou `#include` všade, kde ho je treba. Takže do súboru [jama.h](#) napišem

```

1 extern int lopata;
2 int jama(int);
```

do súboru [tulipany.h](#) napišem

```

1 extern int cibulka;
2 int tulipany();
```

a výsledný program bude

```

1 #include "jama.h"
2 #include "tulipany.h"
3
4 int main() {
5     lopata = 10;
6     cibulka = 7;
7     int x = jama(tulipany());
8 }
```

Metódy tried sú funkcie ako každé iné, a preto pre ne platia rovnaké pravidlá o deklaráciách a definíciah, ako pre normálne funkcie. Treba len myslieť na to, že ak sa na metódu odvolávame mimo definície jej triedy, treba ju volať jej celým menom, t.j. pred menom metódy treba vždy uviesť (s dvomi dvojbodkami) triedu, ktorej patrí. Porovnaj si tieto dva zápisy

```

1 struct Bod {
2     int x, y;
3     int sucet() { return x + y; }
4 };
```

```

1 struct Bod {
2     int x, y;
3     int sucet();
4 };
5
6 int Bod::sucet() { return x + y; }
```

Doteraz sme všetky metódy tried písali tak, ako na príklade vľavo. Oba zápisy robia to isté, ale je tam drobný rozdiel, ktorý som ti doteraz nespomienul: zápis vľavo považuje metódu `sucet` za tzv. *inline*: komplilátor ju neskomplikuje priamo, ale vždy, keď sa v programe vyskytne jej volanie, komplilátor ho nahradí programom z

## Ako udržať poriadok vo veľkých projektoch

---

definície<sup>5</sup>. Preto zápis vľavo môže byť v header súbore, ale zápis vpravo musíme riešiť rovnako, ako pri ostatných funkciách.

Zvykom je v definícii typu (tryedy) napísaf inline iba krátke malé metódy, pri ktorých nevadí, že sa v skompliovanom programe objavia veľakrát. Pri ostatných metódach napísat v headri iba ich deklarácie a ich definície písat neskôr.

Skúsme si to ukázať na trochu zložitejšom príklade. Vyrob si tri súbory. Prvý bude `vec.h` a bude v ňom

```
1 #ifndef __vec_h__
2 #define __vec_h__
3 struct Vec {
4     struct Proxy {           // tento typ sa celým menom volá Vec::Proxy
5         Proxy(Vec *a);      // konštruktor Vec::Proxy (deklarácia)
6         Proxy &operator++(); // zvýši hodnotu x a vráti referenciu na seba (deklarácia)
7     private:
8         int *x;              // x je privátne pre Vec::Proxy
9     };
10    Vec(int x);            // konštruktor Vec (deklarácia)
11    int &operator()();     // vráti referenciu na mojaVec (deklarácia)
12    Proxy vyrobProxy();   // vytvorí Vec::Proxy, ktorého x ukazuje na mojaVec (deklarácia)
13    private:
14        int mojaVec;        // privátnej hodnota pre Vec
15    };
16 #endif
```

Je v ňom definovaný typ `Vec`, ktorý má privátnu premennú `mojaVec`, má konštruktor s parametrom `int`, `operator()()` (t.j. operátor `()` bez parametrov) a metódu `vyrobProxy`. Všetky tri sú iba deklarované, definície budú v inom súbore. Okrem toho má typ `Vec` svoj vlastný typ `Proxy` (podobne ako iterátory v STL), ktorý sa celým menom volá `Vec::Proxy`. Ten má svoj vlastný konštruktor a (prefixový) operátor `++`. Tieto sú tiež v súbore `vec.h` iba deklarované a bude treba ich definovať. Definície budú v súbore `vec.cc`:

```
1 #include "vec.h" // include treba, aby kompilátor vedel o typoch Vec a Proxy
2                                     // ked' vyrába object code pre vec.cc
3 Vec::Proxy::Proxy(Vec *a) { x = &a->mojaVec; } // konštruktor pre Vec::Proxy
4                                         // x je pointer na a-čkovu mojaVec
5 Vec::Vec(int x) { mojaVec = x; } // konštruktor pre Vec
6 int &Vec::operator()() { return mojaVec; }
7
8 // zavolaj konštruktor Proxy a daj mu ako parameter pointer na seba
9 Vec::Proxy Vec::vyrobProxy() { return Proxy(this); }
10
11 Vec::Proxy &Vec::Proxy::operator++() {
12     (*x)++;
13     return *this;
14 }
```

Zápis `Vec::Proxy::Proxy(Vec *a)` treba chápať takto: `Vec` je typ, v ňom je typ `Vec::Proxy`. Hocijaká funkcia `fun`, ktorá by patrila typu `Vec::Proxy` sa preto celým menom volá `Vec::Proxy::fun`. Konštruktor je metóda, ktorá sa volá rovnako, ako typ (v našom prípade `Proxy`). Preto celé meno konštruktora je `Vec::Proxy::Proxy`. A napokon hlavný súbor `main.cc`:

```
1 #include "vec.h"
2 #include <iostream>
3 using namespace std;
```

<sup>5</sup>v niektorých prípadoch je to rýchlejšie, ale rastie veľkosť skompliovaného programu

```

4 int main() {
5     Vec v(42);           // konštruktor spraví Vec v, kde v.mojaVec bude 42
6     cout << v() << endl; // v() vráti referenciu na v.mojaVec, vypíše sa 42
7     v() += 5;            // v() je referencia, takže viem v.mojaVec meniť,
8                     // aj keď mojaVec je private
9     cout << v() << endl; // vypíše sa 47
10    auto p = v.vyrobProxy(); // p je Vec::Proxy, jeho x je pointer na v.mojaVec
11    ++++p;               // +++p inkrementuje v.mojaVec a vráti referenciu na seba
12                     // pretoo ++++p je ++(++p)
13    cout << v() << endl; // vypíše sa 49
14 }

```

Teraz môžeš napísť

```

g++ -c vec.cc -o vec.o
g++ main.cc vec.o -o main

```

a po spustení `./main` by sa mali vypísať čísla 42, 47 a 49 (všimni si, že pri jednom volaní komplilátora môžem zadať viacero vstupných súborov, ktoré môžu byť zmes programov a object code súborov).

Máme hotový mechanizmus, ako udržovať v programe poriadok a zároveň obmedziť čas komplilácie: časti programu, ktoré logicky patria k sebe (napr. niekoľko tried, ktoré robia jadnu vec a je šanca, že ich použijeme aj inokedy), dáme dokopy: typy a deklarácie idú do súboru `.h` a príslušné definície do súboru `.cc`. Súbor `.cc` skomplilujeme do object code, súbor `.h` vložíme pomocou `#include` kam treba. Ak sa niečo zmení, prekomplilujeme iba príslušnú časť a všetko zlinkujeme dokopy. Jediná nepraktická vec je, že na vyrobenie programu musíme veľakrát spúštať komplilátor s rôznymi parametrami, a musíme si pamätať, čo sa zmenilo a čo treba prekomplilovať. Ako toto celé riešiť automaticky si ukážeme o chvíľu, predtým jedno dôležité upozornenie.

Dôležité upozornenie: celý tento mechanizmus, ktorý sme si ukázali, nefunguje pre šablóny. Opäť si to ukážeme na príklade. Zober si nasledovný súbor, v ktorom je trieda `Dvojica`, ktorá má dve premenné nejakého typu a vie ich vymeniť:

```

1 template <typename T>
2 struct Dvojica {
3     T x, y;
4     void vymen();
5 };
6
7 template <typename T>
8 void Dvojica<T>::vymen() {
9     T tmp = x;
10    x = y;
11    y = tmp;
12 }

```

```

1 #include <iostream>
2 using namespace std;
3 int main() {
4     Dvojica<int> d;
5     d.x = 3;
6     d.y = 4;
7     d.vymen();
8     cout << d.x << " " << d.y << endl;
9 }

```

V prvom rade si všimni, akým spôsobom zapisujeme metódu `vymen`: nestačí písať `void Dvojica::vymen()`, lebo `Dvojica` sama osebe nie je typ. Aj pred definíciou treba pridať príslušnú šablónu, aby sme mohli napísat celý typ, ktorý je `Dvojica<T>`. Kým je celý program v jednom súbore, všetko funguje. Skús ho teraz rozdeliť na tri súbory, ako sme to robili pred chvíľou: šablóna typu `Dvojica` pôjde do súboru `dvojica.h`, definícia metódy `vymen` do súboru `dvojica.cc` a zvyšok ostane v `main.cc` (s tým, že `main.cc` aj `dvojica.cc` budú mať `#include "dvojica.h"`). Keď skúsiš súbory skomplilovať tak, ako doteraz, t.j.

## Ako udržať poriadok vo veľkých projektoch

---

```
g++ -c dvojica.cc -o dvojica.o
g++ -c main.cc -o main.o
g++ dvojica.o main.o -o main
```

linker sa zrazu bude sťažovať, že

```
/usr/bin/ld: main.o: in function 'main':
main.cc:(.text+0x1e): undefined reference to 'Dvojica<int>::vymen()'
collect2: error: ld returned 1 exit status
```

Čo sa stalo? Odpoved' dá `objdump -Ct dvojica.o`:

```
dvojica.o:      file format elf64-x86-64

SYMBOL TABLE:
0000000000000000 1    df *ABS*          0000000000000000 dvojica.cc
```

Súbor `dvojica.o` neobsahuje žiadne funkcie. Prečo? V súbore `dvojica.cc` nie je zapísaná žiadna funkcia, iba šablóna, ako príslušnú funkciu vyrobí, keď ju bude treba. Ale nie je tam žiadne miesto, kde by bolo treba vytvoriť funkciu `Dvojica<int>::vymen()`, takže sa nevytvorí. Bolo by ju treba až v súbore `main.o`, ale to zistí až linker, keď je už neskoro. Keby si do súboru `dvojica.cc` pridal funkciu, ktorá prinúti kompilátor šablónu použiť, napr.:

```
1 #include "dvojica.h"
2 template <typename T>
3 void Dvojica<T>::vymen() {
4     T tmp = x;
5     x = y;
6     y = tmp;
7 }
8
9 void hlupaFunkcia() {
10     Dvojica<int> d;
11     d.vymen();
12 }
```

všetko by už fungovalo (skontroluj si `objdump -Ct dvojica.o`). Samozrejme, takéto riešenie je zlé, preto pri šablónach je zvykom aj ich definície<sup>6</sup> písat do súboru `.h`.

<sup>6</sup>Treba si ale dať pozor pri špecializácii. Keď máš napr. šablónu

```
1 template <typename T>
2 struct Vec { void daj(); };
```

tak definícia jej metódy

```
1 template <typename T>
2 void Vec<T>::daj() { ... }
```

je šablóna a vyrobí sa, až keď je jasné, s akým typom `T` je treba. Preto patrí do súboru `.h`. Ale keď tú šablónu plne špecializuješ

```
1 template <>
2 struct Vec<int> { void daj(); };
```

tak jej metóda (všimni si, že nepíšem prázdné `template<>`: toto nie je špecializácia metódy, ale metóda špecializovanej triedy)

Tak a teraz sa môžeme dostať k druhej časti tejto kapitoly, a to ako automatizovať, ktoré súbory sa kedy majú prekompilovať. To sa veľmi dobre dá pomocou programu `make`. `make` je program, ktorý vie vykonať nejaké akcie (napr. zavolať komplilátor), ak sú splnené nejaké podmienky (napr. súbor treba prekompilovať, lebo sa zmenil). Väčšinou sa `make` používa na komplilovanie programov, ale dá akcie môžu byť ľubovoľné<sup>7</sup>. Je to veľmi flexibilný a silný nástroj, tu si z neho ukážeme iba zopár vecí, ktoré budeme používať.

Ak spustíš `make` bez parametrov, bude hľadať v aktuálnom adresári súbor `Makefile` a z neho bude čítať pravidlá. Pravidlo má tvar

```
ciel': prerekvizity
    akcia
    ...
```

V najjednoduchšej forme je `ciel` názov súboru, ktorý treba vyrobiť, ak sa niektorý zo súborov uvedených v časti `prerekvizity` zmení. `akcia` hovorí, čo treba urobiť na vyrobenie cieľa. Akcia môže mať viacero riadkov, ale každý musí začínať tabulátorom. Zoberme si opäť príklad so súbormi `main.cc`, `jama.cc`, `jama.h`, `tulipany.cc`, `tulipany.h`. Najjednoduchší `Makefile` by vyzeral takto:

```
main: main.o jama.o tulipany.o
    g++ main.o jama.o tulipany.o -o main

jama.o: jama.cc jama.h
    g++ -c jama.cc -o jama.o

tulipany.o: tulipany.cc tulipany.h
    g++ -c tulipany.cc -o tulipany.o

main.o: main.cc tulipany.h jama.h
    g++ -c main.cc -o main.o
```

Ak spustíš `make` bez parametrov, zoberie si za hlavný cieľ prvý v poradí. Konkrétny cieľ môže dať ako parameter pri spustení, takže v našom prípade je `make` a `make main` to isté. `make` zistí, že na vyrobenie súboru `main` treba mať aktuálne súbory `main.o` a `tulipany.o`. Najprv ide skontrolovať prerekvizity. Pozrie sa, či je aktuálny súbor `main.o`. Zistí, že taký súbor neexistuje, ale existuje cieľ, ktorým sa dá vyrobiť. Prerekvizity cieľa `main.o` sú `main.cc`, `tulipany.h` a `jama.h`, ktoré všetky existujú, takže sa vykoná akcia `g++ -c main.cc -o main.o` a súbor `main.o` je v poriadku. Podobne sa prejdú ciele `jama.o` a `tulipany.o` a vyrobia sa príslušné súbory. Keď sú prerekvizity cieľa `main` v poriadku, vykoná sa akcia `g++ main.o jama.o tulipany.o -o main` a hotovo (vyskúšaj si to). Ak napíšeš `make` znova, znova sa prečíta `Makefile` a skontrolujú sa ciele. Teraz všetky súbory existujú, preto `make` iba skontroluje čas modifikácie<sup>8</sup>. Postupne zistí, že `main.o` bol modifikovaný neskôr ako `main.cc`, `tulipany.h` aj `jama.h`, takže akciu pre aktualizáciu cieľa `main.o` netreba robiť. Rovnako to dopadne aj pre ostatné ciele, nakoniec sa zistí, že aj súbor `main` je aktuálny, a netreba nič komplilovať. Ak trez zmeníš napr. súbor `jama.cc` a spustíš `make`, ciele `tulipany.o` aj `main.o` budú aktuálne, takže sa iba prekompiluje `jama.o` a všetko sa zlinkuje dokopy. Ak to zhrnieme: pravidlo v `Makefile` hovorí, že ak sa zmení niektorá z prerekvizít, treba aktualizovať cieľ pomocou príslušnej akcie.

Ciele sa chápú ako názvy súborov, ale dá sa to využiť aj inak. Na koniec `Makefile` si dopis

```
1 void Vec<int>::daj() { ... }
```

je už normálna funkcia ako každá iná a komplilátor ju vyrobí, len čo ju zbadá. Ak ju dás do súboru `.h`, tak sa komplilátor môže stažovať na viacnásobnú definíciu.

<sup>7</sup> `make` napríklad používam aj pri generovaní tohto textu

<sup>8</sup> Operačný systém si pri každom súbore vedie rôzne štatistiky, okrem iného aj čas poslednej modifikácie. `make` podľa toho vie rozpoznať, ktorý súbor sa zmenil.

## Ako udržať poriadok vo veľkých projektoch

---

```
clean:  
  rm -f *.o  
  rm -f main
```

Čo sa stane, ak zavoláš `make clean`? Skontroluje sa cieľ `clean` a keďže taký súbor neexistuje, `make` sa ho pokúsi vytvoriť príslušnou akciou (prerekvizity žiadne nie sú, takže sa nič nekontroluje). Lenže tá akcia nevytvorí súbor `clean`, iba zmaže všetky vygenerované súbory. `make clean` sa teraz dá použiť na upratovanie: kedykoľvek ho napíšeš, upracú sa zbytočné súbory. Toto je fajn, ale ak by sa v adresári nejak ocitol súbor, ktorý by sa volal `clean`, prestane to fungovať. `make` má pre tento účel špeciálny cieľ `PHONY`, ktorý hovorí, že všetky jeho prerekvizity treba aktualizovať, aj keby existovali. Preto sa zvykne písat

```
.PHONY: clean  
clean:  
  rm -f *.o  
  rm -f main
```

Ak máš projekt, v ktorom je veľa súborov, začne byť ručné písanie pravidiel pre každý súbor neprehľadné. `make` má preto možnosť<sup>9</sup> ako veci automatizovať. Jendou z nich sú premenné: premenná v `Makefile` môže obsahovať postupnosť reťazcov. Premenná sa vyrobí tak, že sa do nej priradí `meno = hodnota` a pristupuje sa knej cez znak `$`, t.j. `$(meno)`. Napr.

```
comp = g++ -O3  
sources = $(wildcard *.cc)  
objects = $(sources:.cc=.o)  
  
main: $(objects)  
  $(comp) $(objects) -o main
```

vyrobí premennú `comp`, v ktorej bude "`g++ -O3`", premennú `sources`, v ktorej bude výsledok príkazu `wildcard *.cc`<sup>10</sup>, t.j. "`main.cc tulipany.cc jama.cc`" a premennú `objects`, ktorá obsahuje<sup>11</sup> všetky reťazce z `$(sources)`, ale koncovka ".cc" sa nahradí ".o", t.j. "`main.o tulipany.o jama.o`". Cieľ pre `main` teda funguje rovnako, ale keď sa pridá ďalší súbor, netreba meniť `Makefile`. Ostatné pravidlá vyzerajú všetky podobne: na výrobenie súboru `.o` treba skompilovať súbor `.c`. V `Makefile` sa na to dajú použiť tzv. *pattern rules*, t.j. šablóny pravidiel, v ktorých vystupuje znak `%`: pri hľadaní vhodného pravidla sa zaňho môže dosadiť čokoľvek. Napr. `Makefile` z predchádzajúceho rámčeka by sme mohli doplniť

```
comp = g++ -O3  
sources = $(wildcard *.cc)  
objects = $(sources:.cc=.o)  
  
main: $(objects)  
  $(comp) $(objects) -o main  
  
%.o: %.cc  
  $(comp) -c $< -o $@
```

<sup>9</sup>Tých možností je ozaj veľa, odporúčam pozrieť pozriet manuál: [www.gnu.org/software/make/manual/make.html](http://www.gnu.org/software/make/manual/make.html)

<sup>10</sup>To je vstavaný príkaz v `make`, ktorý vráti zoznam všetkých súborov v aktuálnom adresári, ktorých mená sa končia na `.cc` (hviezdička tradične známená *hocíčko*)

<sup>11</sup>opäť špeciálny zápis, ktorý `make` poskytuje

Posledné pravidlo hovorí, že ak treba vyrobiť hocjaký súbor `kvik.o`, treba skontrolovať prerekvizitu `kvik.cc` a ak treba, skompilovať ho. Premenná `$<` je špeciálna premenná, ktorá označuje prvú prerekvicuzu (napr. `kvik.cc`) a `$@` označuje cieľ, t.j. `kvik.o`. Toto bude fungovať, až na to, že sa neberú do úvahy súbory `.h`. Ak napr. zmeníš `jama.cc`, správne sa prekompliuje `jama.o` a zlinkuje sa výsledok, ale ak zmeníš `jama.h`, nestane sa nič. Toto je hlbší problém, pretože by som chcel pre každý súbor `.cc` zistíť, ktoré súbory si vkladá pomocou `#include`, a dať príslušné prerekvizity. Práve kvôli tomuto má `g++` prepínač `-MM`, ktorý vypíše závislé súbory. Ak napíšeš `g++ -MM main.cc`, vypíše sa

```
main.o: main.cc jama.h tulipany.h
```

teda presne to, čo by som chcel mať v hlavičke pravidla v `Makefile`. To viem použiť spolu s mechanizmom `include` v `Makefile` na to, aby som spravil niečo takéto

```
sources = $(wildcard *.cc)
objects = $(sources:.cc=.o)
depends = $(sources:.cc=.d)

main: $(objects)
    g++ $(objects) -o $@

%.d: %.cc
    g++ -MM $< -o $@

include $(depends)

%.o: %.cc
    g++ -c $< -o $@
```

Podme sa pozrieť, čo tieto pravidlá hovoria. Okrem toho, čo sme tu už mali, si pre každý súbor chceme vyrobíť súbor `.d` (ich názvy mám v premennej `depends`). Mám pravidlo, ktoré mi hovorí, ako vyrobiť súbor `.d` zo súboru `.cc`. Riadok `include $(depends)` znamená, že na toto miesto treba vložiť obsah súborov `main.d`, `jama.d` a `tulipany.d`. Ak tie súbory neexistujú, vyrobia sa podľa dostupého pravidla. Keď existujú, vložia sa tam a budú dávať dodatočné prerekvizity pre jednotlivé `.o` súbory<sup>12</sup> Skús si tento `Makefile` vyskúšať tak, že budeš meniť rôzne veci v rôznych súboroch a sledovať, kedy sa čo prekompliuje. Je tam ešte jeden drobný problém. Nájdeš ho?

Ak si ho našiel, gratulujem. Problém je v tom, že súbor `.d` závisí iba od príslušného súboru `.cc`. Dá sa to vidieť na príklade. Výrob si nejaký súbor `krtko.h`. Po tom, čo spustíš `make`, pridaj do `jama.h` riadok `#include "krtko.h"`. Zavoláš `make` a výsledný súbor sa prekompliuje, ale súbor `jama.d` sa nezmení. Keď teraz zmeníš súbor `krtko.h`, nič sa neprekompiluje<sup>13</sup>. Aby sme to opravili, pravidlo pre generovanie súborov `.d` upravíme tak, aby pridal závislosti aj pre samotný súbor `.d`, takže napr. namiesto riadku `jama.o: jama.cc` `jama.h` bude `jama.d jama.o: jama.cc jama.h`. A úplne posledná vec: je prehľadnejšie, ak sa generované súbory nemiešajú s tými, ktoré píšeme. Takže si spravíme adresár (napr. `build`) a všetky dočasné súbory budeme umiestňovať tam. Výsledný `Makefile` by preto mohol vyzerať takto:

<sup>12</sup>Po vložení bude mať napr. súbor `main.o` dve pravidlá: jedno, ktoré vzniklo vložením súboru `main.d`, a kde má prerekvizity `main.cc jama.h tulipany.h`, ale žiadnu akciu a druhé, kde sa použije `%.o: %.cc`. Pri spracovaní `make` funguje tak, že pri viacerých pravidlach pre jeden cieľ vyžaduje, aby práve jedno malo nastavenú akciu. Potom pozbiera všetky prerekvizity do jednej a ak treba, akciu urobí.

<sup>13</sup>Všimni si, že keby si riadok `#include "krtko.h"` pridal do `jama.cc`, prerobi sa aj súbor `jama.d` a všetko bude v poriadku.

```
sources  = $(wildcard *.cc)
builddir = ./build
objects  = $(addprefix $(builddir)/,$(sources:.cc=.o))
depends   = $(objects:.o=.d)

main: $(objects)
    mkdir -p $(builddir)
    g++ $(objects) -o $@

$(builddir)/%.d: %.cc
    mkdir -p $(builddir)
    echo -n $@ " $(builddir)/> $@
    g++ -MM $< >> $@

include $(depends)

$(builddir)/%.o: %.cc
    mkdir -p $(builddir)
    g++ -c $< -o $@

PHONY.: clean
clean:
    rm -rf $(builddir)
```

Príkaz `mkdir -p` urobí nový adresár, ale iba ak ešte neexistuje. Príkaz `echo -m oznam` vypíše `oznam` (-n hovorí, aby na koniec nedal endl). Presmerovanie `>` znamená *Pripoj výstup z programu na koniec súboru*.

Po odbočke k organizácii projektu sa vráime k hre Breakthrough a urobme si prípravné práce. Poličko budeme mať zapamätané ako dvojicu *stĺpec* (*file*, *f*) a *riadok* (*rank*, *r*). A ľah si budeme pamätať ako dvojicu poličok *odkial*, *kam*. To nám dáva prirodzené triedy

```
1 struct Square {
2     uint8_t f, r;
3     bool legal();
4 };
5
6 struct Move {
7     Square from, to;
8 };
```

s tým, že do `Square` som pridal metódu `legal`, ktorá skontroluje rozsah (od 0 do 7). Premenné `f`, `r` som definoval s typom `uint8_t`. S takým typom sme sa ešte nestretli, ale je to jeden z číselných typov. Okrem typov ako `int`, `unsigned long` a pod., ktorých veľkosť závisí od systému, sú aj názvy typov s fixnou dĺžkou. Takže `uint8_t` znamená 8-bitový `unsigned int`, t.j. typ, ktorý vždy zaberá 1 byte a dajú sa v ňom pamätať čísla 0...255. Podobne `int32_t` je 32-bitový `int` so znamienkom, `uint64_t` 64-bitový `unsigned int` a pod. Tieto typy sú definované v knižnici `<cstdint>`, takže keď ich chceš používať, treba zavolať `#include <cstdint>`.

Šachovnicu si najjednoduchšie môžeme pamätať ako pole  $8 \times 8$  poličok, kde 0 znamená biely, 1 čierny a 2 prázdne poličko. Ale zhodou okolností šachovnica má presne 64 poličok, tak sa podľame pocvičiť v bitových operáciach: budeme mať bitovú mapu pre biele a čierne kamene, `uint64_t pmap[2]` (`pmap[0]` pre biele a `pmap[1]` pre čierne). Obidve sú 64-bitové čísla, pričom príslušný bit je nastavený na 1, ak je na danom poličku kameň tej správnej farby. Ak máme poličko na stĺpci (*file*) *f* a riadku (*rank*) *r* (obidve v rozsahu 0 až 7), tak číslo

bitu môže byť  $(f \ll 3) | r^{14}$ . Môžem si spraviť funkciu `idx(f, r)`, ktorá vráti 64-bitové číslo, ktoré má nastavený iba bit zodpovedajúci poličku `(f, r)`, t.j.  $2^{8 \cdot f + r}$ , resp. `((uint64_t)1 << ((uint8_t)((f << 3) | r))`.

Od triedy `board` by sme nateraz čakali niečo takéto:

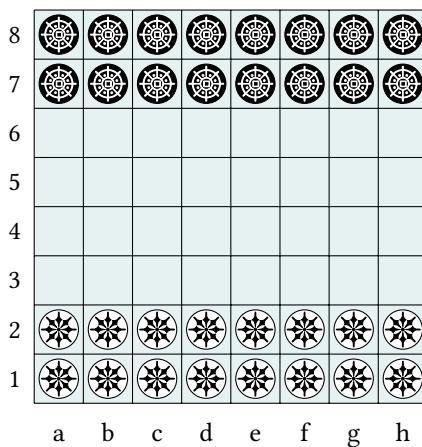
```

1 const uint8_t White = 0;
2 const uint8_t Black = 1;
3 const uint8_t Empty = 2;
4
5 struct Board {
6     Board(); // konštruktor - nastaví začiatočnú pozíciu
7
8     uint64_t idx(int f, int r) const; // bit v mape zodpovedajúci poličku
9     uint8_t get(int f, int r) const; // kto má kameň na poličku
10    void set(int f, int r, uint8_t val); // nastav poličko
11
12    int winner() const; // má už pozícia víťaza?
13    bool legal(Move m) const; // je m prípustný ľah?
14    std::vector<Move> legalMoves() const; // vráť všetky prípustné ľahy
15    Board &operator+=(Move m); // urob ľah m
16
17    int ply; // číslo polťahu
18    uint64_t pmap[2]; // mapa kameňov bieleho a čierneho
19 };

```

Okrem mapy šachovnice (`pmap`) si chceme pamätať *polťah* (`ply`): po každom potiahnutí sa zvýší, takže ak je na ťahu biely, je `ply` párne, ak je na ťahu čierny, je nepárne. Pravidlá hry sú schované v metódach `winner`, a `legal`: `winner` má vrátiť `White`, `Black`, alebo `Empty`, podľa toho, či je víťaz alebo nie (vtedy vráti `Empty`; všimni si, že remíza nemôže nastať). Metóda `legal` má povedať, či je daný ťah v danej pozícii prípustný. Na začiatok sa budú hodí ešte dve metódy: `legalMoves` vráti zoznam všetkých prípustných ťahov z danej pozície a `operator+=` urobí ťah.

Konštruktor má nastaviť hrací plán na začiatok hry, t.j.



To môžem urobiť buď tak, že veľkrát zavolám `set`, ale môžem napísať aj `pmap[0]=0x303030303030303ULL`; `pmap[1]=0xc0c0c0c0c0c0c0c0ULL`. Čo je toto za zápis? `0x` na začiatku čísla znamená, že je to číslo v šestnásťkovnej sústave<sup>15</sup>. Prípona `ULL` na konci znamená, že je to `unsigned long long`. Ako sme vraveli v poznámke

<sup>14</sup>hodnotu `f` posuniem o tri bity dolava a na uvolnené tri bity dám hodnotu `r`, čo je to isté, ako  $8 \cdot f + r$ , pretože `r` je nanajvýš 7

<sup>15</sup>O 16-kovej sústave pozri poznámku na str. 14; za chýbajúce cifry používame `a = 10, b = 11, c = 12, d = 13, e = 14` a `f = 15`, na veľkosťi nezáleží, takže `e` aj `E` je 14

## Ako udržať poriadok vo veľkých projektoch

---

pri bitových operáciach na str. 23, prevod šestnásťkovej sústavy do dvojkovej je príjemný v tom, že jedna šestnásťková cifra sú presne 4 dvojkové, takže viem prevádzkať cifru po cifre: `0x03` je v dvojkovej `00000011`, `0x0303` je `0000001100000011` atď. Podobne `0xc0` je `11000000`, `0xc0c0` je `1100000011000000` atď.

Teraz ešte môžeme pridať funkcie

```
1 std::ostream &operator<<(std::ostream &, const Square &);  
2 std::ostream &operator<<(std::ostream &, const Move &);  
3 std::ostream &operator<<(std::ostream &, const Board &);
```

aby sa nám veci pekne vypisovali. Všimni si, že v deklaráciách stačí pri parametroch napísť typ, netreba písť aj meno parametra (to treba až vtedy, keď sa ten parameter má použiť v tele funkcie).

**Úloha 124.** *Priprav si projekt s `Makefile`, súborom `board.h`, v ktorom budú definície, čo sme tu napísali a súborom `board.cc`, v ktorom budú príslušné definície funkcií. Urob jednoduchý `main.cc`, ktorý vypíše pozíciu zo strany 190 a pod ňu všetky možné tahi bieleho:*

```
    a   b   c   d   e   f   g   h  
+---+---+---+---+---+---+---+---+  
8 |   |   |   |   |   |   |   | 8  
+---+---+---+---+---+---+---+---+  
7 |   |   |   |   | 0 |   | 0 | 7  
+---+---+---+---+---+---+---+---+  
6 |   | X |   |   |   | X |   | 6  
+---+---+---+---+---+---+---+---+  
5 |   |   |   |   |   |   |   | 5  
+---+---+---+---+---+---+---+---+  
4 |   |   |   |   |   |   |   | 4  
+---+---+---+---+---+---+---+---+  
3 |   | X | X |   |   | 0 | 0 | 3  
+---+---+---+---+---+---+---+---+  
2 |   | X |   |   |   | X |   | 2  
+---+---+---+---+---+---+---+---+  
1 |   |   |   |   |   |   |   | 1  
+---+---+---+---+---+---+---+---+  
    a   b   c   d   e   f   g   h  
  
b2-a3  b3-a4  b3-b4  b3-c4  b6-a7  
b6-b7  b6-c7  c3-b4  c3-c4  c3-d4  
g2-f3  g2-h3  g6-f7  g6-g7  g6-h7
```

Cieľom tejto časti je naprogramovať hráča, ktorý by hral Breakthrough. Začneme skromne:

**Úloha 125.** Pokračuj v projekte z úlohy 124. Do súborov `board.h`/`board.cc` pridaj načítanie ďahu zo vstupu (načíta sa reťazec napr. "c5-d6").

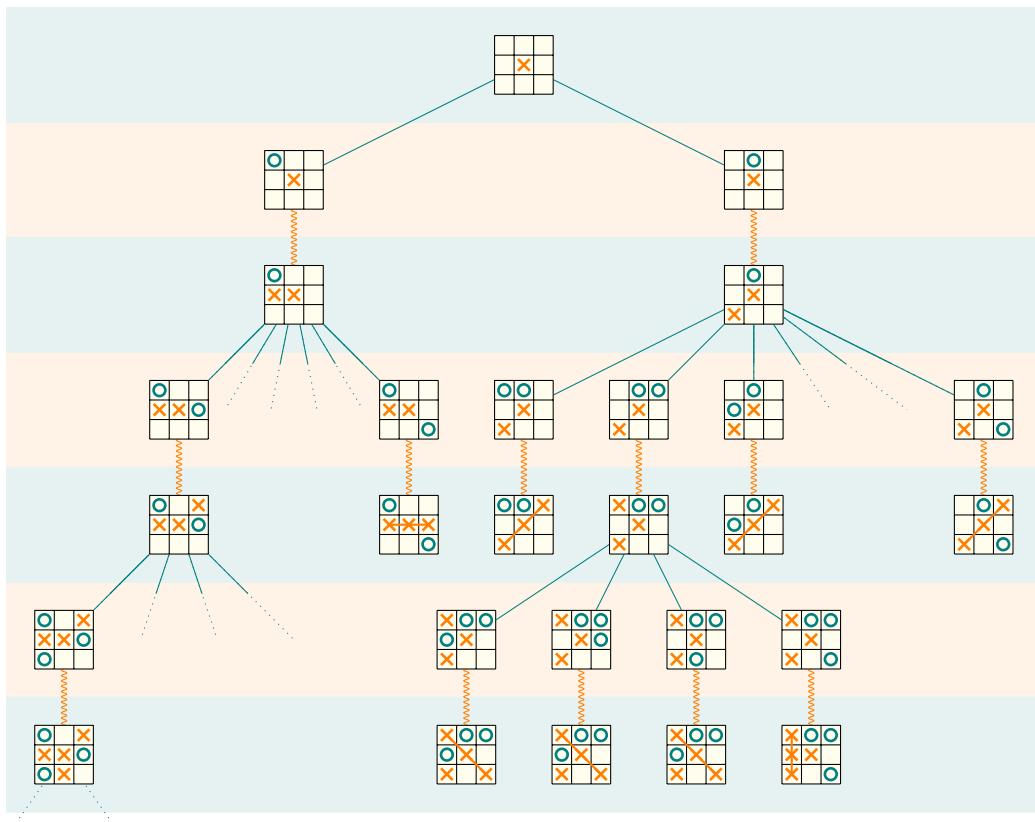
```
1 std::istream& operator>> (std::istream &, Move &);
```

Pridaj rovnaké načítanie aj pre `Square a Board` (to môže byť zoznam políčok bieleho a zoznam políčok čierneho, oba ukončené bodkočiarkou a nasledované číslom poltahu, napr. "a3 b4; c7 d8; 12"). Vyrob triedu `RandomPlayer` s metódou

```
1 Move findMove(const Board& b)
```

ktorá vráti náhodný prípustný ďah. Napokon uprav `main.cc` tak, aby hral hru: načíta ďah bieleho, skontroluje, či je korektný, vypíše šachovnicu, spraví náhodný ďah čierneho, ... až kým niekto nevyhra.

Toto nebolo ľahké, ale zjavne to má od dôstojného protivníka ďaleko. Ak chceme naprogramovať lepsieho hráča, je dobré si najprv preskúmať, čo vlastne chceme. To, čo si budeme hovoriť, platí nielen pre Breakthrough, ale všeobecne pre hry dvoch hráčov (ktorí sa striedajú na ďahoch), sú s *úplnou informáciou* (t.j. obaja hráči vidia všetko, na rozdiel napr. od väčšiny kartových hier), sú *deterministické* (t.j. nehrá pri nich úlohu náhoda, ako napr. pri hre Monopoly), a sú s *nulovým súčtom* (zero-sum t.j. výhra jedného hráča je prehrou druhého). V takýchto hrách budeme hľadať stratégii pre jedného hráča. Čo je to stratégia? Niečo, čo mi v každej situácii povie, ako mám hrať. Zoberme si napr. Tic-Tac-Toe. Ako by mohla vyzerat stratégia pre hráča **X**? Začne napr. tým, že stratégia mu povie dať prvý ďah do stredu. Teraz je na ďahu **O** a **X** musí byť pripravený odpovedať na všetky možné ďahy **O**. V skutočnosti sú iba dva, ostatné možnosti sú symetrické.



## Prehľadávanie herných stromov

Ak  $\textcircled{O}$  zahrá , stratégia povie  $\texttimes$  zahrať  a ak  $\textcircled{O}$  zahrá , tak mu povie zahrať  atď. Stratégia pre  $\texttimes$  je teda strom, v ktorom vo vrcholoch sú herné pozície, pričom ak je na ťahu  $\texttimes$ , vyberie sa jeden ťah a ak je na ťahu  $\textcircled{O}$ , musia sa brať do úvahy všetky ťahy. Pozri sa teraz, čo sa stane, ak sa vyskytne pozícia  (na ťahu je  $\textcircled{O}$ ). Nech  $\textcircled{O}$  zahrá hocičo,  $\texttimes$  môže vyhrať buď v nasledujúcom ťahu, alebo (ak  $\textcircled{O}$  zahrá  a následne  $\texttimes$  zahrá ) v druhom ťahu. Pozíciu, z ktorej pre  $\texttimes$  existuje vyhľadávajúca stratégia (t.j. má zaručenú výhru nech  $\textcircled{O}$  hrá hocijako) voláme vyhľadávajúca (a dávame jej hodnotu 1). Podobne, ak existuje vyhľadávajúca stratégia pre  $\textcircled{O}$  (t.j.  $\textcircled{O}$  vie vyhrať, nech  $\texttimes$  robí čokoľvek), takúto pozíciu voláme prehľadávajúca (pozéráme sa na svet z pohľadu  $\texttimes$ ) a dávame jej hodnotu -1. V našom prípade sme videli, že  je vyhľadávajúca pozícia, a preto aj  je vyhľadávajúca ( $\texttimes$  je na ťahu a môže zahrať ). Takže od  $\textcircled{O}$  je chyba zahrať na začiatku .

Pozrime sa teraz na pozíciu . Stratégia pre  $\texttimes$  (ktorú som celú nedokreslil, ale je ľahké si ju domyslieť) mu zaručí, že nech  $\textcircled{O}$  robí čokoľvek,  $\texttimes$  nikdy neprehrá. Takže  $\texttimes$  má *neprehľadávajúcu* stratégii. Môže mať inú stratégii, ktorá by bola vyhľadávajúca?

**Úloha 126.** Nájdi neprehľadávajúcu stratégii pre  $\textcircled{O}$  z pozície  (na ťahu je  $\texttimes$ ).

Čo to pre hráča  $\texttimes$  znamená, že  $\textcircled{O}$  má neprehľadávajúcu stratégii?  $\texttimes$  vie, že ak by sa odchýlil od svojej neprehľadávajúcej stratégie,  $\textcircled{O}$  ho určite nenechá vyhrať, a navyše sa mu môže ešte stať, že prehrá. Preto je prehľadávajúca stratégia lepšia. A rovnako pre  $\textcircled{O}$ .  je príkladom pozície, ktorej hovoríme *remízová* a dávame jej hodnotu 0. Je to pozícia, v ktorej obaja hráči majú neprehľadávajúcu stratégii – optimálne hra vždy viedie k remíze. Môžu byť aj iné pozície ako vyhľadávajúca, prehľadávajúca a remízová? Zjavne nie. Predstav si totiž kompletný strom hry. Každý koncový vrchol má hodnotu 0, 1, alebo -1, lebo na konci hry buď niekto vyhrať, alebo je remíza. Teraz sa pozrime na také vrcholy, ktoré vedú len do koncových vrcholov. Ak je v takomto vrchole  $V$  na ťahu  $\texttimes$  a existuje ťah, v ktorom by vyhral (teda ťah do vrchola s hodnotou 1),  $\texttimes$  urobí ten ťah a vyhra. Hodnota  $V$  je preto 1. Ak ťah, ktorý by  $\texttimes$  zaručil výhru neexistuje, ale existuje ťah do remízy,  $\texttimes$  môže urobiť ten a hodnota  $V$  bude 0. V opačnom prípade (t.j. nech  $\texttimes$  urobí čokoľvek, tak prehrá) bude hodnota  $V$  -1. Inými slovami, ak je vo  $V$  na ťahu  $\texttimes$ , hodnota  $V$  bude maximum hodnôt všetkých jeho synov. Podobne ak je vo vrchole  $V$  na ťahu  $\textcircled{O}$ , hodnota  $V$  je minimum hodnôt jeho synov. Takto môžeme postupovať ďalej: vždy zoberieme vrchol  $V$ , ktorého všetci nasledovníci majú priradenú hodnotu (rozmysli si, že taký vrchol vždy musí existovať) a priradíme hodnotu aj  $V$ .

Týmto sme sa presvedčili, že pre akúkoľvek (deterministickú, s úplnou informáciou, s nulovým súčtom) hru je každá pozícia vyhľadávajúca, prehľadávajúca, alebo remízová. To nielenže znamená, že vždy existuje optimálna stratégia, ale v predchádzajúcim odstavci sme aj opísali postup (nazývaný MINIMAX), ako ju nájsť. Tento postup vieme naprogramovať jednoduchou rekurzívnu procedúrou `int value(const Board &b)`: ak je to koncová pozícia, vrátíme výfazu, inak vyrobíme všetky možné ťahy, rekurzívne zistíme ich hodnoty a vrátíme minimum alebo maximum, podľa toho, ktorý hráč je na ťahu.

Aby sme nemuseli zvlášť riešiť maximalizujúceho hráča (volajme ho biely,  $\texttimes$ , alebo MAX) a minimalizujúceho hráča (čierny,  $\textcircled{O}$ , MIN), je pri implementácii príjemnejšie vracať hodnotu pozície z pohľadu hráča, ktorý je práve na ťahu (t.j. hodnota 1 znamená vyhľadávajúca pozícia pre hráča, ktorý je práve na ťahu). V tomto pohľade každý hráč hľadá ťah, ktorý maximalizuje hodnotu pozície. Pri tom sa dá využiť vlastnosť nulového súčtu: čo je výhra (1) z pozície bieleho, je prehra (-1) z pozície čierneho. Kedže rekurzívne zistíme hodnoty nasledovníkov, dostanem ich z pozície súpera, takže z môjho pohľadu budú čísla opačné. Tento spôsob programovania sa zvykne volať NEGAMAX. Ak si do triedy `Board` dorobíme metódu `int Board::toPlay() const`; ktorá vráti 1, ak je na ťahu biely a -1 ak je na ťahu čierny, môžem napísať:

```

1 // hodnota pozície b
2 int value(const Board &b) {
3     int side = b.toPlay(); // 1, -1 podľa toho, kto je na ťahu
4     int w = b.winner(); // má b víťaza?
5     if (w == White) return side; // ak vyhral biely a je práve na ťahu, tak 1,
6                                // ak vyhral biely a na ťahu je čierny, tak -1
7     if (w == Black) return -side; // ak vyhral čierny, tak podobne
8
9     auto ms = b.legalMoves(); // vytvorím všetky prípustné ťahy
10    int mx = -1; // hodnota z pohľadu toho, kto je na ťahu
11
12    for (int i = 0; i < ms.size(); i++) { // ideme testovať každý možný ťah
13        Board bb = b;
14        bb += ms[i]; // urob daný ťah
15        int v = -value(bb); // rekurzívne vyráta hodnotu nasledovníka
16                                // výsledok je z pohľadu protivníka,
17                                // takže z môjho pohľadu je opačný
18        if (v > mx) mx = v; // pamäťam si doteraz najlepšiu možnosť
19        if (mx == 1) break; // lepšie už nemôže byť, netreba pokračovať
20    }
21    return mx;
22 }
```

**Úloha 127.** Napiš program, ktorý prečíta zo vstupu pozíciu, nájde jej hodnotu (1 alebo -1 z pohľadu bieleho) a vypíše ju. Potom vypíše všetky prípustné ťahy a hodnoty pozícií, do ktorých vedú. Používateľ potom môže zadat niektorý ťah a opäť sa nájde hodnota pozície a všetkých prípustných ťahov.

```

c6 d5 h6; d6 b3 g7 h8; 0
  a   b   c   d   e   f   g   h
+---+---+---+---+---+---+---+
8 |   |   |   |   |   |   |   | 0 | 8
+---+---+---+---+---+---+---+
7 |   |   |   |   |   |   |   | 0 | 7
+---+---+---+---+---+---+---+
6 |   |   | X | 0 |   |   |   | X | 6
+---+---+---+---+---+---+---+
5 |   |   |   | X |   |   |   |   | 5
+---+---+---+---+---+---+---+
4 |   |   |   |   |   |   |   |   | 4
+---+---+---+---+---+---+---+
3 |   | 0 |   |   |   |   |   |   | 3
+---+---+---+---+---+---+---+
2 |   |   |   |   |   |   |   |   | 2
+---+---+---+---+---+---+---+
1 |   |   |   |   |   |   |   |   | 1
+---+---+---+---+---+---+---+
  a   b   c   d   e   f   g   h
1
c6-b7: 1
c6-c7: 1
c6-d7: 1
d5-e6: -1
h6-g7: 1
h6-h7: 1
```

## Prehľadávanie herných stromov

Problém s týmto programom je v tom, že na to, aby našiel hodnotu nejakej pozície, musí prehľadať celý jej podstrom. Pre pozície, ako tá na predchádzajúcom príklade, to je dostatočne málo, ale koľko by to bolo, keby si zadal štartovaciu pozíciu? Čažko to povedať presne, ale keby sme veľmi zhruba odhadli, že v priemere má pozícia 12 prípustných ťahov (je 16 kameňov, takže najviac môže byť 48 možných ťahov, ale väčšinou je to menej) a partia trvá 30 ťahov (t.j. 60 polťahov), tak strom by mal

$$12^{60} = 56347514353166785389812313795980500551139163800306781874894667776$$

vrcholov. Tielo čakať určite nechceš.

Ked chceme použiť MINIMAX na praktické hranie, najjednoduchší spôsob je obmedziť hĺbku stromu. Namiesto toho, aby sme pre danú pozíciu prehľadávali celý podstrom až po koncové pozície, pôjdeme iba do fixnej hĺbky (napr. štyri ťahy dopredu) a potom odhadneme, do ako dobrej pozície sme sa dostali. Na to budeme potrebovať nejakú funkciu `eval`, ktorá zoberie pozíciu a vráti odhad jej hodnoty: číslo z rozsahu  $[-1, 1]$ . Môžeš sa spýtať, prečo teda, ak budeme mať takú výhodnocovaciu funkciu, ju nepoužijeme priamo, ale strácame čas prehľadávaním. Problém je v tom, že vyrobí takú funkciu vôbec nie je ľahké a funkcia, ktorú budeme mať, bude nutne nepresná. A dúfame, že prehľadávanie nám pomôže: predpokladáme, že aj taký zlý ťah, ktorý by naša funkcia neodhalila, sa po pár ťahoch prejaví tak, že aj jednoduchá funkcia (napr. taká, ktorá len počíta počet kameňov) to zistí.

**Úloha 128.** Naprogramuj hráča `MinimaxPlayer`. Podobne ako `RandomPlayer` z úlohy 125 bude mať metódu `Move findMove(const Board& b)`. V tej sa pre každý prípustný ťah nájde hodnota výslednej pozície pomocou prehľadávania podobne ako vo funkciu `value`, ale navyše s parametrom `hľbka`, ktorý sa v každom rekúzívnom volaní bude zmenšovať. Ked dosiahne nulu, zavolá sa metóda `MinimaxPlayer::eval(const Board&)`. Môžeš zatiaľ použiť niečo veľmi jednoduché, napr.

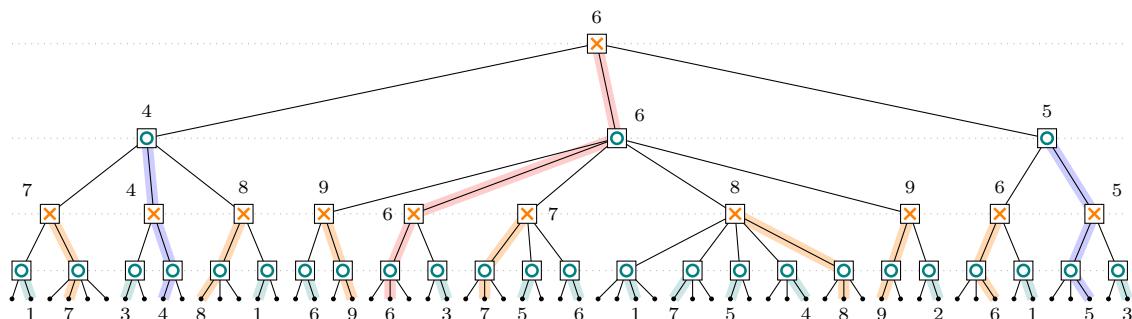
```
1 (počet_kamenov_bieleho - počet_kamenov_cierneho)/16.0
2
```

(to delenie 16.0 je tam preto, aby som dostal číslo z intervalu  $[-1, 1]$ , ale vlastne to ani nepotrebujem). Výsledkom volania `findMove` bude ten ťah, ktorý sa dostane do najlepšej pozície (ak je takých viacero, tak vyber náhodný). Podobne ako v úlohe 125 sprav, aby `MinimaxPlayer` hral hru.

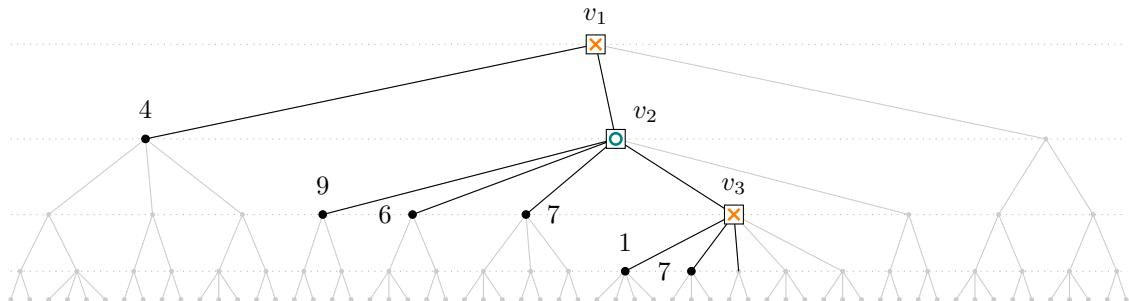
Čím väčšiu hĺbku prehľadávania zvolíme, tým dlhšie trvá urobiť ťah. Pri hĺbke 4 to u mňa išlo ešte celkom rýchlo (zhruba 1-2s na ťah), pri hĺbke 5 to už ale bolo aj okolo 2min na ťah. A sice hrá oveľa lepšie, ako náhodný hráč, stále to ani s hĺbkou 5 nie je nič moc. Je ale veľa spôsobov, ako sa to dá zlepšiť.

### $\alpha\beta$ -orezávanie

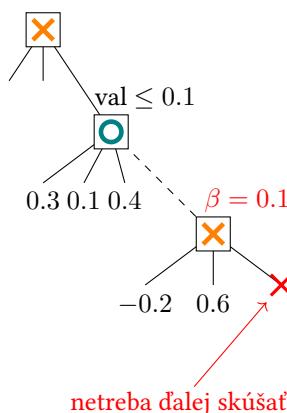
Prvý spôsob, ako vylepšiť MINIMAX, je založený na zistení, že dosť často sa veľa vrcholov prehľadáva zbytočne. Ak sa na MINIMAX pozrieme pohľadom zdola-hore, máme listy, ktoré majú hodnoty a vnútorné vrcholy, ktoré vždy vyberú niektorého (najväčšieho alebo najmenšieho) syna. Celé prehľadávanie si preto môžeme prestaviť tak, že jednotlivé hodnoty sa presúvajú z listov smerom hore



Cieľom pri prehľadávaní je nájsť ten list, ktorého hodnota sa dostane až úplne hore. Predpokladajme teraz, že hľadáme hodnotu vrchola  $v_1$ , ktorý je maximalizačný. Zistili sme, že prvý syn má hodnotu 4 a pokračovali sme hľadať hodnotu  $v_2$ . Tam sme našli synov s hodnotami 9, 6, a 7 a opäť rekúrzívne hľadáme hodnotu  $v_3$ . O tom sme zistili, že má synov s hodnotami 1 a 7. Čo teraz vieme povedať? Nech by zvyšní synovia  $v_3$  mali akékoľvek hodnoty, hodnota  $v_3$  bude aspoň 7. Zároveň vidíme, že hodnota  $v_2$  bude najviac 6. V tomto momente preto môžeme prestaviť vyhodnocovať synov  $v_3$ , lebo určite sa nikto z nich nedostane nad  $v_2$ .



Toto pozorovanie vieme zovšeobecniť: ak prehľadávam nejaký vrchol a viem, že na niekde ceste do koreňa je minimalizačný vrchol s hodnotou najviac  $\beta$ , tak viem, že žiadna hodnota väčšia ako  $\beta$  sa do koreňa nedostane, a teda akonáhle zistím, že nejaký vrchol má hodnotu viac ako  $\beta$ , nemusím ďalej prehľadávať:



Symetrická situácia je pre minimalizačné vrcholy. Ak prehľadávam nejaký vrchol a viem, že na ceste do koreňa je maximalizačný vrchol s hodnotou aspoň  $\alpha$ , tak viem, že žiadna hodnota menšia ako  $\alpha$  sa do koreňa nedostane a môžem prestaviť prehľadávať.

Naprogramoval by som to tak, že do rekúrzívnej funkcie `findMove` by som pridal dva parametre: `alpha` a `beta`, v ktorých si pamätam najväčšiu známu hodnotu v maximalizačnom vrchole a najmenšiu známu hodnotu v minimalizačnom vrchole na ceste do koreňa. Keď prehľadávam minimalizačný vrchol, zmenšujem hodnotu `beta`, akonáhle nájdem menšieho syna. Ak som v maximalizačnom vrchole, zväčšujem hodnotu `alpha` akonáhle nájdem väčšieho syna. Do rekúrzívneho volania posielam aktualizované hodnoty `alpha` a `beta`. Akonáhle zistím, že hodnota vrchola, ktorý prehľadávam, je mimo intervalu  $\langle \alpha, \beta \rangle$ , prestanem s vyhodnocovaním.

Podobne ako pri pôvodnom MINIMAX prehľadávaní, aj tu môžeme využiť vlastnosť nulového súčtu a spojiť obe vetvy do jednej. Funkcia `search` bude opäť vracať nie skutočnú hodnotu (t.j. 1 ako výhra bieleho a -1 ako výhra čierneho), ale hodnotu z pohľadu hráča, ktorý je práve na ťahu (t.j. 1 vyhrá a -1 prehrá ten, kto je v pozícii `b` na ťahu). Dostali by sme napr. niečo takéto:

```

1 float search(const Board& b, int hlbka, double alpha, double beta) {
2     // tu treba otestovať, či má pozícia víťaza alebo či hĺbka dosiahla 0
3     // ...
4     // tu začína rekúrzívna časť
5     auto ms = b.legalMoves(); // vytvoríme všetky prípustné ťahy

```

```

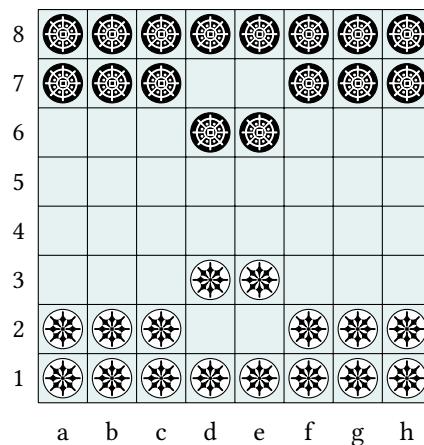
6   float best = -10;           // -10 je zarážka, skutočné hodnoty máme z rozsahu -1 .. 1
7   for (int i = 0; i < ms.size(); i++) {
8     Board bb = b;
9     bb += ms[i];
10    // striedame minimalizáciu a maximalizáciu, preto vymeníme znamienko
11    // a navyše vymeníme alfu a betu
12    float v = -search(bb, hlbka - 1, -beta, -alpha);
13    if (v > best) best = v;           // pamäťame si doteraz najlepšiu hodnotu
14    if (best > alpha) alpha = best;  // podľa nej aktualizujeme alfu
15    if (alpha >= beta) break;        // ak sme našli viac ako beta, nejdeme ďalej
16  }
17  return best;
18 }
```

**Úloha 129.** Doprogramuj triedu `AlphaBetaPlayer`.

Koľko  $\alpha\beta$ -orezávanie pomáha, to závisí do veľkej miery od toho, v akom poradí prechádzame strom. Ak vždy ako prvý vyberieme najlepší ťah, ostatné možnosti sa orežú oveľa rýchlejšie. Preto sa zvykne pole ťahov `ms` utriediť tak, aby sa zvýšila šanca, že dobrý ťah pôjde skôr. Napr. môžem najprv skúsať ťahy, kde sa vyhodil kameň a pod. Na druhej strane, čas, ktorý sa spotrebuje triedením v každom vrchole, môže narastať, ak by som na triedenie použil príliš zložitú funkciu. Takže je to niečo, s čím sa treba pohrať a dobre to vyladiť pre konkrétnu hru.

**Transpozičné tabuľky**

V `Breakthrough` a aj v iných hrách sa často vyskytuje situácia, že sa do tej istej pozície dá dostať pomocou rôznych postupností ťahov (takéto postupnosti sa zvyknú volať *transpozicie*): napr. zo začiatnej pozícii `1.d2-d3, d7-d6, 2.e2-e3, e7-e6` aj `1.d2-e3, e7-d6, 2.e2-d3, d7-e6` pridú do pozícii



V našom prehľadávaní to nijak neberieme do úvahy a každú takúto pozíciu prehľadávame nanovo. Podobne, ako keď sme hovorili o memoizácii v kapitole 14, aj teraz je prirodzené snažiť sa nepočítať viackrát to isté, ale si vypočítané hodnoty odkladať a dúfať, že sa ešte niekedy budú hoditi. Urobíme si preto hešovaciu tabuľku, kde si pre každú už navštívenú pozíciu budeme pamätať jej hodnotu.

Na to, aby sme mohli použiť `unordered_map` potrebujeme, ako sme hovorili v kapitole 32, špecializovať `std::Hash<Board>`. V type `Board` máme uložené dve 64-bitové čísla na mapy bielych a čiernych kameňov, a ešte číslo polťahu. Aby sme ich dobre zamixovali, začneme s hešovacou funkciou pre celé čísla; tu som si zobrať tzv. *murmur hash*. Do súboru `board.h` si pridám

```

1 template <>
2 struct std::hash<Board> {
3     uint64_t murmur64(uint64_t) const;
4     uint64_t operator()(const Board &b) const;
5 };

```

a do súboru `board.cc` dám definíciu pre `murmur`:

```

1 uint64_t std::hash<Board>::murmur64(uint64_t x) const {
2     x ^= x >> 27;
3     x *= 0x3C79AC492BA7B653ULL;
4     x ^= x >> 33;
5     x *= 0x1C69B3F74AC4AE35ULL;
6     x ^= x >> 27;
7     return x;
8 }

```

Ešte ostáva vymyslieť, ako skombinovať jednotlivé zložky z `Board`. To spravím cez `XOR` napr. takto:

```

1 uint64_t std::hash<Board>::operator()(const Board &b) const {
2     return murmur64(b.pmap[0] ^ 0xFEEDDADBEFFull) ^ murmur64(b.pmap[1]) ^
3             murmur64(b.ply);
4 }

```

Ked' ešte pridám priamočiaro napísaný `bool Board::operator==(const Board&)`, môžem `Board` používať ako kľúč v hešovacej tabuľke. Ešte to má jeden háčik: pretože používame  $\alpha\beta$ -orezávanie, nevyrátame vždy skutočnú hodnotu pozície, ale iba nejaký jej odhad (napr. že je väčšia ako  $\beta$ ). Navyše, môže sa stať, že z tej istej pozícii púšťam prehľadávanie do rôznej hľbky, a teda dostanem aj rôzne výsledky. Preto si o pozícii budem pamätať dolný (*lower bound*, `lb`) a hroný (*upper bound*, `ub`) odhad jej hodnoty (t.j. viem, že skutočná hodnota z pozície hráča, ktorý je na ťahu, je medzi `lb` a `ub`), hľbku, do akej som z nej prehľadával, a najlepší ťah:

```

1 struct AlphaBetaPlayer {
2     ...
3
4     struct TTableEntry {
5         float ub = 100, lb = -100;
6         int hlbka = -1;
7         Move m;
8     };
9
10    std::unordered_map<Board, TTableEntry> tt;
11 }

```

Vo funkciu `search` sa najprv pozriem, či nemám v transpozičnej tabuľke zapamätanú pozíciu `b` s dostatočnou hľbkou. Ak áno, upravím si hodnoty `alpha` a `beta`. Pretože pri prehľadávaní ma zaujímajú iba hodnoty menšie ako `beta`, ak zistím, že skutočná hodnota je menšia ako `ub`, môžem si rovno znížiť `beta = ub`. Podobne pre `alpha`. Na konci funkcie aktualizujem hodnotu v tabuľke (ak medzitým tabuľka priliš narastla, tak ju vymažem):

```

1 MoveValue AlphaBetaPlayer::search(const Board& b, int hlbka, float alpha, float beta) {
2     ...
3
4     // na začiatku prečítame z transpozičnej tabuľky
5     auto it = tt.find(b);

```

```

6   if (it != tt.end()) {
7     auto& e = it->second;
8     if (e.hlbka >= hlbka) {
9       if (e.lb >= beta) return {e.m, e.lb};
10    if (e.ub <= alpha) return {e.m, e.ub};
11    alpha = std::max(alpha, e.lb); // max a min z knižnice <algorithm>
12    beta = std::min(beta, e.ub);
13  }
14}
15 ...
16 ...
17
18 // na konci aktualizujeme transpozičnú tabuľku
19 if (tt.size() > 10000000) tt.clear();
20 auto& e = tt[b];
21 if (e.hlbka <= hlbka) {
22   e = TTableEntry();
23   e.hlbka = hlbka;
24   e.m = res.m;
25   if (res.val <= alphaOrig)
26     e.ub = res.val;
27   else if (res.val >= beta)
28     e.lb = res.val;
29   else
30     e.ub = e.lb = res.val;
31 }
32 ...
33 ...
34 }
```

## Evaluáčná funkcia

Ďalším miestom na vylepšenie je funkcia `eval`. V článku [Lorentz, Horey: Programming Breakthrough<sup>1</sup>](#) autori navrhujú evaluačnú funkciu, z ktorej som si vybral tieto tri časti:

1. Hráč, ktorý má viac kameňov dostane 10 bodov za každý kameň navyše.
2. Každý hráč dostane body za každý svoj kameň podľa jeho pozície (tabuľka je z pohľadu bieleho)

36	36	36	36	36	36	36	36	36
20	28	28	28	28	28	28	28	20
16	21	21	21	21	21	21	21	16
11	15	15	15	15	15	15	15	11
7	10	10	10	10	10	10	10	7
4	6	6	6	6	6	6	6	4
2	3	3	3	3	3	3	3	2
5	15	15	5	5	15	15	15	5

<sup>1</sup>[https://doi.org/10.1007/978-3-319-09165-5\\_5](https://doi.org/10.1007/978-3-319-09165-5_5)

3. Každý bezpečný kameň dostane 50% bonus oproti hodnote z bodu 2. Kameň je bezpečný, ak má aspoň toľko obrancov, ako útočníkov (t.j. súper nemôže sériou výmen získať kameň). Napr. v nasledovnej pozícii sú všetky kamene bezpečné, ale ak by bol na ťahu biely a potiahol by **f3-g4**, kameň na pozícii **g4** by už bezpečný neboli. Rovnako, keby biely pohol **e4-d5**, kameň na **d5** by nebol bezpečný, lebo by mal dvoch útočníkov (**c6** a **e6**) a iba jedného obrancu (**c4**); čierny by teda sériou výmen získal o kameň viac.

8							
7							
6							
5							
4							
3							
2							
1							
	a	b	c	d	e	f	g
							h

**Úloha 130.** Doprogramuj do triedy `AlphaBetaPlayer` transpozičné tabuľky a vylepšenú evaluačnú funkciu. Naprogramuj šablónu

```

1 template <typename P1, typename P2>
2 int duel() {
3     P1 p1;
4     P2 p2;
5     ...
6 }
7

```

ktorá si vytvorí dvoch hráčov, zohrá partiu a vráti výsledok.

Teraz je čas na experimenty. Je veľa rôznych možností, ako hráča vylepšíť. Na jednej strane je kľúčový parameter počet prehľadaných vrcholov, takže sa oplatí jednotlivé časti programu zrýchľovať. Napr. predrátať hešovací klúč, používať rýchlejšiu špeciálne upravenú hešovaciu tabuľku, rýchlejšie rátať evaluačnú funkciu a pod. Na druhej strane je snaha viac stromu odrezať, takže napr. lepšie triediť ťahy pri prehľadávaní, prípadne použiť tzv. *iterative deepening*, kde sa postupne spúšťa prehľadávanie s väčšími a väčšími hĺbkami a na triedenie ťahov sa používajú hodnoty vyrátané v predchádzajúcej iterácii. Ďalej sú to vylepšenia samotného algoritmu: dá sa pridať knižnica otvorení (nechať hrať hráča samého proti sebe a pozerať sa, ktoré pozicie na začiatku častejšie vedú k výhre; tie si potom zapamätať a snažiť sa ich častejšie voliť pri hre), dá sa pridať riešenie koncoviek (buď v situácii, keď už ostáva iba málo kameňov, alebo keď je niektorý kameň blízko cieľa sa dá buď použiť dynamické programovanie alebo si nejaké patterny predrátať) a pod. Takisto sa dá robiť niečo proti tzv. *horizon effect*: keďže prehľadávanie ide do fixnej hĺbky, môže sa stať, že originálny algoritmus zastane uprostred série výmen (napr. biely vyhodí čierneho a má o kameň naviac, ale už sa nezistí, že čierny môže vyhodiť tiež a budú mať rovnako); preto sa zvykne pred skončením prehľadávania v nulovej hĺbke pokračovať v sérii výmen, kým sa nedostane do "tichej" pozície.

Zopár dobrých nápadov by malo stačiť na to, aby program hral lepšie ako bežný netrénovaný človek. Môžeš používať `duel` na to, aby si si overil, ktoré zmeny program zosilnia a ktoré naopak oslabia.

## 36 2D grafika: knižnica SDL

Doteraz všetky naše programy číitali vstup zo štandardného vstupu (`cin`) a vypisovali na štandardný výstup (`cout`), prípadne do súboru. V tejto časti bude našim hlavným cieľom naučiť sa písat programy, ktoré pracujú s grafikou, a dať nášmu programu na Breakthrough pekné grafické prostredie.

Na úvod zlá správa: C++ nemá v štandardných knižničiach žiadne funkcie na prácu s grafikou. Nie je to tým, že by vývojári na to zabudli, je to zámer: C++ sa používa v rôznych zariadeniach, od chladničiek po datacentrá, takže si ľahko predstaví rozumnú sadu funkcií, ktorá by všetkých uspokojila.

Programovanie grafiky je preto plné krvavých detailov: sú rôzne operačné systémy, rôzne grafické systémy, rôzny hardvér, a pre každú kombináciu to treba robiť jemne inak. Dobrá správa je, že existujú knižnice, ktoré špinavú robota urobia za nás. Tento tutoriál sa snaží byť "self-contained", to znamená, že okrem komplilátora, štandardných knižníc (a programu `make`) nepoužívame žiadne externé knižnice. Teraz ale urobíme výnimku a predstavíme si knižnicu [Simple DirectMedia Layer](#)<sup>1</sup> (SDL). Aj keď nie je súčasťou štandardu, je veľmi rozšírená a dá sa nainštalovať úplne všade. Zároveň je *open source*, takže si napr. na [GitHub](#)<sup>2</sup> môžeš pozrieť ako je naprogramovaná. Zlá správa je, že SDL je napísaná v jazyku C a nie C++. Dobrá správa je, že nám to nijak nevadí: v C++ ju môžeš normálne používať. Najprv ale: čo je to knižnica?

Z kapitoly 34 vieš, ako skompilovať program po častiach: najprv sa urobí object code v súboroch `.o` a tie sa potom zlinkujú do výsledného programu (na to musí linker nájsť práve jednu funkciu, ktorá sa volá `main`, a tá sa spustí ako prvá v programe). Ak nemáš funkciu `main`, stále môžeš vyrobiť object code, ale nedá sa z neho vytvoriť spustiteľný súbor. Viacero object code súborov sa ale dá zabalíť do špeciálneho archív, ktorý sa volá knižnica. Tá sa potom dá jedným príkazom pridať linkeru. Knižnica SDL je rozdelená na viacero častí (na font, zvuk, bitmapy a pod.). Na to, aby si ich mohol používať, ich treba mať nainštalované. Ako to presne spraviť, záleží od operačného systému, napr. v linuxových distribúcích ako Debian, Ubuntu a pod. sa dá napísat `sudo apt-get install libSDL2-dev`. Tým sa jednak nakopíruje knižnica do nejakého štandardného adresára, kam sa komplilátor pozera (napr. `/usr/lib/x86_64-linux-gnu/libSDL2.so`) a jednak sa nakopírujú súbory `.h` s deklaráciami do nejakého iného štandardného adresára (napr. `/usr/include/SDL2`).

Ak máš knižnicu SDL nainštalovanú, môžeš použiť [Makefile](#)<sup>3</sup> z kapitoly 34 a v linkovacom príkaze pre hlavný súbor pridať

```
main: $(objects)
    mkdir -p $(builddir)
    g++ -O3 -std=c++20 $(objects) -o $@ -lSDL2
```

Tým hovoríš linkeru, že okrem tvojich object code súborov `$(objects)` má brať do úvahy aj tie, ktoré nájde v súbore `libSDL2.so`. Teraz otestujeme, či všetko funguje. Napíš si tento program `main.cc`<sup>4</sup>

```
1 #include <SDL2/SDL.h>
2
3 SDL_Window* window = nullptr;
4 SDL_Renderer* renderer = nullptr;
5
6 int main() {
7     SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER | SDL_INIT_AUDIO);
8
9     window = SDL_CreateWindow("prvy_pokus", SDL_WINDOWPOS_CENTERED,
```

<sup>1</sup><https://www.libsdl.org/>

<sup>2</sup><https://github.com/libsdl-org/SDL>

<sup>3</sup><https://github.com/pocestny/programovanie/raw/master/materialy/sdl/01/Makefile>

<sup>4</sup><https://github.com/pocestny/programovanie/raw/master/materialy/sdl/01/main.cc>

```

10             SDL_WINDOWPOS_CENTERED, 800, 600,
11             SDL_WINDOW_RESIZABLE | SDL_WINDOW_SHOWN);
12 renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);
13
14     SDL_SetRenderDrawColor(renderer, 250, 180, 0, 255);    // nastav žltú farbu
15     SDL_RenderClear(renderer);                            // vyfarbi celú plochu
16
17     SDL_Rect r{.x=300, .y=200, .w=200, .h=200};        // obdĺžnik v strede okna
18     SDL_SetRenderDrawColor(renderer, 20, 30, 180, 255);  // nastav modrú farbu
19     SDL_RenderFillRect(renderer, &r);                   // vyplň obdĺžnik
20
21     SDL_RenderPresent(renderer);                        // zobraz nakreslené veci
22
23     SDL_Delay(2000);                                 // 2s čakaj
24
25     SDL_DestroyWindow(window);
26     SDL_Quit();
27 }
```

a spusti `make`. Mal by sa ti vyrobil súbor `main`, ktorý keď spustíš, otvorí sa žlté okno s modrým štvorcom a po dvoch sekundách zmizne. Ak sa ti to podarilo, môžeme pokračovať a pozrieť sa, ako je SDL urobená. Je v nej dosť vecí, tu ti ukážem iba základných zopár, ktoré budeme potrebovať. Zoznam všetkých funkcií nájdete na [SDL2 wiki](#)<sup>5</sup> v časti [API reference](#).

Prvá vec je celkový dizajn. Keďže SDL je písaná v jazyku C, nemá konštruktory, deštruktory, ani metódy (iba typy a "obyčajné" funkcie), preto dizajnéri SDL ich simulujú "ručne". Jednoduchý príklad: keby si chcel mať spievajúcu uhorku, v C++ si spravíš typ, s ktorým sa bude pracovať asi takto:

```

1 Uhorka* uh = new Uhorka("Chuck");    // volá sa konštruktor
2 uh->spievaj(42);                    // metóda
3 delete uh;                          // volá sa deštruktur
```

Alebo takto:

```

1 {
2     Uhorka uh("Chuck"); // volá sa konštruktor
3     uh.spievaj(42);   // metóda
4 } // tu sa tiež volá deštruktur
```

V SDL (ako v skoro každej Ččkovskej knižnici) by to bolo urobené nejak takto:

```

1 Uhorka* uh = SDL_VyrobUhorku("Chuck"); // "konštruktor" alokuje pamäť a vráti pointer
2 SDL_UhorkaSpievaj(uh, 42); // *this sa pošle ručne
3 SDL_ZmazUhorku(uh);      // "deštruktur" uvoľní pamäť
```

Podme si rozober nás `main.cc`. Najprv treba (riadok 1) vložiť deklarácie zo `SDL2/SDL.h`. Pred prvým volaním nejakej funkcie treba celú knižnicu zapnúť (riadok 7) volaním `SDL_Init(...)`; parameter udáva, ktoré časti sa majú inicializovať<sup>6</sup>. Pred skončením programu je slušné celú knižnicu vypnúť (riadok 26).

Prvým typom, ktorý treba použiť, je `SDL_Window`, v ktorom sú všetky premenné potrebné na spravovanie okna na obrazovke. Podobne ako väčšina typov v SDL, je `SDL_Window` tzv. *opaque* typ: nikdy nebudeme potrebovať pristupovať k jeho premenným, vždy ho iba pošleme ako parameter do príslušných funkcií. Hneď na začiatku na

<sup>5</sup><https://wiki.libsdl.org/SDL2>

<sup>6</sup>To `SDL_INIT_VIDEO`, `SDL_INIT_TIMER` atď. sú konštandy, ktoré sú postupne mocniny 2, preto v dvojkovej sústave každému zodpovedá jeden bit. Takže keď ich spojíš pomocou bitového OR, dostaneš číslo, v ktorom o každom subštémeste zistíš, či ho treba zapnúť, tak, že skontroluješ príslušný bit. Toto je technika, ktorá sa často používa na to, aby si mal sadu prepínačov v jednom čísle.

riadku 3 si spravíme globálnu premennú `SDL_Window *window`. Prvá vec, ktorú po zapnutí knižnice urobíme, je vytvorenie okna. Na riadku 9 zavoláme “koštruktor”, ktorý vytvára okno<sup>7</sup>. V tomto momente sa na obrazovke objaví prázdne okno. Po skončení práce okno zavrieme volaním “deštruktora” na riadku 25.

Zobrazovacie funkcie nepracujú priamo s oknom, ale s premennou typu `SDL_Renderer`<sup>8</sup>. Na riadku 12 vytvoríme renderer, ktorý bude kresliť do okna `window`. V SDL má ľavý horný roh okna súradnice  $(0, 0)$ ,  $x$ -ová súradnica ide smerom doprava,  $y$ -ová smerom dole a všetky sú celočíselné (t.j. pixely).

V nasledujúcich riadkoch ideme konečne kresliť: na riadku 14 rendereru povíme, aby použil tmavožltú farbu (parametre sú RGBA zložky) a následne na riadku 15 aby prekreslil celé pozadie.

Na riadku 17 vytvoríme premennú typu `SDL_Rect`, ktorá reprezentuje obdĺžnik (premenné `x`, `y` sú pozícia ľavého horného rohu v okne, `w`, `h` sú šírka (`width`) a výška (`height`). Tu som navyše použil nový spôsob priradenia. Doteraz si vedel, že premennú typu `struct` viēš nastavíš tak, že v kučeravých zátvorkách napíšeš všetky jej zložky. Problém je, že pri tom záleží iba na ich poradí v definícii typu, takže sa ľahko dá pomýliť. Aby sa predišlo omylom, dá sa písat bodka a meno položky. Zápis na riadku 17 je preto to isté, ako `SDL_Rect r; r.x=300; r.y=200; r.w=200; r.h=200;`<sup>9</sup>

Nasledujúce dva riadky sú zrejmé: nastavíme modrú farbu a zavoláme `SDL_RenderFillRect`, čo je funkcia, ktorá, ako napovedá názov, vyplní obdĺžnik aktuálne nastavenou farbou. Podobných funkcií je niekoľko:

---

<code>SDL_RenderDrawPoint</code>	<code>(SDL_Renderer *renderer, int x, int y)</code> Nakreslí bod $(x, y)$ .
<code>SDL_RenderDrawPoints</code>	<code>(SDL_Renderer *renderer, const SDL_Point *points, int count)</code> Nakreslí <code>count</code> bodov uložených v poli <code>points</code> . <code>SDL_Point</code> je typ, ktorý má dve premenné: <code>x</code> a <code>y</code> .
<code>SDL_RenderDrawLine</code>	<code>(SDL_Renderer *renderer, int x1, int y1, int x2, int y2)</code> Nakreslí čiaru z $(x_1, y_1)$ do $(x_2, y_2)$ .
<code>SDL_RenderDrawLines</code>	<code>(SDL_Renderer *renderer, const SDL_Point *points, int count)</code> Nakreslí lomenú čiaru, ktorá spája body v poli <code>points</code> .
<code>SDL_RenderDrawRect</code>	<code>(SDL_Renderer *renderer, const SDL_Rect *rect)</code>
<code>SDL_RenderFillRect</code>	<code>(SDL_Renderer *renderer, const SDL_Rect *rect)</code> Nakreslí, resp. vyplní obdĺžnik.
<code>SDL_RenderDrawRects</code>	<code>(SDL_Renderer *renderer, const SDL_Rect *rects, int count)</code>
<code>SDL_RenderFillRects</code>	<code>(SDL_Renderer *renderer, const SDL_Rect *rects, int count)</code> Nakreslí, resp. vyplní <code>count</code> obdĺžnikov uložených v poli <code>rects</code> .

---

Jednotlivé príkazy na kreslenie renderer nezobrazuje hned, ale robia sa postupne do kopie okna, ktorá je v pamäti. Dôvod je ten, že ak by si chcel napr. robiť animáciu a v každej snímke by si najprv prekreslil pozadie a potom nakreslil obdĺžnik, obraz by blikal. Takto sa stále zobrazuje ten istý obsah a až keď zavoláš funkciu `SDL_RenderPresent`, naraz sa zobrazí všetko.

<sup>7</sup>Parametre sú v poradí názov, pozícia na obrazovke  $(x, y)$ , rozmer (šírka×výška) a nastavenia; ich presný popis je na [https://wiki.libsdl.org/SDL2/SDL\\_CreateWindow](https://wiki.libsdl.org/SDL2/SDL_CreateWindow), nateraz nám stačia tieto.

<sup>8</sup>Renderer je oddelený od okna kvôli tomu, že niekedy chceš kresliť veci nie priamo do okna, ale do poľa v pamäti a to potom ich uložiť do súboru ako obrázok, alebo použiť ako textúru (napr. ak v 3D hre chceš mať zrkadlo). `SDL_CreateSoftwareRenderer` vytvorí renderer, ktorý namiesto okna používa pole v pamäti, takže na kreslenie potom môžeš používať rovnaké funkcie: renderer ako renderer.

<sup>9</sup>Tento zápis funguje v jazyku C od štandardu C99. Do jazyka C++ sa dostal až v revizii C++20. Rôzne komplilátory môžu ako default používať rôzne verzie štandardu, niekedy aj pomerne staré. Ak si chceš byť istý, že sa program kompiluje správne, treba dať komplilátoru (k prepínaču `-O3`) prepínač `-std=c++20`.

Funkcia, `SDL_Delay` iba počká zadaný počet milisekúnd.

Ak by si písal skutočný program, je slušné ošetrovať chyby. Z rôznych príčin sa nemusí podať napr. zapnutú knižnicu, otvoriť okno a pod. Každá z funkcií vracia nejakú hodnotu ("konštruktory" vrátia `nullptr`, volanie `SDL_Init()` vráti niečo <0 a pod.), ktorá znamená chybu. Po každom volaní by sa malo skontrolovať, či nenašla chyba, a ak áno, nejak na to zareagovať (napr. vypísať `cout << SDL_GetError() << endl`; a skončiť program).

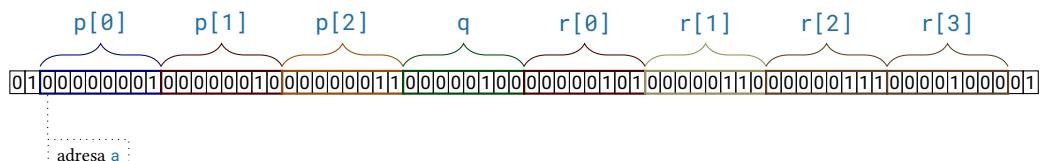
## Spracovanie vstupu: udalosti (events)

Kedž zavolás `SDL_Init`, okrem iného sa zapne spracovanie udalostí (events). Čokoľvek sa stane so vstupnými zariadeniami (pohne sa myš, stlačí sa kláves, ...), vytvorí sa<sup>10</sup> premenná typu `SDL_Event`, ktorá sa uloží do pola udalostí.

`SDL_Event` je zvláštny typ, takže na to, aby som ti ho opísal, ti najprv potrebujem vysvetliť typ `union`. Podobne ako `struct` je to zložený typ, na rozdiel od `struct` ale `union` všetky svoje premenné uloží na tú istú adresu. To znie ako dobrá blbosť, ale o chvíľu to snáď bude dávať zmysel. Dajme tomu, že mám

```
1 struct A {
2     uint8_t p[3], q, r[4];
3 };
4
5 int main() { A a; }
```

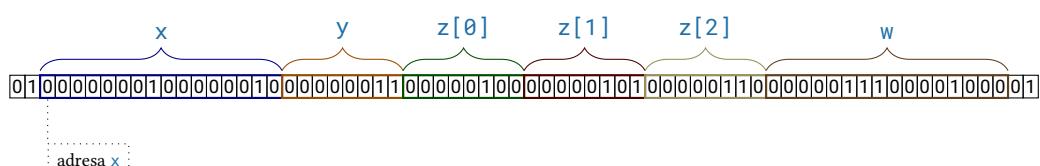
Premenná `a` môže vyzerať v pamäti nejak takto:



Podobne keby som mal

```
1 struct X {
2     uint16_t x;
3     uint8_t y, z[3];
4     uint16_t w;
5 };
6
7 int main() { X x; }
```

tak v pamäti môžem mať



Ak si iteraz spravím

<sup>10</sup>Na to existuje mechanizmus tzv. *prerušení*, takže sa to stane počas toho, ako tvoj program beží.

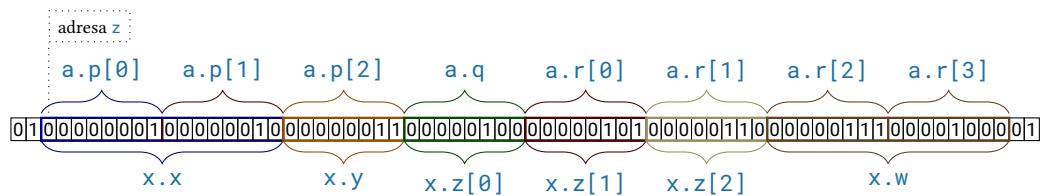
## 2D grafika: knižnica SDL

```

1 union Z {
2     A a;
3     X x;
4 };
5
6 int main() { Z z; }

```

bude v pamäti



Typ **Z** má dve premenné: **a**, ktorá je typu **A** a **x**, ktorá je typu **X**. Obidve sú ale uložené v tom istom mieste v pamäti, takže ak napíšeš `z.a.q=10;`, tak vzápäť bude platíť<sup>11</sup> `z.x.z[0]==10`.

Typ **SDL\_Event** je definovaný asi takto:

```

1 union SDL_Event {
2     uint32_t type;           // typ: podľa toho viem, kam sa mám pozerať
3     SDL_MouseMotionEvent motion; // pohyb myši
4     SDL_MouseButtonEvent button; // kliknutie
5     SDL_KeyboardEvent key;    // stlačenie klávesy
6     SDL_QuitEvent quit;      // zavretie okna zo systému
7     SDL_WindowEvent window;  // zmeny veľkosti, umiestnenia, ... okna
8     ....                      // atď, veľa ďalších typov
9 };

```

Všetky typy majú v pamäti na začiatku `uint32_t type`, takže ak mám premennú `SDL_Event e`; tak bez ohľadu na to, čo je v nej uložené, `e.type` bude identifikátor typu<sup>12</sup>. Jednotlivé typy majú tieto premenné:

```

1 struct SDL_MouseMotionEvent {
2     uint32_t type;           // SDL_MOUSEMOTION
3     uint32_t timestamp;      // čas, kedy udalosť vznikla
4     uint32_t windowID;       // okno, ktoré má fokus
5     uint32_t which;          // číslo myši alebo SDL_TOUCH_MOUSEID
6     uint8_t state;           // stav gombíkov
7     int32_t x;               // x-ová súradnica vzhľadom na okno
8     int32_t y;               // y-ová súradnica vzhľadom na okno
9     int32_t xrel;            // pohyb v smere osi x
10    int32_t yrel;            // pohyb v smere osi y
11 };

```

<sup>11</sup>Toto je definícia z jazyka C, kde sú veci zjednodušené tým, že tam neexistujú metódy, konštruktory a pod. V C++ veci začnú byť zložitejšie: typy **A** aj **X** by mohli mať konštruktory a rôzne metódy, takisto **Z** môže mať konštruktor a svoje metódy a potom je otázne, čo presne sa má stať, ak za napr. v premennej typu **Z** zavolá konštruktor **A** ale volá sa metóda **X**. V C++ je preto zakázané čítať z typu `union` inú premennú, ako tú, cez ktorú sa doňho zapisovalo. Celkovo použitie typu `union` v C++ tí okrem veľmi špeciálnych prípadov neodporúčam, ako vrávi známy citát: "Všetko môžem, ale nie všetko osož". Ak potrebujete schovať viac typov do jedného (napr. keď si navrhujete nejaký protokol na komunikáciu), je lepšie použiť `std::variant` zo štandardnej knižnice. My tu budeme používať typ `union` iba pri `SDL_Event` v jeho základnej verzii bez konštruktov a metód.

<sup>12</sup>Na to sú definované konštance ako `SDL_QUIT`, `SDL_MOUSEMOTION`, `SDL_KEYDOWN` a pod., takže typický test je napr. `if (e.type == SDL_QUIT) running=false;`

```

1 struct SDL_MouseButtonEvent {
2     uint32_t type;           // SDL_MOUSEBUTTONDOWN alebo SDL_MOUSEBUTTONUP
3     uint32_t timestamp;      // čas, kedy udalosť vznikla
4     uint32_t windowID;       // okno, ktoré má fokus
5     uint32_t which;          // číslo myši alebo SDL_TOUCH_MOUSEID
6     uint8_t button;          // stav gombíkov
7     uint8_t state;           // SDL_PRESSED alebo SDL_RELEASED
8     int32_t x;                // x-ová súradnica vzhľadom na okno
9     int32_t y;                // y-ová súradnica vzhľadom na okno
10 }

```

```

1 struct SDL_KeyboardEvent {
2     uint32_t type;           // SDL_KEYDOWN alebo SDL_KEYUP
3     uint32_t timestamp;      // čas, kedy udalosť vznikla
4     uint32_t windowID;       // okno, v ktorom je aktívny vstup
5     uint8_t state;           // SDL_PRESSED alebo SDL_RELEASED
6     uint8_t repeat;          // nenulové, ak vzniklo automatickým opakováním
7     SDLK_Sym keysym;        // klávesa
8 }

```

Stav gombíkov je bitové **OR** z konštánt `SDL_BUTTON_LMASK`, `SDL_BUTTON_MMASK` a `SDL_BUTTON_RMASK`. Typ `SDL_KeyboardEvent` má premennú typu `SDL_Keysym`. Ten je trochu zložitejší, lebo dáva informáciu o fyzickej klávese (scancode, t.j. čudlík klávesnice) aj o logickej (keycode t.j. znak).

```

1 struct SDL_Keysym {
2     SDL_Scancode scancode;    // kód fyzickej klávesy
3     SDLK_Sym sym;            // kód logickej klávesy
4     uint16_t mod;             // stlačené modifikátory
5 }

```

My budeme klávesnicu používať iba na zistenie, kedy sa stlačí nejaká klávesová kombinácia, čo sa dá napr. takto:

```

1 if (e.type == SDL_KEYDOWN) {
2     SDL_Keysym k = e.key.keysym;
3     if (k.sym == SDLK_q && ((k.mod & (KMOD_LCTRL | KMOD_RCTRL)) != 0))
4         running = false;
5 }

```

V tomto zápise `KMOD_LCTRL | KMOD_RCTRL` je číslo, ktoré má nastavené bity (pomocou **OR**) pre stlačenú ľavú a pravú klávesu Ctrl a `k.mod` má nastavené bity stlačených modifikátorov. Preto (`k.mod & (KMOD_LCTRL | KMOD_RCTRL)`) je číslo, ktoré má nastavené bity zodpovedajúce stlačeným Ctrl; ak je nenulové, nejaký Ctrl je stlačený. Predchádzajúci kus programu preto testuje, či je stlačené Ctrl+Q.

Len pre úplnosť dodám, že ak by si chcel načítavať text, ktorý používateľ píše, je lepšie namiesto `SDL_KEYDOWN` použiť udalosť `SDL_TEXTINPUT` s príslušným typom `SDL_TextInputEvent e.text`. Ak chceš, naopak, zisťovať, ktoré klávesy sú práve stlačené (napr. chceš niečo ovládať šípkami), je lepšie nečakať na udalosti a zavolať funkciu `SDL_GetKeyboardState`<sup>13</sup>. V online dokumentácii SDL sú aj zoznamy konštant pre `SDL_Keycode`<sup>14</sup> a `SDL_Keymod`<sup>15</sup>.

`SDL_WindowEvent` tu príliš rozoberať nebudem, použijeme ho iba na to, aby sme vedeli zareagovať, keď sa zmení veľkosť okna. Aby sme to celé dali dokopy, ostáva predstaviť funkciu na detekciu udalostí

<sup>13</sup>[https://wiki.libsdl.org/SDL2/SDL\\_GetKeyboardState](https://wiki.libsdl.org/SDL2/SDL_GetKeyboardState)

<sup>14</sup>[https://wiki.libsdl.org/SDL2/SDL\\_Keycode](https://wiki.libsdl.org/SDL2/SDL_Keycode)

<sup>15</sup>[https://wiki.libsdl.org/SDL2/SDL\\_Keymod](https://wiki.libsdl.org/SDL2/SDL_Keymod)

## 2D grafika: knižnica SDL

---

`SDL_WaitEvent(SDL_Event * event)`, ktorá zastaví program a čaká, kým nenastane nejaká udalosť, a potom ju uloží do premennej (ako parameter dostane pointer na ňu). Takže jednoduchý program `main.cc`<sup>16</sup>, ktorý čaká na stlačenie `Ctrl+Q` a pri každej udalosti prekreslí okno, by mohol vyzerať takto:

```
1 #include <SDL2/SDL.h>
2 #include <iostream>
3 using namespace std;
4
5 SDL_Window* window = nullptr;
6 SDL_Renderer* renderer = nullptr;
7
8 int main() {
9     int ww = 800, wh = 600;
10    SDL_Init(SDL_INIT_VIDEO | SDL_INIT_TIMER | SDL_INIT_AUDIO);
11    window = SDL_CreateWindow("prvy_pokus", SDL_WINDOWPOS_CENTERED,
12                               SDL_WINDOWPOS_CENTERED, ww, wh,
13                               SDL_WINDOW_RESIZABLE | SDL_WINDOW_SHOWN);
14    renderer = SDL_CreateRenderer(window, -1, SDL_RENDERER_ACCELERATED);
15    SDL_SetRenderDrawBlendMode(renderer, SDL_BLENDMODE_BLEND);
16
17    bool running = true;
18    SDL_Rect r{.x = 100, .y = 100, .w = 100, .h = 100};
19
20    while (running) {
21        SDL_Event e;
22        SDL_WaitEvent(&e); // počkaj na udalosť a ulož ju do premennej e
23
24        if (e.type == SDL_WINDOWEVENT && e.window.event == SDL_WINDOWEVENT_SIZE_CHANGED) {
25            ww = e.window.data1; // toto sú premenné z typu SDL_WindowEvent,
26            wh = e.window.data2; // v ktorých je nová veľkosť okna
27            cout << "nova_velkosť_okna:" << ww << "x" << wh << endl;
28        } else if (e.type == SDL_QUIT)
29            running = false;
30        else if (e.type == SDL_KEYDOWN) {
31            SDL_KeySym k = e.key.keysym;
32            cout << "stlacene:" << SDL_GetKeyName(k.sym) << ", mod" << k.mod << endl;
33            if (k.sym == SDLK_q && ((k.mod & (KMOD_LCTRL | KMOD_RCTRL)) != 0))
34                running = false;
35        }
36        // prekresli okno ako minule
37        SDL_SetRenderDrawColor(renderer, 250, 180, 0, 255);
38        SDL_RenderClear(renderer);
39        SDL_SetRenderDrawColor(renderer, 20, 30, 180, 255);
40        SDL_RenderFillRect(renderer, &r);
41        SDL_RenderPresent(renderer);
42    } // koniec while-cyklu, zatvoríme okno a končíme
43
44    SDL_DestroyWindow(window);
45    SDL_Quit();
46 }
```

Tu si môžeš všimnúť typickú štruktúru grafických programov: v premenných máme uložené, čo sa má kam zobraziť. V cykle vždy spracujeme udalosť, aktualizujeme premenné a nakoniec prekreslíme obrazovku.

**Úloha 131.** Napiš program, v ktorom bude používateľ môcť modrý štvorec premiestňovať myšou: keď klikne myšou na štvorec, prejde sa do režimu ťahania, v ktorom štvorec sleduje kurzor, až kým používateľ nepustí gombík myši.

<sup>16</sup><https://github.com/pocestny/programovanie/raw/master/materialy/sdl/02/main.cc>

Zároveň treba zabezpečiť, aby štvorec nikdy netrčal mimo okna a pri zmene veľkosti okna sa vždy posunul tak, aby bol celý viditeľný.

Tu urobím ešte jednu odbočku a ukážem ti príkaz `switch`. Doteraz som ti ho neukázal, lebo nie je na písanie programov nevyhnutný, ale ak si budeš niekedy pozerať programy, hlavne také, ktoré používajú knižnice ako SDL, určite sa s ním stretneš. Všimni si, že v predchádzajúcim programe je pomerne typická konštrukcia

```

1 if (x == 0) kvak();
2 else if (x == 1) kvik();
3 else if (x == 2) kvok();
4 else if (x == 3) kvuk();
5 ...

```

keď veľakrát za sebou testujem tú istú premennú na rôzne hodnoty. Príkaz `switch(x)` bude nasledovaný zloženým príkazom, ktorý obsahuje tzv. *návestia case* pre rôzne hodnoty. Zoberme si funkciu

```

1 void f(int x) {
2     switch (x) {
3         case 0:
4             cout << "0\u";
5         case 1:
6             cout << "1\u";
7         case 2:
8             cout << "2\u";
9         case 3:
10            cout << "3\u";
11        case 4:
12            cout << "4\u";
13        case 5:
14            cout << "5\u";
15            break;
16        case 6:
17            cout << "6\u";
18        case 7:
19            cout << "7\u";
20        case 8:
21            cout << "8\u";
22        case 9:
23            cout << "9\u";
24    }
25 }
26

```

Ked zavolám `f(x)`, v príkaze `switch(x)` sa vyhodnotí výraz `x` a začne sa vykonávať zložený príkaz od toho `case`, ktoré má rovnakú hodnotu až do konca. Príkaz `break` v tomto prípade preruší vykonávanie. Preto napr. `f(3)` začne vykonávať príkazy od `case 3:` a skončí tesne pred `case 6:`, lebo tam nazarí na `break`. Program

```

1 int main() {
2     for (int i=0; i<10; i++) {
3         f(i);
4         cout << endl;
5     }
6 }

```

by napísal

## 2D grafika: knižnica SDL

```
0 1 2 3 4 5
1 2 3 4 5
2 3 4 5
3 4 5
4 5
5
6 7 8 9
7 8 9
8 9
9
```

Koniec odbočky, vráťme sa ku grafike s SDL. Máme základnú schému programu, ktorá vyzerá zhruba takto:

```
1 int main() {
2     SDL_Init(...);
3     ... // nastaviť všetko, čo na začiatku treba
4
5     bool running = true;
6     while (running) { // kým netreba skončiť
7
8         SDL_Event e;
9         SDL_WaitEvent(&e); // počkaj na udalosť
10
11        switch (e.type) { // spracuj rôzne typy udalostí
12            case SDL_QUIT:
13                running = false;
14                break;
15            case SDL_MOUSEMOTION:
16                ...
17                break;
18            ...
19        }
20
21        SDL_SetRenderDrawColor(renderer, 250, 180, 0, 255); // zmaž pozadie
22        SDL_RenderClear(renderer); // nakresli všetko, čo treba
23        ...
24        SDL_RenderPresent(renderer);
25    }
26}
```

Teraz chcem upraviť program z Úlohy 131 tak, aby modrý štvorec pulzoval: plynule menil farbu aj veľkosť. Aby som si udržal poriadok v programe, budem si držať všetko, čo potrebujem na vykreslenie štvorca, v jednej premennej, ktorá bude mať metódu `render(SDL_Renderer *)` (t.j. ako parameter dostane pointer na renderer, a ním sa vykreslí). Na zmenu farby použijem triedu `Gradient` z Úlohy 119. Pretože budem chcieť meniť veľkosť štvorca, budem si pamätať jeho pozíciu `cx, cy` a rozmery (`w×h`). Ešte si môžem vyrobiť pomocnú funkciu `contains`, ktorá mi povie, či bod so súradnicami `x, y` leží vnútri štvorca.

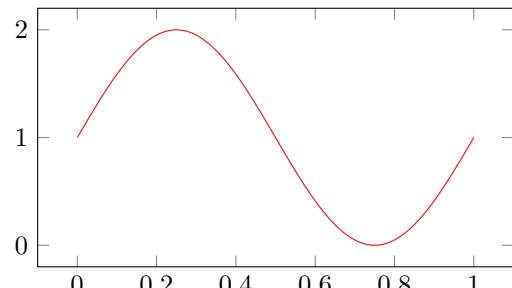
```
1 struct Stvorec {
2     int cx = 100, cy = 100, w = 100, h = 100;
3     Gradient gradient;
4     Stvorec(const Gradient& _g) : gradient{_g} {} // konštruktor dostane gradient
5     void render(SDL_Renderer* renderer); // zobrazovacia funkcia
6     bool contains(int x, int y) {
7         return (abs(x - cx) < w / 2 && abs(y - cy) < h / 2);
8     }
9 }
```

Na vykreslenie štvorca budem potrebovať hodinky, aby mohol pravidelne pulzovať. Môžem použiť tie z kapitoly 30, ale SDL má aj vlastnú implementáciu `uint32_t SDL_GetTicks()`, ktorá vráti počet milisekúnd od spustenia programu. Keď potrebujem niečo pravidelne opakovať každých `s` milisekúnd a napíšem `SDL_GetTicks() % s`, tak dostanem číslo, ktoré sa pravidelne mení v rozsahu 0 až `s` každú milisekundu. Keď potom napíšem `(double)(SDL_GetTicks() % s) / (double)s`, budem mať číslo, ktoré sa pravidelne mení medzi 0 a 1 za `s` milisekúnd. Renderovacia funkcia by mohla vyzerať takto:

```

1 void Stvorec::render(SDL_Renderer* renderer) {
2     uint32_t now = SDL_GetTicks(); // zistí čas
3
4     // nastav farbu podľa gradientu (potrebujem ho mať symetrický)
5     Vec3 clr = gradient((double)(now % 10000) / 10000.0L); // jeden cyklus bude 10s
6     // renderer chce uint8_t, ja mám reálne čísla medzi 0..1, takže prenásobím
7     SDL_SetRenderDrawColor(renderer, 255 * clr.x, 255 * clr.y, 255 * clr.z, 255);
8
9     SDL_Rect r; // obdĺžnik, ktorý budeme renderovať
10    double a = 0.8 * (1.0 + sin(2.0 * M_PI * (double)(now % 340) / 340.0L));
11    r.x = cx - w / 2 + a;
12    r.y = cy - h / 2 + a;
13    r.w = w - 2 * a;
14    r.h = h - 2 * a;
15
16    SDL_RenderFillRect(renderer, &r);
17 }
```

Na riadku 10 rátam hodnotu `a`, o ktorú sa zmenšia rozmerы štvorca. Aby sa menili plynule, použil som funkciu `sin` (treba `#include <cmath>`), ktorá robí "vlnku" na intervalu  $[0, 2\pi]$ . Keď mám  $x$  z intervalu  $[0, 1]$ , tak na vlnku potrebujem funkciu  $\sin(2\pi x)$ . Tá má ale hodnoty od  $-1$  po  $1$ , takže  $1 + \sin(2\pi x)$  bude mať hodnoty od  $0$  po  $2$  a bude vyzerať takto:



Hlavný cyklus ale teraz musím urobiť inak. Keďže `SDL_WaitEvent` čaká na udalosť, štvorec by sa menil iba kým by si hýbal myšou (alebo iným spôsobom vyrábal udalosti). Existuje ale podobná funkcia `SDL_PollEvent`, ktorá na nič nečaká, ale ak je nejaká udalosť pripravená v poli udalostí, skopíruje ju do zadaného pointra a vráti `true`, inak vráti `false`. Keďže udalostí môže nastaviť aj viacero naraz, typický hlavný cyklus vyzera takto:

```

1 while (running) {
2     SDL_Event e;
3
4     while (SDL_PollEvent(&e)) { // spracuj všetky udalosti, kým sú
5         switch (e.type) {
6             case SDL_QUIT:
7                 running = false;
8                 break;
9             case SDL_MOUSEMOTION:
10                ...
11                break;
12                ...
13            }
14        }
15 }
```

```
16     // prekresli okno  
17 }
```

Posledná vec, ktorú treba spomenúť je efektivita. Takto napísaný cyklus beží stále dokola na plný výkon počítača. To nie je ideálne, hlavne na notebooku to veľmi rýchlo minie baterku. Preto je slušné obmedziť počet snímkov za sekundu (*frames per second, fps*). Na začiatku vykreslovania si odmeriam čas pomocou [SDL\\_GetTicks](#). Rovnako urobím aj na konci a zistím, kolko milisekúnd trvalo vykreslovanie. Ak chcem mať fixné fps, vykreslif jeden snímok má trvať  $1000/\text{fps}$  milisekúnd. Ak som bol s vykreslovaním hotový skôr, počkám potrebný počet milisekúnd pomocou [SDL\\_Delay](#). Skús si to, čo sme tu teraz hovorili, naprogramovať:

**Úloha 132.** Uprav Úlohu 131 tak, aby štovrec pulzoval a menil farby.

## Práca s obrázkami

Pomocou SDL sa dajú priamo vykreslovať útvary zložené z čiar, bodov a obdĺžnikov, ale hlavná sila SDL je v práci s obrázkami. V knižnici sú dva hlavné typy, ktoré vedia s obrázkami pracovať [SDL\\_Surface](#) a [SDL\\_Texture](#).

[SDL\\_Surface](#) je vlastne pole pixelov v pamäti. Je to trošku komplikovanejšie, lebo pixely môžu byť uložené rôzne (napr. RGB, RGBA, ako odtiene sivej, ako indexy do palety, ...). Premenná [format](#) popisuje, akým spôsobom treba k pixelom pristupovať.

```
1 struct SDL_Surface {  
2     SDL_PixelFormat *format;      // spôsob uloženia pixelov  
3     int w, h;                   // rozmery  
4     int pitch;                 // dĺžka riadka v bytoch  
5     void *pixels;              // pole pixelov  
6     void *userdata;            // sem si môžeš zapísat', čo chceš  
7     SDL_Rect clip_rect;        // mimo tohto obdĺžnika sa nekreslí  
8     ...                          // atď.  
9 };
```

Naproti tomu [SDL\\_Texture](#) je tiež pole pixelov, ale kde je uložené, to závisí od konkrétneho systému. Ak to totiž hardvér podporuje, pole sa umiestni do pamäti grafickej karty. To znamená, že z programu nemôžeš priamo pristupovať k jednotlivým pixelom, ale prekreslovanie je oveľa rýchlejšie, lebo si to grafická karta vie zariadiť sama. Dáta z pamäte vieš poslať na kartu pomocou [SDL\\_CreateTextureFromSurface](#)), ale naopak to nejde.

SDL samotná podporuje iba obrázkový formát [.bmp](#). Ak chceš pracovať aj s inými, treba použiť rozšírenie [SDL\\_Image](#). To znamená v programe pridať `#include <SDL2/SDL_image.h>`, v [Makefile](#) pridať pri linkovaní `-lSDL2_image` a možno rozšírenie aj nainštalovať (napr. `sudo apt-get install libSDL2-image-dev`).

Pri písaní programu treba po zavolaní [SDL\\_Init](#) zvoliť, aké formáty sa majú používať, napr. [IMG\\_Init\(IMG\\_INIT\\_PNG\)](#); a na konci programu, pred volaním [SDL\\_Quit\(\)](#) treba zavolať [IMG\\_Quit\(\)](#). Ak chceš, aby správne fungovali transparentné pixely (*alpha channel*), treba predtým zapnúť [SDL\\_SetRenderDrawBlendMode\(renderer, SDL\\_BLENDMODE\\_BLEND\)](#);

Na prečítanie obrázku zo súboru sú dve funkcie, ktoré robia presne to, čo hovorí definícia (t.j. zoberie súbor s daným menom a vyrábí textúru alebo surface):

```
1 SDL_Surface * IMG_Load(const char *file);  
2 SDL_Texture * IMG_LoadTexture(SDL_Renderer *renderer, const char *file);
```

Ked už máš textúru v pamäti karty (t.j. v premennej typu [SDL\\_Texture](#)), dá sa vykresliť pomocou funkcií

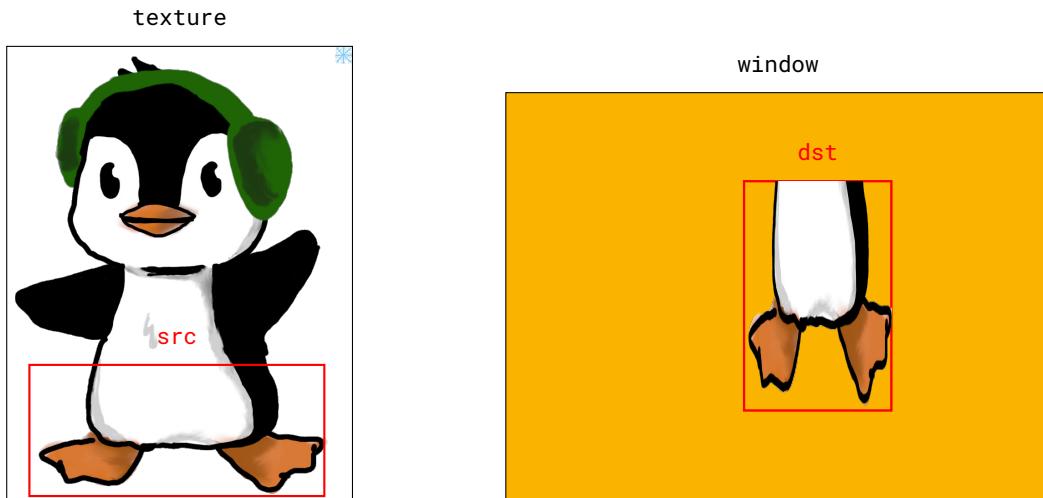
```
1 SDL_RenderCopy( SDL_Renderer *, SDL_Texture *, SDL_Rect *src, SDL_Rect *dst );  
2  
3 SDL_RenderCopyEx( SDL_Renderer *, SDL_Texture *, SDL_Rect *src, SDL_Rect *dst,  
4                     double angle, SDL_Point *center, int flip);
```

Čo presne robia? Prvé dva parametre sú jasné: renderer, ktorý má vykresľovať a textúra, ktorú treba vykresliť. Druhé dva parametre sú pointre na obdĺžniky. Z textúry sa zoberie obdĺžnik `src` a vyrenderuje sa na pozíciu `dst` (ak treba, tak sa naškáluje<sup>17</sup>)

Napríklad ak mám textúru s obrázkom tučniaka<sup>18</sup>, načítam si ju pomocou

```
1  SDL_Texture* penguin = IMG_LoadTexture(renderer, "penguin.png");
```

Ak z nej potom vyberiem obdĺžnik `src` a vyrenderujem ho pomocou `SDL_RenderCopy` do okna na obdĺžnik `dst`, dotsanom niečo takéto:



Keby som chcel vykresliť celého tučniaka do stredu okna (celý program je `tu`<sup>19</sup>), budem mať

```
1  SDL_SetRenderDrawColor(renderer, 250, 180, 0, 255);
2  SDL_RenderClear(renderer);
3  // umiestnenie tučniaka v textúre
4  SDL_Rect src{.x = 0, .y = 0, .w = 722, .h = 1002};
5
6  // ww a wh sú rozmery okna, chcem sa naškálovať na 90% menšieho z nich
7  double scale = 0.9 * min((double)ww / 722.0, (double)wh / 1002.0);
8
9  int nw = scale * 722, nh = scale * 1002;           // rozmery tučniaka v okne
10 SDL_Rect dst{.x = (ww - nw) / 2, .y = (wh - nh) / 2, .w = nw, .h = nh};
11 SDL_RenderCopy(renderer, penguin, &src, &dst);
12 SDL_RenderPresent(renderer);
```

Druhá funkcia, `SDL_RenderCopyEx` je podobná, iba navyše môže obrázok aj zrotovať okolo bodu `center` o `angle` stupňov v smere hodinových ručičiek a zrkadlovo prevrátiť. Parameter `flip` je bitové OR z konštánt `SDL_FLIP_NONE`, `SDL_FLIP_HORIZONTAL` a `SDL_FLIP_VERTICAL`. Ak namiesto niektorého obdĺžnika zadám `nullptr`, berie sa celá textúra, resp. celé okno.

Ak už surface, resp. textúru nepotrebuješ, treba ručne zavolať “deštruktor” `SDL_FreeSurface(surf)` resp. `SDL_DestroyTexture(txt)`.

**Úloha 133.** Textúra tučniaka má od pozicie (722, 0) snehovú vločku rozmerov  $40 \times 40$ . Napíš program, ktorý vykreslí tučniaka v strede okna a za ním bude snežiť (t.j. bude tam veľa snežových vločiek padáť zhora dolu).

<sup>17</sup>Na presnejšie, ale pomalšie škálovanie sa dá zavolať `SDL_SetHint(SDL_HINT_RENDER_SCALE_QUALITY, "2")`.

<sup>18</sup><https://github.com/pocestny/programovanie/raw/master/materialy/sdl/03/penguin.png>

<sup>19</sup><https://github.com/pocestny/programovanie/raw/master/materialy/sdl/03/main.cpp>

**Úloha 134.** Na stránke [Spritesheet Character Generator<sup>20</sup>](https://sanderfrenken.github.io/Universal-LPC-Spritesheet-Character-Generator) si vytvor postavu a stiahni si [.png](#) obrázok s rozfázaným pohybom, tzv. spritesheet. Urob program, v ktorom postava chodí po obrazovke a ovláda<sup>21</sup> ju šípkami.

## Písanie textu, fonty

Na vykreslovanie textu potrebuješ font, t.j. (spravidla vektorový) popis, ako vyzerajú jednotlivé písmená. Veľmi častý formát na ukladanie fontov je **TrueType**, súbory majú príponu **.ttf**. Je veľmi veľa miest, kde sa dajú nájsť a stiahnuť voľne prístupné súbory **.ttf**. Na vykreslovanie bude treba, aby si si svoj oblúbený font našiel a dal do pracovného adresára. Ja budem používať font **Montserrat<sup>22</sup>**.

Na prácu s TrueType fontami je v SDL samostatné rozšírenie, ktoré treba, podobne ako prácu s obrázkami inicializovať: pridať `#include <SDL2/SDL_ttf.h>`, v **Makefile** pri linkovaní pridať `-lSDL2_ttf` a možno ho predtým zvlášť nainštalovať, napr. `sudo apt-get install libSDL2-ttf-dev`. Rovnako, podobne ako pri rozšírení **IMG**, treba na začiatku programu zavolať **TTF\_Init()** a na konci **TTF\_Quit()**. Zoznam všetkých funkcií v rozšírení **TTF** je tu<sup>23</sup>.

Prvá vec, ktorú treba s fontom urobiť, je nahrať ho do pamäte volaním

```
1 TTF_Font *font = TTF_OpenFont("Montserrat-Bold.ttf", 50);
```

Prvý parameter je meno súboru, druhý je veľkosť. Keď už font netreba, dá sa odstrániť volaním "deštruktora" **TTF\_CloseFont(font)**. Keď mám font, môžem vyrenderovať text vnejakej farbe pomocou

```
1 SDL_Surface * TTF_RenderUTF8_Bladed(TTF_Font *font, const char *text, SDL_Color fg);
```

Podobných funkcií je niekoľko, líšia sa tým, ako je zadaný text (v našom prípade mám **UTF-8**, čo je bežné kódovanie pre texty s diakritikou) a ako kvalitne (tým pádom aj pomaly) sa má renderovať (v našom prípade má brať do úvahy transparentnosť). Dostaneme bitmapový obrázok v pamäti (**SDL\_Surface** má, okrem iného, premenné **w** a **h**, ktoré udávajú rozmery. Bez toho, aby sme text naozaj renderovali, vieme zistiť jeho rozmery volaním **TTF\_SizeText(font, text, &w, &h)**). Na vykreslenie potom musíme obrázok poslať na grafickú kartu, t.j. vyrobiť **SDL\_Texture**, a vykresliť ho ako každú inú textúru. Ak chceme text renderovať veľakrát, vyrobený **SDL\_Surface** a **SDL\_Texture** si môžeme odložiť. Jednoduchá trieda na písanie textov by mohla vyzerať takto:

```
1 struct Text {
2     SDL_Surface *surf;
3     SDL_Texture *txt;
4     SDL_Rect box;
5
6     Text(SDL_Renderer *renderer, TTF_Font *font, const char *text,
7           SDL_Color color) {
8         surf = TTF_RenderUTF8_Bladed(font, text, color);
9         box.h=surf->h;
10        box.w=surf->w;
11        txt = SDL_CreateTextureFromSurface(renderer, surf);
12    }
13
14    void render(SDL_Renderer *renderer, int x, int y) {
15        box.x=x;
```

<sup>20</sup><https://sanderfrenken.github.io/Universal-LPC-Spritesheet-Character-Generator>

<sup>21</sup>Ako som spomíнал, je lepšie nespoliehať sa na udalosti, ale priamo kontrolovať, ktoré klávesy sú stlačené. Ak zavoláš `const uint8_t *keys = SDL_GetKeyboardState(nullptr);` budeš mať pole, v ktorom je pre každý scancode jednotka, ak je daná klávesa stlačená a 0 ak nie je. Preto vieš zisťovať napr. `if (keys[SDL_SCANCODE_DOWN] == 1) {...}`.

<sup>22</sup><https://github.com/JulietaUla/Montserrat/tree/master>

<sup>23</sup>[https://wiki.libsdl.org/SDL2\\_ttf/CategoryAPI](https://wiki.libsdl.org/SDL2_ttf/CategoryAPI)

```

16     box.y=y;
17     SDL_RenderCopy(renderer, txt, nullptr, &box);
18 }
19
20 ~Text() {
21     SDL_DestroyTexture(txt);
22     SDL_FreeSurface(surf);
23 }
24 };

```

Použili by sme ju napr.

```

1 TTF_Font *font = TTF_OpenFont("Montserrat-Bold.ttf", 50);
2 Text t(renderer, font, "môj\u00fat\u00e1xt", SDL_Color{.r=10,.g=80,.b=150});
3 ...
4 while (running) {
5     ...
6     t.render(renderer,x,y);
7     ...
8 }

```

**Úloha 135.** Napíš program, ktorý prečíta zo vstupu text a potom ho donekonečna v okne skroluje zľava doprava (keď zájde za pravý okraj, znova vyjde spoza ľavého).

Nielen pri renderovaní textu sa hodí mať kontrolu nad cieľovým obdĺžnikom, do ktorého sa vykresluje. Dá sa to sice urobiť pri volaní `SDL_RenderCopy`, ale ak máš viacero vykreslovaní za sebou, je príjemné zavolať `SDL_RenderSetClipRect(SDL_Renderer *, SDL_Rect *)`. Tým nastavíš orezávanie (*clipping*) a všetky ďalšie príkazy budú vykreslovať iba do zadaného obdĺžnika. Keď potom zavoláš `SDL_RenderSetClipRect`, kde druhý parameter bude `nullptr`, orezávanie sa vypne a vykreslovať sa bude opäť všetko.

## Ako pridať zvuk

Aj keď sa to možno nezdá, prehrávanie zvuku môže byť pomerne komplikovaná záležitosť. SDL vie elegančne zkryť všetky technické ťažkosti použitím ďalšieho (už posledného) rozšírenia: `SDL_Mixer`. Nainštaluje sa rovnako, ako `SDL_Image` a `SDL_TTF`: do programu sa pridá `#include <SDL2/SDL_mixer.h>`, v `Makefile` sa prilinkuje `-lSDL2-mixer` (a, samozrejme, nainštaluje sa príslušná knižnica, napr. `libsdl2-mixer-dev`).

`SDL_Mixer` pridáva dva hlavné typy pre zvuk: `Mix_Chunk` a `Mix_Music`. Rozdiel medzi nimi je v tom, že `Mix_Chunk` sa celý dekóduje do pamäte, kým `Mix_Music` sa priebežne dekóduje pri prehrávaní z disku. `Mix_Chunk` na jednej strane zaberá miesto v pamäti, na druhej strane sú operácie ako zastavenie, skok na začiatok a pod. rýchlejšie. `Mix_Chunk` sa preto používa na krátke zvukové efekty, kým napr. na hudbu na pozadí by si radšej použil `Mix_Music`.

`Mix_Chunk` sa prehráva v stopách (*channels*), pričom v jednej stope môže naraz hrať jeden zvuk. Môžeš si vyrobiť viacero stôp, aby mohlo hrať viacero efektov naraz, ale `Mix_Music` vždy hrá najviac jedna.

Na začiatku treba inicializovať knižnicu napr.

```

1 Mix_OpenAudio(44100, MIX_DEFAULT_FORMAT, 2, 1024);

```

Pre vysvetlenie (ale asi to nebudeš potrebovať meniť): prvý parameter je vzorkovacia frekvencia, druhý je formát, tretí je počet kanálov (dva pre štandardné dvokjanálové stereo) a posledný parameter je veľkosť interného buffera pri prehrávaní. Po tom, ak inicializuješ audio, chceš vyrobiť stopy, v ktorých budeš prehrávať zvuky:

```

1 Mix_AllocateChannels(nChannels);

```

## 2D grafika: knižnica SDL

---

Ďalšie použitie je pomerne priamočiare, tu je zopár funkcií, ktoré môžeš chcieť použiť (kompletný zoznam je tu<sup>24</sup>.

---

```
Mix_Chunk * Mix_LoadWAV(const char *file)
```

```
Mix_Music * Mix_LoadMUS(const char *file)
```

“Konštruktor”: prečíta zvuk zo súboru

---

```
int Mix_PlayChannel(int channel, Mix_Chunk *chunk, int loops)
```

```
int Mix_PlayMusic(Mix_Music *music, int loops)
```

Začne hrať zvuk. Pre `chunk` parameter `channel` udáva stopu, kde sa má zvuk hrať. Ak je `channel == -1` začne hrať na prvej volnej stope. `loops` je počet opakovaní: 0 znamená *zahraj raz a skonči*, -1 znamená *stále opakuj*

---

```
int Mix_MasterVolume(int volume)
```

```
int Mix_Volume(int channel, int volume)
```

```
int Mix_VolumeMusic(int volume)
```

Nastaví hlasitosť v rozmedzí 0 a `MIX_MAX_VOLUME`. Ak je parameter `volume` záporný, funkcia vráti súčasnú hlasitosť a nič nezmení.

---

```
void Mix_Pause(int channel)
```

```
void Mix_PauseAudio(int pause_on)
```

```
void Mix_PauseMusic(void)
```

```
void Mix_Resume(int channel)
```

```
void Mix_ResumeMusic(void)
```

Preruší a pokračuje v prehrávaní.

---

```
double Mix_GetMusicPosition(Mix_Music *music)
```

```
void Mix_RewindMusic(void)
```

```
int Mix_SetMusicPosition(double position)
```

Zistí pozíciu (v sekundách), pretočí na začiatok, prípadne skočí na danú pozíciu.

---

```
int Mix_HaltMusic(void)
```

```
int Mix_HaltChannel(int channel)
```

Zastaví prehrávanie. Ak je `channel == -1`, zastavia sa všetky stopy.

---

```
void Mix_HookMusicFinished(void (SDLCALL *music_finished)(void))
```

```
void Mix_ChannelFinished(void (SDLCALL *channel_finished)(int channel))
```

Pointer na funkciu (ako v kapitole 25), ktorá sa spustí, keď hudba dohrá. Môže napr. spustiť ďalšiu skladbu volaním `Mix_PlayMusic`

---

```
void Mix_FreeChunk(Mix_Chunk *chunk)
```

```
void Mix_FreeMusic(Mix_Music *music)
```

“Deštuktory”.

---

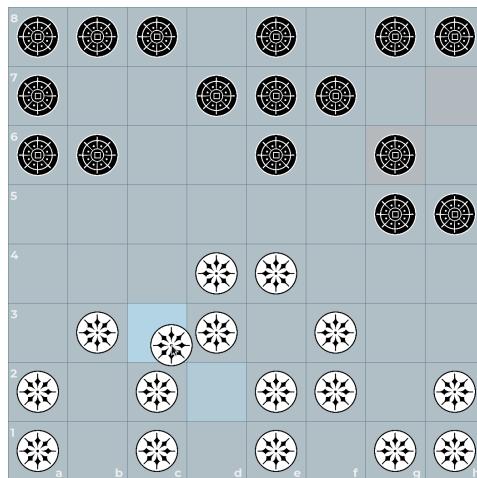
## Grafické prostredie pre Breakthrough

Teraz je už všetko pripravené na to, aby si napísal prvú verziu grafického prostredia pre Breakthrough. Chceme, aby to fungovalo takto: v zadnom obdĺžniku niekde v okne sa zobrazuje šachovnica (reaguje na

---

<sup>24</sup>[https://wiki.libsdl.org/SDL2\\_mixer/CategoryAPI](https://wiki.libsdl.org/SDL2_mixer/CategoryAPI)

zmenu veľkosti okna) s rozohratou pozíciou. Hráč môže kedykoľvek chytiť myšou nejaký svoj kameň a pohybovať ním (možno sa zvýrazňuje políčko, odkiaľ kameň zobraza a kde je práve teraz). Keď kameň pustí, vráti sa na pôvodné miesto. Ak ale kameň pustí vtedy, keď je na ťahu a vyrobí sa prípustný ťah, tento ťah sa vykoná a aktualizuje sa pozícia. Potom sa zistí ťah počítača a začne sa animovať plynulý presun kameňa (ťah sa tiež môže zvýrazniť). Keď kameň príde na svoje miesto, aktualizuje sa pozícia a opäť je na ťahu hráč. Screenshot by mohol vyzerať asi takto:



Naprogramujme to tak, že budeme mať triedu `BoardView`, ktorá sa bude starať o vykreslovanie. Bude si pamätať obdĺžnik `SDL_Rect screen`, do ktorého sa bude vykreslovať. Pre pohodlie si môže pamätať aj `renderer`, keďže ju nikdy nebudem chcieť renderovať inam. Premenná `screen` sa bude meniť volaním `resize`. To je preto, že pri zmene veľkosti si možno budeš chcieť prepočítať nejaké ďalšie veci, ktoré s veľkosťou okna súvisia. Napríklad rozmer šachovnice môžeš nastaviť ako menší z dvoch rozmerov okna a pod. Bude si pamätať aj `obrázok`<sup>25</sup> hracích kameňov, aktuálnu polohu myši, či práve používateľ presúva kameň, či sa práve animuje a ak áno tak odkiaľ a kam atď. Hodia sa aj pomocné funkcie, ktoré pre políčko šachovnice vrátia pozíciu na obrazovke a naopak. Jadro triedy by mohlo vyzerať takto:

```

1 struct BoardView {
2     SDL_Renderer* renderer;
3     Board board;           // aktuálna pozícia
4     SDL_Texture* pieces;   // textúra s bielym a čiernym kameňom
5     uint8_t human;         // človek je Biely(0) / Čierny (1)
6     bool flippedBoard;    // zobrazujem šachovnicu z pohľadu čierneho
7
8     bool dragging, animating; // ťahá hráč / počítač
9     Square dragOrig;        // odkiaľ hráč začal ťah
10    Move aMove;             // počítačový ťah, ktorý sa animuje
11    const int animLen = 200; // dĺžka animácie (ms)
12    uint32_t animEndTime;   // kedy skončí aktuálna animácia
13
14    SDL_Rect screen;        // obdĺžnik, kam sa vykreslujem
15    int mouseX, mouseY;    // poloha myši
16    ...                     // rôzne veci, ktoré potrebujete pri vykreslovani
17
18    BoardView(SDL_Renderer* r); // konštruktor
19    void resize(SDL_Rect scr); // nastaví všetko potrebné pri zmene okna
20    void render();           // spustí animáciu ťahu protihráča
21    void operator+=(const Move& m); // spustí animáciu ťahu protihráča
22

```

<sup>25</sup><https://github.com/pocestny/programovanie/raw/master/materialy/sdl/pieces.png>

## 2D grafika: knižnica SDL

---

```
23 void onMouseMotion(SDL_MouseMotionEvent& e);
24 void onMouseDown(SDL_MouseButtonEvent& e);
25 Move onMouseUp(SDL_MouseButtonEvent& e);
26 ...
27 };
```

V hlavnom cykle sa pri spracovaní eventov zavolajú funkcie `onMouse...`, ktoré nastavia `mouseX`, `mouseY` a ošetria prípadný začiatok a koniec ťahania. Špeciálne ak sa v `onMouseUp` urobí ťah, tento sa vráti (inak vracia nejaký neprípustný ťah ako zarážku). V hlavnom programe sa tento ťah podhodí `RandomPlayer` z úlohy 125, ktorý nájde ťah. Ten sa späť pošle do zobrazovača cez `operator+`: spustí sa animácia a keď skončí, aktualizuje sa pozícia. Kus hlavného programu by mohol vyzerať takto (bv je `BoardView`, plr je `RandomPlayer`):

```
1 case SDL_MOUSEBUTTONDOWN:
2     if (bv.onMouseUp(e.button).from.legal() && // bv.onMouseUp vráti Move(from,to)
3         bv.board.winner() == Empty) {           // ak je to legálny ťah a hra neskončila
4         Move m = plr.findMove(bv.board);        // zisti ťah hráča
5         bv += m;                                // aktualizuj BoardView
6     }
7     break;
```

**Úloha 136.** Naprogramuj grafické rozhranie, ako sme si ho tu opísali.

Predpokladám, že ťa zarazilo, prečo som ti v minulej časti pri programovaní grafického prostredia povedal, aby sa hralo proti `RandomPlayer`, keď už máme oveľa lepšieho `AlphaBetaPlayer`. Možno si aj skúsil `AlphaBetaPlayer` použiť. Ak nie, urob to teraz. Čo si zistil? V programe sa funkcia `plr.findMove()` sa volá pri spracovaní udalostí v hlavnom cykle. To ale znamená, že ak `plr.findMove()` trvá dlho, žiadne iné udalosti sa počas toho času nespracovávajú a program sa na tú dobu "zasekne" (nereaguje na pohyby myšou, neprekresluje okno, nič...).

Aby sme to opravili, bolo by treba `findMove` rozdeliť na krátke kúsky a vždy v jednej iterácii hlavného cyklu zavolať len jeden kúsok, napr. takto:

```

1 struct AlphaBetaPlayer {
2     struct MoveValue {...}
3
4     std::mt19937 rnd;
5     bool thinking, finished;
6
7     AlphaBetaPlayer();
8
9     void startThinking(const Board& b);
10    void step();
11    Move getResult();
12
13    float eval(const Board& b);
14 };

```

V programe by sme namiesto `findMove` zavolali funkciu `startThinking`. Tam sa nastaví `thinking=true` a `finished=false`. Hlavnú robota bude robiť funkcia `step`, ktorá vždy prehľadá kúsok stromu a prípadne nastaví premennú `finished`. V programe by to mohlo vyzerat takto:

```

1 while (running) {
2     SDL_Event e;
3     while (SDL_PollEvent(&e)) {
4         switch (e.type) {
5             ... // spracovať ostatné typy udalostí
6
7             case SDL_MOUSEBUTTONDOWN:
8                 if (bv.onMouseDown(e.button).from.legal() && bv.board.winner() == Empty)
9                     plr.startThinking(bv.board); // krátke volanie, ktoré nastaví začiatok
10                break;
11            }
12        }
13        ...
14        if (plr.thinking) {
15            while (!plr.finished) { // tu voláme step(), kým máme čas v rámci zvoleného fps
16                plr.step();
17                uint32_t now = SDL_GetTicks();
18                if (now > lastFrame + msPerFrame) break;
19            }
20            if (plr.finished) { // ak skončil, prečítame výsledok a nastavíme novú pozíciu
21                Move m = plr.getResult();
22                bv+=m;
23            }
24        }
25        ...
26        // prípadne SDL_Delay, ak zostáva čas
}

```

## Ako robiť viac vecí naraz

---

Ako urobiť **step**? Pripomienim, že vo **findMove** robí hlavnú časť práce rekurzívna funkcia

```
1 MoveValue search(const Board& b, int hlbka, float alpha, float beta)
```

v nej sa, v najjednoduchšej verzii, robí zhruba takýto cyklus

```
1 auto ms = b.legalMoves();
2 for (int i = 0; i < ms.size(); i++) {
3     Board bb = b;
4     bb += ms[i];
5     float v = -search(bb, hlbka - 1, -beta, -alpha).val;
6     if (v > res.val) {
7         res.val = v;
8         res.m = ms[i];
9     }
10    if (res.val > alpha) alpha = res.val;
11    if (alpha >= beta) break;
12 }
```

Počas rekurzívneho volania sa postupne vytvára veľa svetov pre rôzne volania funkcie **search**. Ak chceme, aby sa dala prerušíť, nemôžeme použiť rekurziu, ale bude treba, aby sme si svety vytvárali sami. Urobíme si typ, v ktorom sa dá zapamätať jeden svet funkcie **search**: musí obsahovať všetky jej parametre aj lokálne premenné, napr.

```
1 struct State {
2     Board b;
3     int hlbka;
4     double alpha, beta;
5     MoveValue res;
6     std::vector<Move> ms;
7     int i;
8 };
```

**AlphaBetaPlayer** potom bude mať **vector<State>**, ktorý bude slúžiť ako zásobník a kde bude mať uložené všetky rozpracované svety. Volanie **search** sa pozrie na posledný svet: ak sú ešte ľahy, ktoré treba spracovať, tak vyrobí nový svet a pridá ho na koniec vektora, v opačnom prípade upraví hodnoty v predposlednom svete a posledný zmaže.

**Úloha 137.** Naprogramuj prehľadávanie tak, ako sme tu povedali, aby sa dalo v grafickom prostredí hrať.

Ked si vyriešil predchádzajúcu úlohu, asi vidíš nevýhody tohto prístupu. Prerobiť existujúceho hráča tak, aby vedel bežať "zároveň" so spracovaním udalostí vyžaduje veľa práce. V našom prípade bola len jedna rekurzívna funkcia, v ktorej sa robila všetka práca, ale keby si mal program, kde je viacero funkcií, z ktorých každá môže pracovať dlho a vzájomne sa volajú, celé by to bolo ešte oveľa zložitejšie.

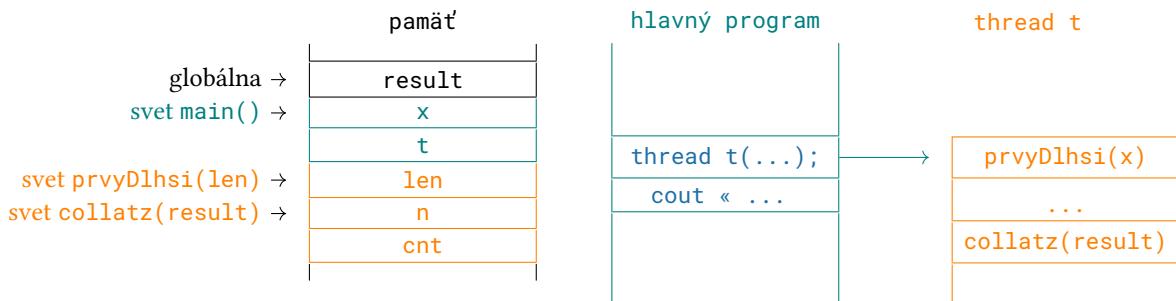
V C++ sa dá dosiahnuť, aby viacero funkcií bežalo "naraz", pomocou tzv. *vlákien (threads)*. V štandardnej knižnici **<thread>** je šablóna **thread**, ktorá umožňuje spúštať funkcie nezávisle od hlavného programu. Pozri si tento program:

```

1 #include <iostream>
2 #include <thread>
3 using namespace std;
4
5 int result;
6
7 int collatz(int n) {
8     int cnt;
9     for (cnt = 0; n > 1; cnt++)
10        if (n % 2 == 0) n = n / 2;
11        else n = 3 * n + 1;
12    return cnt;
13 }
14
15 void prvyDlhsy(int len) {
16     for (result = 1; true; result++)
17         if (collatz(result) > len) return;
18 }
19
20 int main() {
21     int x;
22     cin >> x;
23     thread t(prvyDlhsy, x);
24     cout << "Thread u pocita a program pokracuje" << endl;
25     t.join();
26     cout << "Thread vypocital" << result << endl;
27 }
```

Funkcia `prvyDlhsy` vyráta prvé číslo, ktorého Collatzova postupnosť (pozri Úlohu 9) je dlhšia ako zadané číslo. Napr. pre vstup 512 je výsledok 4484223.

V hlavnom programe sa na riadku 23 vyrobí premenná `t`, ktorá je typu `thread`. Jej konštruktor je šablóna, ktorá ako prvý parameter zoberie niečo spustiteľné (funkciu, funktor, lambdu<sup>1</sup>) a spustí to. Ďalšie parametre konštruktora sú parametre, ktoré sa použijú pri spustení<sup>2</sup>. V našom prípade sa spustí funkcia `prvyDlhsy(x)`. Akonáhle sa zavolá konštruktor, thread začne pracovať nezávisle a program pokračuje ďalej. Hlavný program aj novovytvorený thread majú spoločnú pamäť, takže to môže vyzerať takto:



V pamäti je globálna premenná `result`. Spustí sa hlavný program (funkcia `main`), ktorý vytvorí svoj svet s lokálou premennou `x`. Potom sa vyrobí premenná `t` a keď sa volá konštruktor `thread` na riadku 23, spustí sa zároveň aj funkcia `prvyDlhsy`, ktorá si v pamäti vyrobí svoj svet a potom zavolá funkciu `collatz`, ktorá si opäť urobí vlastný svet. Hlavný program beží ďalej a môže pristupovať k premenným `result`, `t` a `x`, vnútro funkcie `collatz` môže pristupovať k premenným `result`, `n` a `cnt`.

<sup>1</sup>pripomeň si kapitolu 25

<sup>2</sup>To, že konštruktor `thread` môže mať vždy iný počet parametrov, nie je nejaká záhadná špeciálna výnimka. Pomocou šablón sa dajú spraviť funkcie, ktoré nemajú pevné daný počet parametrov, ale pri každom volaní môžu mať iný počet. Kedže ale ten zápis je zo začiatku trochu neprehľadný a nikde sme to príliš nepotrebovali, tak som ti o tom nepovedal. Ak fa to zaujíma, treba si nájsť kľúčové slovo *parameter pack*, napr. [https://en.cppreference.com/w/cpp/language/parameter\\_pack](https://en.cppreference.com/w/cpp/language/parameter_pack)

## Ako robiť viac vecí naraz

So samotnou premenou typu `thread` sa toho moc robiť nedá, ale je dôležité, aby ostala v pamäti po celý čas, kým beží funkcia `thread`. Nasledujúci program zavolá funkciu `rob`, ktorá spustí thread s lambdou, čo dlho počíta. Funkcia `rob` vzápätí skončí, takže sa zavolá deštruktör premennej `t`. Lenže príslušná funkcia ešte beží, a preto program skončí s chybou.

```
1 void rob() {
2     thread t([](){
3         int j = 0;
4         for (int i = 0; i < 100000; i++) j += i;
5     });
6 }
7
8 int main() { rob(); }
```

Jedna z mála metód typu `thread` je `join`, ktorá slúži presne na to, aby sa takýmto situáciám vyhlo. Ak sa zavolá `t.join()` (ako v našom programe na riadku 25), aktuálny thread bude čakať, kým thread z premennej `t` dobehne. Potom je už bezpečné zavolať deštruktör.

Pri threadoch sice hovoríme, že bežia naraz, ale v skutočnosti ti to nikto nesľúbi. Väčšina procesorov má viac jadier<sup>3</sup> a každé môže v jednom momente bežať jeden thread. Kým je v celom systéme menej threadov ako jadier, systém sa spravidla snaží dať nový thread na samostatné jadro, takže thready naozaj bežia naraz. Ak je threadov viac, musí na jednom jadre bežať viacero threadov. Niektoré systémy to robia tak, že ich striedajú: nejaký krátky čas beží jeden, potom sa preruší, beží ďalší a tak dookola. Ale na niektorých systémoch (spravidla sú to malé procesory, ktoré sa používajú do rôznych zariadení) je také prepnutie veľmi drahé. Robia preto to, že zoberú jeden thread a ten beží, kým sa dá: to znamená až kým neskončí alebo nezačne čakať, či už na čítanie vstupu, nejakú udalosť, alebo kvôli volaniu ako `SDL_Delay`. Potom zoberú iný thread, ktorý chce bežať a spustia zase ten. Štandard jazyka C++ slúbuje len to, čo je pre všetky tieto systémy spoločné. Treba mať preto na pamäti, že pri programovaní threadov nemáš žiadnu záruku okrem toho, že ak nejaké jadro nemá čo robiť, tak sa naňom spustí aspoň jeden nejaký thread, ktorý chce práve bežať.

Ak si si spustil prvý príklad, tak pravdepodobne sa text z riadku 24 vypísal hneď a potom sa čakalo na dobehnutie funkcie `prvyDlhsy` v druhom threade pri volaní `join`. Ale pokojne to mohlo byť aj naopak: začal by pracovať thread `t` a kým má čo robiť (v našom prípade až kým neskončí), tak ho systém nechá bežať a až keď skončí, dá slovo hlavnému threadu. Existuje funkcia `this_thread::yield()`, ktorá signalizuje systému *Pozri sa, či nechce náhodou bežať nejaký iný thread, ktorý už dlho čaká. Ak áno, radšej ma preruš a začni vykonávať ten*. Čo presne sa ale stane po zavolani `yield()`, to tiež závisí od systému (štandard C++ to nijak neurčuje).

Užitočná je aj funkcia `this_thread::sleep_for`, ktorá robí podobnú vec ako `SDL_Delay`, totiž uspí aktuálny thread na daný čas. Kvôli čitateľnosti je ale navrhnutá tak, že parameter nie je `int`, ale `chrono::duration`, takže treba použiť `#include <chrono>` a potom použiť napr.

```
1 this_thread::sleep_for(chrono::milliseconds(1234));
```

Videl si, ako vyrobí thread, v ktorom beží nejaká funkcia. Určite si si to všimol, ale pripomienim, že návratová hodnota z tej funkcie sa zahodí a nemáš sa k nej ako dostať. Thready medzi sebou komunikujú tak, že jeden thread napíše do nejakej spoločnej<sup>4</sup> premennej a druhý odtiaľ prečíta. Možno ti v hlave zablikala kontrolka: ak nemám žiadnu kontrolu nad tým, kedy ktorý thread vykonáva ktorú časť svojho programu, nemôže spoločné čítanie a zapisovanie do zdieľanej premennej spôsobiť problémy? Môže, hneď ti to ukážem. Zober si nasledovný program

<sup>3</sup>Počet jadier sa dá zistiť pomocou `cout << thread::hardware_concurrency() << endl;`.

<sup>4</sup>bud je to globálna premenná ako v našom príklade, alebo napr. pošleš ako parameter funkcie threadu pointer na premennú a pod.

```

1 #include <iostream>
2 #include <set>
3 using namespace std;
4 set<int> s;
5
6 void dump() {
7     for (int x : s) cout << x << " ";
8     cout << endl;
9 }
10
11 void rob() {
12     for (int i = 5; i < 100; i++) {
13         s.erase((i - 5) % 100);
14         s.insert(i % 100);
15         if (i % 10 == 0) dump();
16     }
17 }
18
19 int main() {
20     for (int i = 0; i < 5; i++)
21         s.insert(i);
22     rob();
23 }
```

Máme v ňom globálnu množinu (t.j. vyvážený vyhľadávací strom, pozri kapitolu 27) `s`, do ktorej na začiatku dáme čísla 0, ..., 4. Funkcia `rob` vždy najmenšie číslo z množiny vyberie a vloží tam o jedno väčšie. Berieme iba zvyšky po delení 100, aby bol prehľadnejší výpis. Funkcia `dump` vypíše obsah množiny. Výstup programu vyzerá takto:

```

6 7 8 9 10
16 17 18 19 20
26 27 28 29 30
36 37 38 39 40
46 47 48 49 50
56 57 58 59 60
66 67 68 69 70
76 77 78 79 80
86 87 88 89 90
0 96 97 98 99
```

Teraz program zmeňme tak, že `rob` pobeží v samostatnom thready a bude množinu meniť stále dookola. Hlavný program, po tom, čo spustí thread, vždy chvíľu počká a vypíše obsah množiny. Keď program spustí viackrát, výstupy sa budú lísiť, ale môžu vyzeráť napr. takto:

```

1 #include <chrono>
2 #include <iostream>
3 #include <set>
4 #include <thread>
5 using namespace std;
6 set<int> s;
7
8 void dump() {
9     for (int x : s) cout << x << " ";
10    cout << endl;
11 }
12
13 void rob() {
14     for (int i = 5; i < 1000000; i++) {
15         s.erase((i - 5) % 100);
16         s.insert(i % 100);
17     }
18 }
19
20 int main() {
21     for (int i = 0; i < 5; i++) s.insert(i);
22     thread t(rob);
23     for (int i = 0; i < 10; i++) {
24         this_thread::sleep_for(chrono::milliseconds(1));
25         dump();
26     }
27     t.join();
28 }
```

```

73 24 25 26 25 26
17 23 24
1 7 8
8 14 15
59 65 66 67
8 14 15
44 50 51 52
61 67 68 69
3 9 10 11
26 32 33 34
```

## Ako robiť viac vecí naraz

---

Čo sa stalo? Jeden thread práve robil `insert` alebo `erase` na vyhľadávacom strome, kým druhý thread ho začal prechádzať, aby ho vypisoval. Ale počas toho, ako vypisoval strom, mu prvý thread ten strom pod rukami menil, takže sa vypísalo niečo, čo v žiadnom momente v strome nebolo. Dostali sme sa k problému, ktorému sa hovorí *vzájomné vylúčenie* (*mutual exclusion*): máme program, v ktorom beží viacero threadov, ale je v ňom časť (v našom prípade manipulácia s množinou `s`), v ktorej nikdy nesmie pracovať viac threadov naraz. Základným prostriedkom na zabezpečenie vzájomného vylúčenia je typ `mutex`, ktorý je definovaný v `<mutex>`. Premenné typu `mutex` sa nedajú priradovať ani kopírovať, majú dve základné metódy: `lock()` a `unlock()`. Ako fungujú je opäť najlepšie vidno na príklade. Zoberme program s troma threadmi, z ktorých každý chce postupne trikrát vypísať riadok.

```
1 #include <iostream>
2 #include <string>
3 #include <thread>
4 #include <vector>
5 using namespace std;
6
7 void zdrz() { // dajme tomu, že tu niečo zložité počíta
8     for (int i = 0, j = 0; i < 100000; i++) j += i;
9 }
10
11 void pis(const string &s) {
12     for (int i = 0; i < 3; i++) {
13         for (char c : s) {
14             cout << c;
15             zdrz();
16         }
17         cout << endl;
18         zdrz();
19     }
20 }
21
22 int main() {
23     vector<thread> ts;
24     ts.push_back(thread(pis, "kedsomisielcezhoru"));
25     ts.push_back(thread(pis, "STRETOLSOMTAMPOTVORU"));
26     ts.push_back(thread(pis, "!$^!#!~^*?$!"));
27
28     for (auto &t : ts) t.join();
29 }
```

Ked' ho spustíš, môže to vyzeráť napr.

```
Sk!Te$Rd^Es!T0o?Lm#Si!Os~Mi^Te*Al?Mc$Pe!Oz
Th!Vo$Or^Ru!U?

#kS!Te~Rd^E*sT?o0$Lm!Si
0!sM$iT^Ae!Ml?P#c0!eT~zV^h0*oR?rU$u
!
S
kTeRdEsTo0mLiSs0iMeTlAcMePz0hToVr0uR
U
```

Thready sa nepredvídateľne striedajú v tom, kto práve vypíše znak. Zmeňme teraz program tak, že pridáme `#include <mutex>`, vyrobíme globálnu premennú `mutex m` a funkciu `pis` zmeníme takto:

```

1 void pis(const string &s) {
2     for (int i = 0; i < 3; i++) {
3         m.lock();
4         for (char c : s) {
5             cout << c;
6             zdrz();
7         }
8         cout << endl;
9         m.unlock();
10        zdrz();
11    }
12 }
```

Hlavný program sa nezmení, spustí tri ready. Prvý thread (nevieme, ktorý, ale jeden z nich to bude, nazvime ho *A*), sa dostane na riadok 3 a zavolá `m.lock()`. Tým sa mutex `m` "zamkne". Ďalší thread, dajme tomu, že *B*, ktorý dorazi na riadok 3, tiež zavolá `m.lock()`. Mutex `m` je už ale zamknutý, preto systém thread *B* preruší a zapamätá si, že *B* čaká na mutex `m`. Podobne aj tretí thread. Po čase thread *A* dokončí vypisovanie reťazca a príde na riadok 9, kde zavolá `m.unlock()`. Mutex sa tým "odomkne", systém zistí, že nejaké ready na mutex `m` čakali, vyberie z nich jeden, napr. *B*, mutex zamkne a thread *B* pustí bežať. Z pohľadu threadu *B* sa nič nestalo: zavolal `m.lock()` a pokračoval ďalej (medzitým chvíľu spal, ale to si nijak nevšimol).

Na to, aby to celé fungovalo, musí byť `unlock()` podporovaný operačným systémom. Treba totiž, aby sa celá postupnosť *Odomkni mutex, zisti, či na ňom nejaké ready čakali, ak áno vyber jeden z nich, pusti ho bežať a zamkní mutex* vykonalá *atomicky*, t.j. ako keby to bola jedna inštrukcia. Inak by nejaký iný bežiaci thread mohol nájsť odomknutý mutex v tom krátkom čase medzi tým, keď už systém vybral nový thread na bežanie, ale ešte nezamkol mutex.

Ked' takto upravený program spustíš, už bude pracovať správne, vypíše sa napr.

```
!$^!#!~^*?$!
STRETOLOSMAMPOTVORU
kedsomisielcezhoru
kedsomisielcezhoru
STRETOLOSMAMPOTVORU
!$^!#!~^*?$!
kedsomisielcezhoru
STRETOLOSMAMPOTVORU
!$^!#!~^*?$!
```

Akonáhle sa nejaký thread dostal do tej časti programu, v ktorej sa vypisuje, dokončil vypisovanie celého riadku bez prerušenia. Ostatné ready, ktoré by medzitým chceli vypisovať, poslušne čakali, kým na ne príde rad.

Použitie mutexu je veľmi náchylné na robenie chýb, ktoré sa veľmi zle hľadajú. Keby napr. funkcia `pis` zavolala `return` predtým ako zavolá `m.unlock()`, mutex by ostal zamknutý a ostatné ready by sa nikdy nedostali k slovu. Takisto mutex sa musí používať v správnom poradí, t.j. nejaký thread zavolá `lock()` a potom ten istý thread zavolá `unlock()`. Volať `unlock()` z iného threadu, volať dvakrát za sebou `lock()` z toho istého threadu a pod. môže spôsobiť záhadné chyby, ktoré sa neprejavia pri každom behu programu, ale iba občas. Preto sa `mutex` používa zriedkakedy priamo, ale častejšie prostredníctvom `lock_guard`. Typ `lock_guard` v konštruktore dostane ako parameter premennú typu `mutex` a ešte v konštruktore na ňu zavolá `lock()`. V deštruktore sa potom zavolá `unlock()`. To znamená, že ak vyrobíš premennú typu `lock_guard`, program za ňou je chránený príslušným mutexom, až kým sa nezavolá jej deštruktor. Napr. funkcia `pis` by sa dala napísat takto:

## Ako robiť viac vecí naraz

```
1 void pis(const string &s) {
2     for (int i = 0; i < 3; i++) {
3         {
4             lock_guard guard(m);
5             for (char c : s) {
6                 cout << c;
7                 zdrz();
8             }
9             cout << endl;
10        }
11    }
12 }
```

Zložený príkaz na riadkoch 3 a 10 vytvára svet, v ktorom žije premenná `guard`. Pri jej vytvorení sa na riadku 4 zamkne mutex `m` a v jej deštruktore na riadku 10 sa odomkne. Príjemné na tom je, že ak niekedy v budúcnosti zmeníš program tak, že napr. niekedy počas vypisovania zavoláš `return`, deštruktor `guard` sa zavolá aj tak a nezanesieš si do programu ozaj nepríjemnú chybu.

Niekedy je ombedzenie, že ten istý thread, čo zavolał `lock()` musí zavolať aj `unlock()` dosť nepríjemné. V novších<sup>5</sup> štandardoch C++ je aj všeobecnejší mechanizmus, tzv. *semafór*. Používa sa podobne ako `mutex`: pridá sa `#include <semaphore>` a vyrobí sa premenná typu `counting_semaphore` niekde, kde ju všetky thready vidia (napr. ako globálna premenná). Semafór<sup>6</sup> je vlastne počítadlo, ktoré sa dá atomicky zvyšovať a znižovať. V konštruktore dostane štartovaci hodnotu. Ak máš premennú napr. `counting_semaphore s(4)`, tak ktorýkoľvek thread môže hocikedy zavolať `s.acquire()`, čím sa hodnota počítadla zníži, alebo `s.release()`, čím sa hodnota počítadla zvýši. Hodnota nikdy nebude záporná a ak nejaký thread zavolá `s.acquire()` vtedy, keď hodnota počítadla bola 0, systém ho preruší a zapamäta si, kde čaká. Keď potom nejaký iný thread zavolá `s.release()`, tak podobne ako pri mutexoch sa atomicky vyberie jeden z čakajúcich threadov a spustí sa.

Preuze semafór môže jeden thread zvýšiť a druhý znížiť, používajú sa na dohadovanie medzi threadmi: jeden thread na semafóre zaspí (spraví `acquire()` na nulovom počítadle) a čaká, kým ho druhý zobudí (zavolá `release()`).

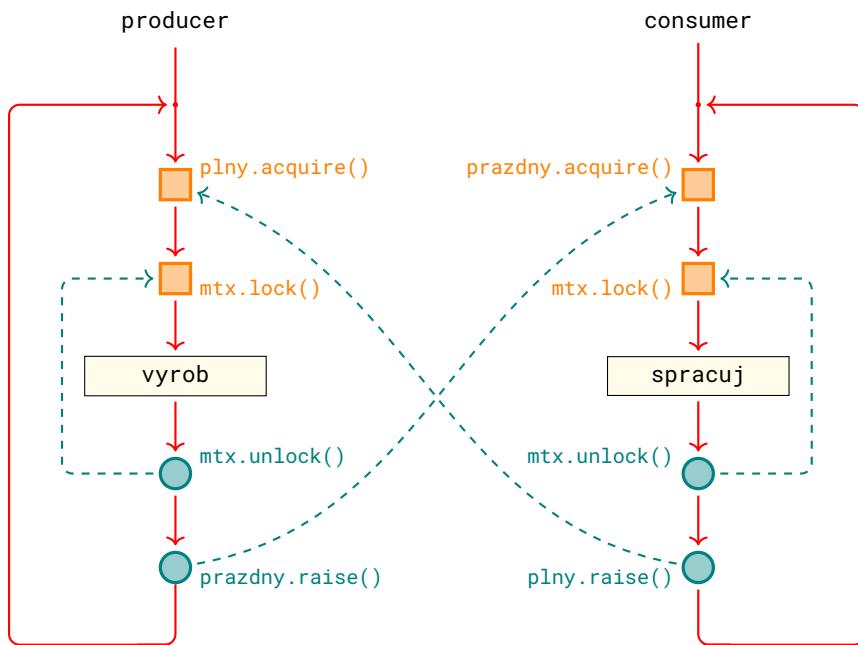
Typický príklad na komunikáciu threadov je tzv. *producer-consumer* problém. Dajme tomu, že chceš vyrobiť dataset, v ktorom budú tváre ľudí z obrázkov na internete. Máš jednu funkciu, ktorá vie stiahnuť náhodný obrázok z internetu a druhú funkciu, ktorá vie zobrať obrázok a vyrezáť z neho oblasť s tvárou. Problém je, že obe tie funkcie môžu trvať rôzne dlho. Raz treba dlho čakať, kým sa podarí nejaký obrázok stiahnuť a inokedy sa obrázky chrlia rýchlo, ale ich spracovanie trvá dlho. Vyriešime to tak, že si urobíme buffer, do ktorého bude prvá funkcia (nazvime ju `producer`) ukladať obrázky a druhá funkcia (nazvime ju `consumer`) z neho bude čítať. Na buffer môžeme použiť napr. triedu `deque` z STL; `deque` sa podobá `vector`u, ale vieme efektívne pridať a uberať z oboch koncov, t.j. máme operácie `push_back`, `push_front`, `pop_back`, `pop_front`, `front` a `back`, ktoré vedia pridať, uberať a pristupovať k prvkom na začiatku a konci poľa<sup>7</sup>. Chceme, aby si `producer` pracoval a keď získa obrázok, uloží ho na koniec buffra. Podobne `consumer` si postupne číta z buffra obrázky a spracováva ich. Ak je buffer prázdny, `consumer` by mal zaspať a zobudíť sa, až keď bude mať robotu. Podobne nechceme, aby buffer príliš narástol, takže ak je v ňom viac ako  $n$  obrázkov, `producer` by mal zaspať a zobudiť sa, až keď bude v buffri miesto. No a ešte by sme chceli aj takú možnosť, aby bežalo naraz viac threadov `producer` a viac threadov `consumer`.

Aby sme zabezpečili, že buffer sa nikdy nepokazí, budeme si prístup k nemu chrániť mutexom `mtx`. Zároveň si jednotlivé `producer` a `consumer` thready budú signalizovať, kedy majú zaspať a kedy sa majú zobudiť. Na to použijeme dva semafóry. Semafór `prazdny` bude mať vždy takú hodnotu počítadla, kolko je v buffri vecí. Začína teda s nulou, `consumer` ho zníži a `producer` ho zvýši. Druhý semafór, `plny`, bude mať hodnotu počítadla vždy takú kolko je v buffri voľných miest. Celý systém by vyzeral takto:

<sup>5</sup>od štandardu C++20 vyššie, takže komplilátoru chceš povedať napr. `-std=c++20`

<sup>6</sup>Pôvodne ho vymyslel E. W. Dijkstra a predstavoval si pri tom také železničné návestie s ukazovateľom, ktorý sa môže posúvať hore a dolu.

<sup>7</sup>Premysli si, ako by si takú triedu mohol vyrobiť sám.



Na tento obrázok sa treba dobre zadívať a presvedčiť sa, že je to naozaj v poriadku, nech by thready pracovali v akomkoľvek poradí: keď si **producer** zamkne **mtx**, v buffri je voľné miesto, keď si **consumer** zamkne **mtx**, je v buffri aspoň jedna vec. Zároveň nikdy nenastane *deadlock*, t.j. situácia, keď všetky thready spia a čakajú a celý program sa tým pádom zasekne.

Ked' už máme program takto navrhnutý, napísaf ho je ľahké. Tu urobíme iba simuláciu, namiesto vyrábania aj spracovávania dáme iba čakanie. Najprv si pripravíme knižnice a globálne premenné. Tu si len treba všimnúť, že **counting\_semaphore** je šablóna, ktorá ako parameter dostane očakávanú maximálnu hodnotu počítadla<sup>8</sup>. Je to viac-menej iba kvôli možnej optimalizácii, v programe môže počítadlo narásť aj viac.

```

1 #include <chrono>
2 #include <deque>
3 #include <iostream>
4 #include <mutex>
5 #include <random>
6 #include <semaphore>
7 #include <thread>
8 #include <vector>
9 using namespace std;
10 const int N = 10;
11
12 deque<int> buffer;
13
14 counting_semaphore<N> plny(N), prazdny(0);
15 mutex mtx;

```

Pre jednoduchosť povieme, že **producer** aj **consumer** budú bežať stále. V naozajstnom programe by si tam chcel mať nejaký spôsob, ako ich zastaviť (napr. zdieľanú premennú, do ktorej sa zapíše, ak treba skončiť). V našom prípade si **producer** vymyslí číslo, chvíľu počká a uloží ho na koniec buffra.

<sup>8</sup>Ako sme už spomínali, šablóny môžu mať okrem typov aj celočíselné parametre, napr. pre šablónu **template <typename T, int N> int f(T x) {return (int)(x) + N;}** je **f<float, 4>** funkcia, ktorá zoberie parameter typu **float**, pretpojuje ho na **int** a priráta k nemu 4. Hodnota 4 je v tele funkcie fixovaná, pre rôzne hodnoty sa zo šablóny vyrobia rôzne funkcie.

## Ako robiť viac vecí naraz

```
1 void producer(int id) {
2     mt19937 rnd{random_device{}()};
3
4     while (true) {
5         int num = rnd() % 1000;
6         this_thread::sleep_for(chrono::milliseconds(rnd() % 1000));
7         plny.acquire();
8         mtx.lock();
9         cout << "Producer" << id << " vyrobil" << num << endl;
10        buffer.push_back(num);
11        mtx.unlock();
12        prazdny.release();
13    }
14 }
```

Podobne napíšeme *consumer*:

```
1 void consumer(int id) {
2     mt19937 rnd{random_device{}()};
3
4     while (true) {
5         prazdny.acquire();
6         mtx.lock();
7         int num = buffer.front();
8         buffer.pop_front();
9         cout << "Consumer" << id << " spracoval" << num << endl;
10        mtx.unlock();
11        plny.release();
12        this_thread::sleep_for(chrono::milliseconds(rnd() % 1000));
13    }
14 }
```

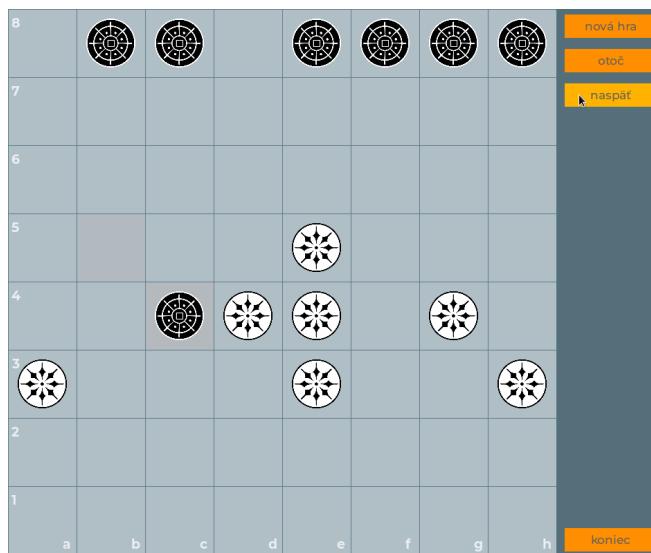
Hlavný program len spustí thready:

```
1 int main() {
2     vector<thread> ts;
3     for (int i = 1; i <= 4; i++) ts.push_back(thread(producer, i));
4     for (int i = 1; i <= 2; i++) ts.push_back(thread(consumer, i));
5     for (auto &t : ts) t.join();
6 }
```

Toto by nateraz mohlo stačiť na základný prehľad o multithreadovom programovaní. Keď sa vrátíme k nášmu Breakthrough projektu: namiesto zložitého prerábania hráča, ako na začiatku tejto kapitoly, je jednoduchšie zaobaliť existujúceho hráča do threadu. V konštruktore sa spustí thread, ktorý spí na semafóre. Keď treba rátať ťah, semafór sa zdvihne a thread počíta. Hlavný program pri spracovaní udalostí kontroluje, či hráčov thread už dorátal ťah.

**Úloha 138.** Naprogramuj multithreadového hráča.

Náš Breakthrough projekt celkom dobre postupuje a chýba už iba GUI<sup>1</sup>. Na to existuje nepreberné množstvo rôznych knižníc, od úplne minimalistických ako napr. [microUI<sup>2</sup>](#) až po obrovské kolosy ako [Qt<sup>3</sup>](#), ale opäť kvôli vysvetleniu nejakých ďalších vecí by som chcel, aby sme si jednoduchú GUI knižnicu naprogramovali. Spravíme iba veľmi jednoduchý základ, s ktorým budeme vedieť vyrobiť takýto program:



V našej knižnici bude celé okno rozdelené na *widgety*. Widget bude trieda, ktorá spravuje časť okna, vie sa prispôsobiť zmenenej veľkosti, vie reagovať na kliknutia myšou (prípadne iné udalosti, napr. stlačenie klávesy) a vie sa vykresliť do SDL renderera. Príkladom widgetu je naša trieda [BoardView](#) z kapitoly 36. Iný widget by mohol byť napr. [Button](#), ktorý predstavuje jeden gombík a stará sa o to, aby sa naňho dalo kliknúť. Ďalší typ widgetov sa stará o rozmiestňovanie iných widgetov. Widget [Layout](#) si pamäta zoznam widgetov a prepočítava ich veľkosť a umiestnenie. V našom prípade chceme mať ako hlavný widget horizontálny layout, ktorý obsahuje dva widgety: [BoardView](#) a vertikálny layout. Ten obsahuje päť widgetov: tri gombíky, jeden widget na vyplnenie miesta a ďalší gombík.

Takže stačí napišeť pre každý widget príslušnú triedu, poskladať to celé dokopy a hotovo. Vidno tu ale dva problémy. Po prvej, je veľa funkcionality, ktorá je pre všetky (prípadne niektoré) widgety spoločná. Napr. každý widget si má pamätať [SDL\\_Rect rect](#), ktorý na obrazovke zaberá. Takisto môže mať okraje nejakej hrúbky. Takže všetky widgety budú mať metódu [resize\(SDL\\_Rect newRect\)](#), v ktorej bude

```

1 rect = newRect;
2 rect.x += padding;
3 rect.y += padding;
4 rect.w -= 2 * padding;
5 rect.h -= 2 * padding;
```

Naďalej widgety [Layout](#) ešte prerátajú rozmery svojich vnútorných widgetov a zavolajú ich [resize](#). Postupne sa bude nabalovať viac a viac kódu, ktorý je na veľa miestach rovnaký, čo je oštara. Na druhý problém naraziš, ak začneš rozmyšľať, ako naprogramovať typ [Layout](#). Potrebuje mať totiž zoznam svojich vnútorných widgetov, ktoré môžu byť rôznych typov. Ukážem ti, že na oba tieto problémy je dobré riešenie mechanizmus, ktorému sa v C++ hovorí *dedičnosť*.

<sup>1</sup>Graphical User Interface: gombíky, scrollovátká a iné grafické ovládátká

<sup>2</sup><https://github.com/rxi/microui>

<sup>3</sup><https://doc.qt.io/qt-6/qtgui-index.html>

## Dedičnosť

Opäť podme od začiatku. Dajme tomu, že by si mal triedu

```
1 struct Widget {  
2     int x, y;  
3     void render();  
4 };
```

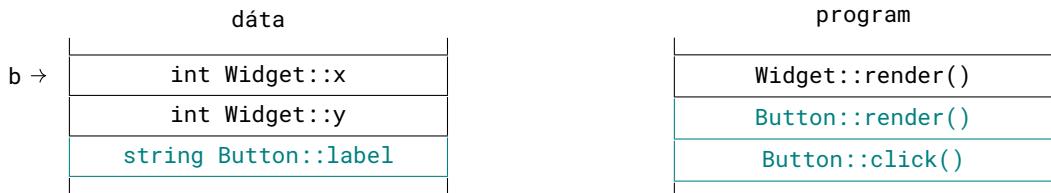
Ked' urobíš premennú `Widget w`, v pamäti to bude vyzeráť takto:



To znamená, že každá premenná typu `Widget` má v pamäti dve premenné `int` a navyše je v programe funkcia `Widget::render()`. Teraz môžeš urobiť triedu `Button`, ktorá bude mať všetko to, čo `Widget` a aj niečo navyše. Hovoríme, že `Button` dedí od `Widget` a zapíše sa to takto:

```
1 struct Button : Widget {  
2     string label;  
3     void render();  
4     void click();  
5 };
```

Každá premenná typu `Button` bude mať tri premenné: dve zdelené od `Widget` a jednu novú. V programe budú tri funkcie, takže `Button b`; bude vyzeráť takto:



Môžeš si to vyskúšať, program

```
1 #include <iostream>  
2 #include <string>  
3 using namespace std;  
4  
5 struct Widget {  
6     int x, y;  
7     void render() { cout << "Widget::render" << x << " " << y << endl; }  
8 };  
9  
10 struct Button : Widget {  
11     string label;  
12     void render() { cout << "Button::render" << x << " " << y << " " << label << endl; }  
13     void click() { cout << "Button::click" << x << " " << y << " " << label << endl; }  
14 };  
15  
16 int main() {  
17     Widget w;  
18     w.x = 12; w.y = 42; w.render();  
19  
20     Button b;  
21     b.x = 17; b.y = 47; b.label = "kikiriki"; b.render(); b.click();  
22 }
```

vypíše

```
Widget::render 12 42
Button::render 17 47 kikiriki
Button::click 17 47 kikiriki
```

Teraz z programu vymaž riadok 12, teda v programe už nebude funkcia `Button::render()`. Mala by sa vypísať chyba, že `b.render()` volá neexistujúcu funkciu `Button::render()`, ale chyba sa nevypíše. Keď upravený program spustíš, vypíše sa

```
Widget::render 12 42
Widget::render 17 47
Button::click 17 47 kikiriki
```

Čo sa stalo? Pri dedení sa totiž dedia nielen premenné, ale aj funkcie. Je to podobne ako s lokálnymi premennými: ak existuje funkcia z daného typu (napr. `Button::render()`), použije sa tá. Ak neexistuje, komplilátor sa skúsi pozrieť na typ, z ktorého sa dedilo, a použije funkciu odtiaľ (napr. `Widget::render()`). Keď sa nad tým zamyslíš, dáva to zmysel: pretože `Button` obsahuje všetky premenné, ktoré má `Widget`, tak každá funkcia, ktorá by pracovala na premennej typu `Widget` môže rovnako dobre pracovať na premennej typu `Button`. V programe sa preto zavolala funkcia `Widget::render()` na premennú `b`.

Teraz by sa zdalo, že to ale nesedí: v kapitole 13 sme hovorili, že metódy sú funkcie, ktoré majú “neviditeľný” prvý parameter `this`, takže v našom prípade máme funkcie

```
1 void Widget::render (Widget* this) {...}
2 void Button::render (Button* this) {...}
```

Keď sa teda zavolá `Widget::render` namiesto `Button::render`, tak sa musí aj zmeniť typ pointra z `Button*` na `Widget*`. To sa aj v skutočnosti udeje a nielen pri volaní metód. Opäť, keď sa nad tým zamyslíš, tak `Widget*` je *adresa v pamäti, kde sa začínajú dátá nejakej premennej typu Widget*. To isté ale platí aj pre `Button*:` je to adresa v pamäti, kde sa začínajú dátá pre nejakú premenňu typu `Widget`. Že za nimi sa potom nachádzajú ďalšie dátá tak, že je to celá premenňa typu `Button`, to je už druhá vec. Preto je vždy možné pointer na “väčší” typ (napr. `Button`) použiť tam, kde sa vyžaduje pointer na “menší” typ (napr. `Layout`).

Zase si to vyskúšaj. Vráť naspäť vymazaný riadok tak, aby `Widget` aj `Button` mali vlastnú funkciu `render` a hlavný program zmeň takto:

```
1 int main()
2     Button *b = new Button;
3     b->x = 17; b->y = 47; b->label = "kikiriki";
4     b->render();
5
6     Widget *q = b;
7     q->render();
8 }
```

Vypíše sa

```
Button::render 17 47 kikiriki
Widget::render 17 47
```

Prečo? Premenná `b` je typu `Button*`, preto volanie `b->render()` na riadku 4 zavolá funkciu `Button::render`. Premenná `q` je typu `Widget*`. Na riadku 6 sa do nej priradí `b`, takže `q` ukazuje na to isté miesto v pamäti, kde je uložená premenňa `*b`. Volanie `q->render()` na riadku 7 preto zavolá `Widget::render` na premennej typu `Widget`, ktorá vznikne “orezaním” `*b` do typu `Widget` (t.j. zabudne sa na `b->label`).

## Dedičnosť

S týmto mechanizmom máme jednu dobrú a jednu zlú správu. Dobrá správa je, že v našom type `Layout` by sme vedeli spraviť pole rôznych widgetov. Ak všetky widgety budú nejakým spôsobom<sup>4</sup> dediť zo základného typu `Widget` tak môžeme spraviť `vector<Widget *> children`, ktorý bude obsahovať pointre na widgety, ktoré môžu byť gombíky, ďalšie layouty a pod.

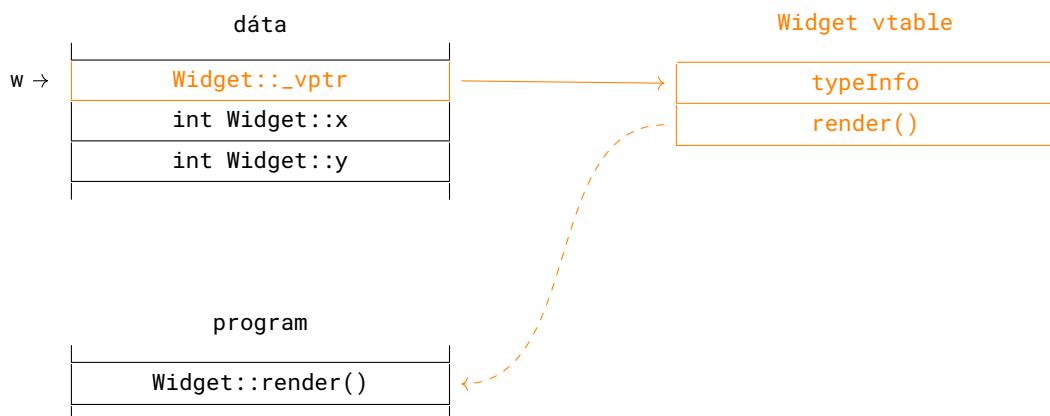
Zlá správa je, že ak potom zavoláme napr.

```
1 for (int i = 0; i < children.size(); i++)
2     children[i]->render();
```

Tak sa na každé z detí zavolá `Widget::render`. Všetky sú totiž typu `Widget*` a komplilátor nemá ako vedieť, že kedysi dávno sme tam priradili iné typy. Chcelo by to nejaký mechanizmus, ktorým by sa zapamätaло, aký typ má ktoré z detí, aby potom sa volala metóda `render` z príslušného typu. Tento mechanizmus sa volá *virtuálne metódy*. Zober si takto zmenenú triedu (pred deklaráciu `render` som pridal slovo `virtual`)

```
1 struct Widget {
2     int x, y;
3     virtual void render();
4 };
5
6 void Widget::render() { cout << "Widget::render" << x << " " << y << endl; }
```

Ked teraz vyrobíš premennú `Widget w`, v pamäti to bude vyzerať takto:

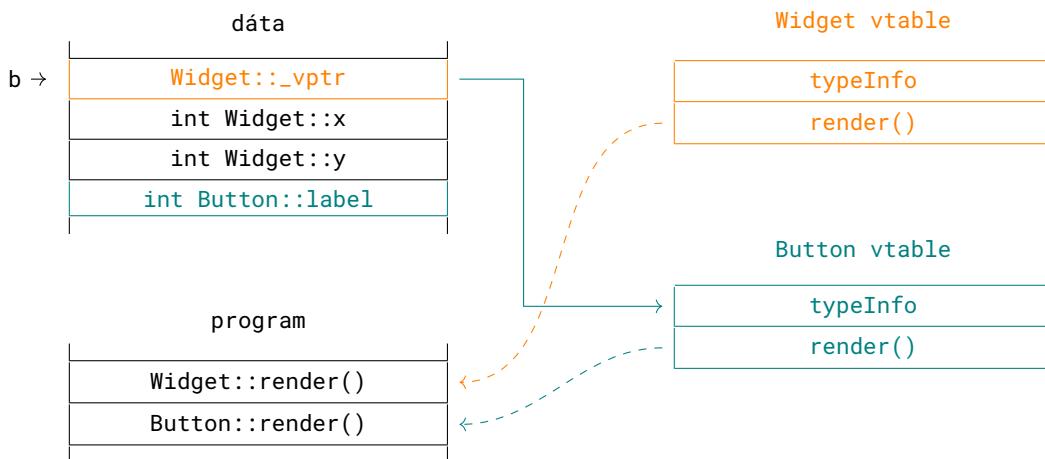


Kompilátor pre každý typ, ktorý obsahuje virtuálne metódy, vyrobí v pamäti zoznam `vtable`, kde sú adresy, na ktorých sú v programe uložené príslušné funkcie. Každá premenná má naviac jednu položku `_vptr`, čo je pointer do `vtable` svojho typu. Virtuálna funkcia, napr. `w.render()` sa teraz nezavolá priamo, ale prečíta sa adresa z `vtable` a zavolá sa príslušná funkcia z nej. Akonáhle je nejaká metóda virtuálna, tak sú virtuálne aj všetky jej verzie v zdedených typoch. Preto ked teraz urobíš

```
1 struct Button : Widget {
2     string label;
3     void render();
4 };
5
6 void Button::render() {
7     cout << "Button::render" << x << " " << y << " " << label << endl;
8 }
```

tak premenná `Button b` bude v pamäti vyzerať

<sup>4</sup>aj nepriamo, napr. `Layout` bude dediť z `Widget` a `HLayout` bude dediť z `Layout`



Typ `Button` zdedil všetky premenné z `Layout` a začiatok pamäte vyzerá rovnako ako v `Layout`, iba tentokrát `_vptr` ukazuje na `vtable`, ktorá patrí `Button`. Teraz je jasné, čo sa stane, ak priradíš napr. `Widget *w = &b;` Pointer `w` bude ukazovať na to isté miesto, kde je uložená `b`, ale keďže je typu `Widget*`, bude "vidieť" iba `_vptr`, `x` a `y`. Pointer `_vptr` bude ale ukazovať na `vtable` pre typ `Button`, preto `w->render()` zavolá `Button::render()`.

Takže si to zhrňme. Ak máš rodičovskú triedu (napr. `Layout`), môžeš z nej zdediť triedu pomocou dvojbodky

```
1 struct Button : Widget { ... };
```

Zdedená trieda bude obsahovať všetky premenné a metódy (funkcie) z pôvodnej. Môže si pridať nové premenné a nové metódy s novým menom, prípadne predefinovať existujúce (napr. náš `Button::render()`). Pointer na zdedenú triedu sa dá použiť všade tam, kde sa vyžaduje pointer na pôvodnú triedu. Ak je metóda označená ako `virtual`, každá premenná si pamätá, s akým typom bola vytvorená a vždy zavolá metódu z toho typu.

Pripomínam, že vždy môžeš používať celé meno metódy s dvoma dvojbodkami, aby si rozlíšil, ktorý typ chceš použiť. Veľakrát chceš urobiť niečo takéto:

```
1 void Button::render() {
2     Widget::render();
3     ...
4 }
```

To znamená, že `Button::render()` (ktorý má "neviditeľný" prvý parameter `Button *this`) najprv zavolá `Widget::render(this)`. To môže urobiť, lebo `Widget::render` chce parameter `this` typu `Widget *` a `Button` je zdedený z `Widget`. Takže ak zavolás `b.render()` na premennú typu `Button`, najprv sa na premennej `b` zavolá `Widget::render()` (ktorá môže robiť spoločné veci, napr. prekresliť pozadie a pod.) a potom sa urobia veci špecifické pre `Button`.

Tento mechanizmus volania "rodičovskej" metódy je trochu iný pri konštruktoroch. Konštruktor sa totiž zavolá vtedy, keď sa premenná vyrába, a potom ho už nemôžeš volať "ručne". Pri zdedených triedach sa automaticky volajú konštruktory v poradí dedenia, takže v našom prípade by sa najprv zavolal konštruktor pre `Widget` a potom konštruktor pre `Button` (pri deštruktorech je to v opačnom poradí). Preto tento program<sup>5</sup>:

```
1 #include <iostream>
2 using namespace std;
3
4 struct Widget {
5     Widget() { cout << "Widget\u00d7constructor" << endl; }
```

<sup>5</sup>Deštruktur som napísal ako `virtual`. Teraz na tom nezáleží, ale ak máš v triede virtuálne metódy, je spravidla dobrý nápad, aby bol aj deštruktur virtuálny. Skús si rozmyslieť, prečo.

## Dedičnosť

```
6     virtual ~Widget() { cout << "Widget_destructor" << endl; }
7 };
8
9 struct Button : Widget {
10     Button() { cout << "Button_constructor" << endl; }
11     virtual ~Button() { cout << "Button_destructor" << endl; }
12 };
13
14 int main() {
15     Button b;
16     cout << "kváák" << endl;
17 }
```

vypíše

```
Widget constructor
Button constructor
kváák
Button destructor
Widget destructor
```

Ak chceš do konštruktora poslať parametre, dá sa to urobiť pomocou volania koštruktora rodičovskej triedy za dvojbodkou rovnako, ako sme volali konštruktory premenných v kapitole 22. Vyskúšaj si takto upravený program:

```
1 #include <iostream>
2 using namespace std;
3
4 struct Widget {
5     int x;
6     Widget(int _x) : x(_x) { cout << "Widget_constructor" << endl; }
7     virtual ~Widget() { cout << "Widget_destructor" << endl; }
8 };
9
10 struct Button : Widget {
11     int y;
12     Button(int _x, int _y) : Widget(_x), y(_y) {
13         cout << "Button_constructor" << endl;
14     }
15     virtual ~Button() { cout << "Button_destructor" << endl; }
16 };
17
18 int main() {
19     Button b(42, 47);
20     cout << "kváák" << b.x << " " << b.y << endl;
21 }
```

Konštruktor **Button** dostane dva parametre. Najprv zavolá konštruktor pre **Widget** s parametrom **\_x**. Ten na staví<sup>6</sup> hodnotu premennej **x**. V konštruktore pre **Button** sa potom premenňa **y** nastaví na hodnotu **\_y** a vypíše sa oznam.

Toto je zhruba všetko, čo budeme o dedičnosti potrebovať vedieť do nášho GUI. Nestarali sme sa tu o riadenie prístupu, len poviem, že privátne premenné a metódy nie sú po dedení viditeľné. Ak by bola v predchádzajúcom príklade premenňa **x** vo **Widget** označená ako **private**, tak pri volaní **b.x** vyhlásí komplilátor chybu. Ničmenej,

<sup>6</sup>využil som, že **int** má v C++ aj konštruktor s jedným parametrom

premenná `b` by obsahovala aj `x`, ale jediný spôsob, ako sa k nemu `b` môže dostať, je pomocou metód zdelených z `Widget`.

Na riadenie prístupu existujú rôzne módy dedenia, takže niekedy môžeš vidieť napr. `struct Button : private Widget {...};`, prípadne namiesto `private` môže byť `public` alebo `protected`. My to tu nebudeme používať, ale ak sa s tým stretneš, kompletné detaily sú napr. tu<sup>7</sup>.

No a nakoniec len pridám, že je možné dediť aj z viacerých tried. Keby si mal napr. triedu `Textual` pre veci obsahujúce text, mohol by si triedu `Button` deklarovať

```
1 struct Button : Widget, Textual { ... };
```

čím by zdedil všetky premenné aj metódy z oboch tried. Niekedy sa to hodí, ale môže to začať byť trochu komplikované, ak si nedáš pozor. Opäť to tu nebudeme rozoberať, ak ľa to zaujíma, skús si vyhľadať “multiple inheritance” a “diamond problem”.

Vráťme sa teraz k nášmu projektu. Základná trieda `Widget` by mohla vyzerať takto:

```
1 struct Widget {
2     static SDL_Point mouse;
3
4     SDL_Rect rect;
5     int fixedHeight = 0;
6     int fixedWidth = 0;
7     int padding = 0;
8
9     virtual ~Widget(){}
10
11    virtual void resize(SDL_Rect _rect);
12    virtual void render(SDL_Renderer *r){}
13    virtual void onMouseMotion(SDL_MouseMotionEvent& e){}
14    virtual void onMouseDown(SDL_MouseButtonEvent& e){}
15    virtual void onMouseUp(SDL_MouseButtonEvent& e){}
16};
```

Máme statickú premennú `mouse`, do ktorej v hlavnom programe pri spracovaní udalostí zapíšeme polohu myši. Každý widget má vlastný obdĺžnik `rect`, ktorý zaberá na obrazovke a okraj `padding`. Môže mať nastavený `fixedWidth` alebo `fixedHeight`, vtedy má vždy danú veľkosť (napr. výška gombíka je vždy rovnaká). Ak nie sú nastavené, tak sa v danom smere môže naťahovať.

Všetky metódy tu majú prázdne telo: funkcionality bude až v zdelených triedach. Výnimkou je `resize`, ktorá nastaví obdĺžnik zmenšený o okraje takto:

```
1 void Widget::resize(SDL_Rect _rect) {
2     rect = _rect;
3     rect.x += padding;
4     rect.y += padding;
5     rect.w -= 2 * padding;
6     rect.h -= 2 * padding;
7 }
```

Ďalej si spravíme triedu `Layout` a z nej zdelené triedy `VLayout` a `HLayout`:

<sup>7</sup>[https://en.cppreference.com/w/cpp/language/derived\\_class](https://en.cppreference.com/w/cpp/language/derived_class)

## Dedičnosť

```
1 struct Layout : Widget {
2     std::vector<Widget *> children;
3     std::vector<SDL_Rect> rects;
4
5     ~Layout();
6     void render(SDL_Renderer *r);
7
8     Layout& operator<<(Widget *w); // pridá w na koniec children
9
10    void onMouseMotion(SDL_MouseMotionEvent& e);
11    void onMouseDown(SDL_MouseButtonEvent& e);
12    void onMouseUp(SDL_MouseButtonEvent& e);
13 };
14
15 struct HLayout : Layout {
16     void resize(SDL_Rect _rect);
17 };
18
19 struct VLayout : Layout {
20     void resize(SDL_Rect _rect);
21 };
```

`Layout` si pamäta zoznam detí a obdĺžnikov, ktoré zaberajú. Všetky metódy `Layout` len zavolajú príslušné metódy detí, napr.

```
1 void Layout::onMouseDown(SDL_MouseButtonEvent &e) {
2     for (auto &c : children)
3         if (SDL_PointInRect(&mouse, &(c->rect))) c->onMouseDown(e);
4 }
```

Hlavnú prácu robia zdelené `VLayout::resize()` a `HLayout::resize()`, ktoré na základe celého obdĺžnika prerátajú obdĺžníky detí: tie, ktoré majú fixný rozmer, nechajú fixný a ostatné rovnomerne rozdelia medzi zvyšok. Nakoniec sa zavolá

```
1 for (int i = 0; i < children.size(); i++) children[i]->resize(rects[i]);
```

Tu je treba sa rozhodnúť, kto bude "vlastniť" widgety v layoute. Inými slovami, kto je zodpovedný za to, aby sa správne zmazali, keď ich už netreba. Ja som sa rozhadol, že ich bude vlastniť layout, to znamená, že deštruktory layoutu zavolá deštruktory detí. Potom si treba ustrážiť, aby sa už inde v programe widgety, ktoré sa pridajú do layoutu, nemazali.

Ešte budeme potrebovať gombíky, napr.

```
1 struct Button : Widget {
2     static TTF_Font *font;
3     Text label, labelDisabled;
4     std::string txt;
5     bool clicked;
6     bool disabled;
7     std::function<void(void)> onClick = [](){};
8
9     SDL_Color bgNormal, bgHovered, bgClicked, bgDisabled, fgNormal, fgDisabled;
10
11    Button(const std::string &_txt=" „");
12
13    void render(SDL_Renderer *r);
14    void onMouseMotion(SDL_MouseMotionEvent& e);
15    void onMouseDown(SDL_MouseButtonEvent& e);
```

```
16     void onMouseUp(SDL_MouseButtonEvent& e);  
17 }
```

Gombík si pamätá, či sa naňho práve kliklo (`onMouseDown` nastaví `clicked` na `true` a `onMouseUp` na `false`) a či nie je vypnutý a podľa toho sa pri volaní `render` vykreslí. Navyše má premennú `onClick`, čo je lambda, ktorá sa zavolá, keď sa na gombík klikne.

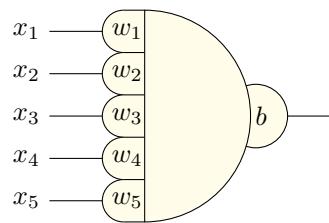
**Úloha 139.** Urob program pre Breakthrough, ktorý vyzerá ako screenshot na začiatku tejto kapitoly: má gombíky na novú hru, vrátenie ťahu, otočenie šachovnice a skončenie programu.

Ak si prišiel až sem, gratulujem. Napísal si celkom veľký, použiteľný program. Určite máš veľa nápadov, ako by sa dal vylepšiť. Pri programoch tejto veľkosti (a väčších) je najväčšia zábava, ak sa spojíte viacerí a skúsíte všetky svoje nápady spoločne realizovať.

## 39 Neurónové siete

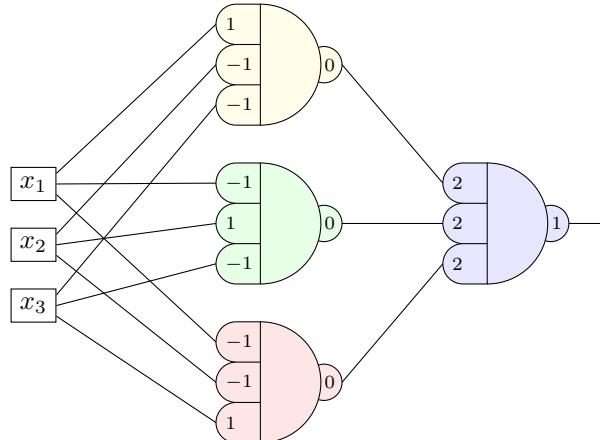
Neurónové siete sú tak úspešné pri riešení najrôznejších úloh, že sa stali synonymom pre celú umelú inteligenciu. V tejto časti ti poviem, ako (zjednodušene) neurónové siete fungujú a pri tom ti navyše ukážem zopár užitočných vecí.

Neurónová sieť je program, ktorý má vstup a výstup. Vstup aj výstup tvorí niekoľko čísel. Namiesto za sebou napísaných príkazov, ako je to pri programoch v bežných programovacích jazykoch, je neurónová sieť zložená zo vzájomne prepojených súčiastok, ktoré sa volajú neuróny, lebo pripomínajú mozgové bunky.



Do neurónu prichádza zvonka  $n$  vstupov (čísel)  $x_1, x_2, \dots, x_n$ . Neurón má pre každý vstupný pin určenú váhu  $w_i$ . Keď príde vstup, v neuróne sa spočíta súčet  $w_1x_1 + w_2x_2 + \dots + w_nx_n = \sum_{i=1}^n w_i x_i$ . Ak  $\sum_{i=1}^n w_i x_i \geq b$ , na výstupe bude 1, inak 0.

Dajme tomu, že chceme riešiť nasledovnú (veľmi jednoduchú) úlohu: *Na vstupe sú 3 čísla. Zistite, či niektoré z nich je väčšie alebo rovné ako súčet zvyšných dvoch.* Riešením môže byť neurónová sieť, ktorá bude vyzerať takto:



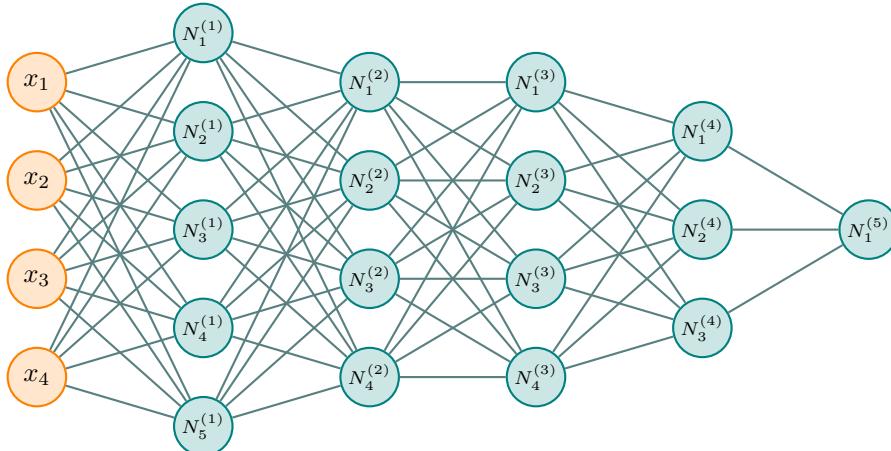
Žltý neurón počíta  $x_1 - x_2 - x_3$ : ak je  $x_1 \geq x_2 + x_3$ , na výstupe bude 1, inak 0. Podobne zelený neurón dá na svoj výstup 1, ak  $x_2 \geq x_1 + x_3$  a červený neurón má na výstupe 1, ak  $x_3 \geq x_1 + x_2$ . Ak má aspoň jeden z neurónov na prvej vrstve výstup 1, na výstupe modrého neurónu je 1, inak je tam 0.

Takýmto spôsobom sa dajú poskladať zložité siete z veľa neurónov, ktoré dokážu riešiť aj celkom zložité úlohy. V porovnaní s normálnymi programovacími jazykmi tu síce nemáme cykly ani podmienky, ale na druhej strane, sieť vyrábame pre fixnú veľkosť vstupu. Zásadná otázka však je, prečo by sme mali produkovať programy takýmto divným spôsobom, keď ich môžeme naprogramovať v nejakom pohodlnom programovacom jazyku. Ide o to, že neplánujeme zostavovať neurónové siete ručne. Našim cieľom bude napísať program (teraz myslím normálny program v C++), ktorý na vstupe dostane úlohu a vyrobí neurónovú sieť, ktorá ju rieši. Ako môže na vstupe "dostať úlohu" a ako takú neurónovú sieť vyrobí, o tom budeme hovoriť o chvíli. Najprv si podme spraviť zvyčajné prípravné práce a naprogramujme si výpočet neurónovej siete.

## Odbočka o maticiach a vektoroch

Skôr, ako sa pustíme do programovania, je dobré si premyslieť, ako budeme veci označovať; pohodlné značenie býva často polovica úspechu. Ak mám neurón s troma vstupmi  $x_1, x_2, x_3$ , tak vo svojom tele počíta  $w_1x_1 + w_2x_2 + w_3x_3$ . Ak si spomenieš na kapitolu 31, kde sme hovorili o vektoroch, tak ak by som si predstavil, že  $\vec{w} = [w_1, w_2, w_3]$  a  $\vec{x} = [x_1, x_2, x_3]$  sú vektory v 3-rozmernom priestore, tak neurón počíta skalárny súčin  $\vec{w} \cdot \vec{x}$ . Keby mal neurón  $n$  vstupov, tiež si môžeme predstaviť, že je to vektor v nejakom priestore, až na to, že teraz ten priestor bude  $n$ -rozmerný. Vôbec nevadí, že si taký priestor nevieme predstaviť, stačí nám vedieť, že vektor v takom priestore je vyjadrený pomocou  $n$  súradníck<sup>1</sup> a že skalárny súčin sa počíta rovnako. Namiesto šípkov hore budem vektory značiť tlstým fontom, takže ak váhy neurónu sú  $\mathbf{w} = [w_1, \dots, w_n]$  a vstup je  $\mathbf{x} = [x_1, \dots, x_n]$ , tak v neuróne sa počíta skalárny súčin  $\mathbf{w} \cdot \mathbf{x} = \sum_{i=1}^n w_i x_i$ .

Aby sa nám veci dobre rátali, tak neurónové siete, ktoré budeme vyrábať, sa budú skladať z vrstiev: na prvej vrstve budú neuróny  $N_1^{(1)}, N_2^{(1)}, \dots$ , ktoré všetky budú napojené na vstup (ale každý s inými váhami), potom na druhej vrstve budú neuróny  $N_1^{(2)}, N_2^{(2)}, \dots$ , ktoré budú všetky napojené na výstupy neurónov z prvej vrstvy atď.



Preto budeme často potrebovať vyrátať hodnotu viacerých neurónov na tom istom vstupe. Napr. na obrázku hore je vstup  $\mathbf{x} = [x_1, x_2, x_3, x_4]$  a v prvej vrstve máme neuróny  $N_1^{(1)}, N_2^{(1)}, N_3^{(1)}, N_4^{(1)}, N_5^{(1)}$ . Váhy neurónu  $N_1^{(1)}$  k vstupom  $x_1, \dots, x_4$  si označíme  $\mathbf{w}_1 = [w_{1,1}, w_{1,2}, \dots, w_{1,4}]$ , takže neurón  $N_1^{(1)}$  ráta  $\sum_{i=1}^4 w_{1,i} x_i = \mathbf{w}_1 \cdot \mathbf{x}$ . Ak chcem vyhodnotiť všetky neuróny z prvej vrstvy, potrebujem vyrátať skalárne súčiny  $\mathbf{w}_1 \cdot \mathbf{x}, \mathbf{w}_2 \cdot \mathbf{x}, \dots, \mathbf{w}_m \cdot \mathbf{x}$ . Aby sme si skrátili zápis, vektory  $\mathbf{w}_1, \dots, \mathbf{w}_m$  si napíšeme ako riadky do tabuľky s  $m$  riadkami a  $n$  stĺpcami, pripíšeme si jeden stĺpec s vektorom  $\mathbf{x}$  a výsledky sa zapíšeme do stĺpca<sup>2</sup> takto:

	$W$				$\mathbf{x}$	$W \cdot \mathbf{x}$
$\mathbf{w}_1$	$w_{1,1}$	$w_{1,2}$	$w_{1,3}$	$w_{1,4}$	$x_1$	$\mathbf{w}_1 \cdot \mathbf{x}$
$\mathbf{w}_2$	$w_{2,1}$	$w_{2,2}$	$w_{2,3}$	$w_{2,4}$	$x_2$	$\mathbf{w}_2 \cdot \mathbf{x}$
$\mathbf{w}_3$	$w_{3,1}$	$w_{3,2}$	$w_{3,3}$	$w_{3,4}$	$x_3$	$\mathbf{w}_3 \cdot \mathbf{x}$
$\mathbf{w}_4$	$w_{4,1}$	$w_{4,2}$	$w_{4,3}$	$w_{4,4}$	$x_4$	$\mathbf{w}_4 \cdot \mathbf{x}$
$\mathbf{w}_5$	$w_{5,1}$	$w_{5,2}$	$w_{5,3}$	$w_{5,4}$		$\mathbf{w}_5 \cdot \mathbf{x}$

V matematike sa tabuľka čísel volá matica (*matrix*). Ak teda máme maticu  $W$ , kde sú v riadkoch váhy  $m$  neurónov  $\mathbf{w}_1, \dots, \mathbf{w}_m$  a vektor  $\mathbf{x}$  je vstup, obrázku hore budeme skrátene hovoriť, že maticu  $W$  vynásobíme vektorom  $\mathbf{x}$  a písť  $W \cdot \mathbf{x}$ . S pomocou tohto zápisu vieme úsporne vyjadriť výpočet celej vrstvy: ak  $i$ -ty neurón porovnáva súčin  $\mathbf{w}_i \cdot \mathbf{x}$  s hodnotou  $b_i$ , tak rovnako dobre môžeme porovnávať  $\mathbf{w}_i \cdot \mathbf{x} - b_i$  s nulou. Preto

<sup>1</sup>Tu je vidieť, prečo sa dynamické pole v C++ volá `std::vector`: je to dátová štruktúra, kde sa dá ukladať  $n$ -rozmerný vektor.

<sup>2</sup>prečo práve do stĺpca bude zrejmé o chvíľu

keď si hodnoty  $b_i$  zoradíme do vektora  $\mathbf{b} = [b_1, \dots, b_m]$ , tak výpočet celej vrstvy neurónov vieme napísť ako  $W \cdot \mathbf{x} - \mathbf{b}$ .

Niekedy sa nám bude hodíť aj počítanie vrstvy neurónov na viacerých vstupoch naraz. Ak ma zaujímajú vstupné vektory  $\mathbf{x}_1, \dots, \mathbf{x}_s$ , môžem si ich zapísť po stĺpcach do matice  $X$  a výsledok si opäť uložiť do  $s$  stĺpcov; dosťanem tak maticu rozmerov  $m \times s$ :

$$\begin{array}{c}
 \begin{array}{c} W \\ \hline \mathbf{w}_1 & w_{1,1} & w_{1,2} & w_{1,3} & w_{1,4} \\ \mathbf{w}_2 & w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} \\ \mathbf{w}_3 & w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} \\ \mathbf{w}_4 & w_{4,1} & w_{4,2} & w_{4,3} & w_{4,4} \\ \mathbf{w}_5 & w_{5,1} & w_{5,2} & w_{5,3} & w_{5,4} \end{array} \cdot \begin{array}{c} \mathbf{x}_1 & x_{1,1} & x_{1,2} & x_{1,3} \\ \mathbf{x}_2 & x_{2,1} & x_{2,2} & x_{2,3} \\ \mathbf{x}_3 & x_{3,1} & x_{3,2} & x_{3,3} \\ \mathbf{x}_4 & x_{4,1} & x_{4,2} & x_{4,3} \end{array} = \begin{array}{c} W\mathbf{x}_1 & W\mathbf{x}_2 & W\mathbf{x}_3 \\ \mathbf{w}_1\mathbf{x}_1 & \mathbf{w}_1\mathbf{x}_2 & \mathbf{w}_1\mathbf{x}_3 \\ \mathbf{w}_2\mathbf{x}_1 & \mathbf{w}_2\mathbf{x}_2 & \mathbf{w}_2\mathbf{x}_3 \\ \mathbf{w}_3\mathbf{x}_1 & \mathbf{w}_3\mathbf{x}_2 & \mathbf{w}_3\mathbf{x}_3 \\ \mathbf{w}_4\mathbf{x}_1 & \mathbf{w}_4\mathbf{x}_2 & \mathbf{w}_4\mathbf{x}_3 \\ \mathbf{w}_5\mathbf{x}_1 & \mathbf{w}_5\mathbf{x}_2 & \mathbf{w}_5\mathbf{x}_3 \end{array} \end{array}$$

Ak sa na tento obrázok pozrie matematik, tak vidí, že robíme presne to, čo sa v matematike deje pri násobení matíc<sup>3</sup>. Ak mám maticu  $A$  rozmerov  $m \times n$  (rozmery matice budem písť ako riadky  $\times$  stĺpce) a maticu  $B$  rozmerov  $n \times s$ , tak  $A \cdot B = C$  rozmerov  $n \times s$  tak, že v  $i$ -tom riadku a  $j$ -tom stĺpci matice  $C$  je skalárny súčin  $i$ -tého riadku matice  $A$  a  $j$ -tého stĺpca matice  $B$ , t.j.

$$c_{i,j} = \sum_{k=1}^n a_{i,k} \cdot b_{k,j}$$

Všimni si, že nemôžeme vynásobiť hocijaké dve matice, počet stĺpcov prvej musí byť rovnaký ako počet riadkov druhej. Z toho hned' vidno, že vo všeobecnosti  $A \cdot B \neq B \cdot A$ , takže treba dávať pozor na to, v akom poradí sú matice pri násobení zapísané. To, že sme si vektory v druhej matici písali ako stĺpce nám ale zaručí, že  $A \cdot (B \cdot C) = (A \cdot B) \cdot C$  (skús si dokázať, že to naozaj platí), takže môžeme napísť  $A \cdot B \cdot C$  a nestarať sa o zátvorky.

Posledná vec v tejto odbočke je značenie  $A^\top$ , ktoré znamená otočenie (transpozíciu) matice  $A$ , t.j. výmenu riadkov a stĺpcov:

$$A = \begin{array}{|c|c|c|} \hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline \end{array} \quad A^\top = \begin{array}{|c|c|} \hline 1 & 4 \\ \hline 2 & 5 \\ \hline 3 & 6 \\ \hline \end{array}$$

Naspäť k programovaniu. Keď si uvedomíš, že vektor je vlastne špeciálna matica, ktorá má len jeden stĺpec, na programovanie neurónových sietí nám bude stačiť naprogramovať prácu s maticami.

**Úloha 140.** V kapitole 22 sme vyrobili triedu [Tabulka](#). Môžeš sa ľuďou inspirovať a vypočítať triedu [Matrix](#), ktorá bude vedieť pracovať s maticami čísel [double](#). Pridáme k nej niekoľko užitočných metód, takže by mala splňať takúto špecifikáciu:

<sup>3</sup>Spomeň si, ako sme v kapitole 32 vyrábali na strane 185 hešovaciu funkciu. Vidíš tam nejakú podobnosť?

```

1 using Num = double;
2
3 struct Matrix {
4     int n,m;
5     Num* _data;
6
7     // konštruktory
8     Matrix(int _n = 1, int _m = 1, Num init = (Num)(0));
9     Matrix(int _n, int _m, const std::vector<Num>&);
10    Matrix(const Matrix&);
11    Matrix(Matrix&&);
12
13    // operátory priradenia
14    Matrix& operator=(const Matrix&);
15    Matrix& operator=(Matrix&&);
16
17    // prístup k dátam
18    Num& operator()(int i, int j);
19    const Num& operator()(int i, int j) const;
20
21    // deštruktor
22    ~Matrix();
23
24    template <typename F> Matrix& fill(F f) { ... }
25    template <typename F> Matrix& apply(F f) { ... }
26
27    Matrix& operator+=(const Matrix& b);
28    Matrix& operator-=(const Matrix& b);
29    Matrix& addMultiple(Num delta, const Matrix& b);
30    Matrix& addRow(const Matrix& b);
31    Matrix& addColumn(const Matrix& b);
32    Matrix transposed() const;
33 };
34

```

Konštruktor `Matrix(Matrix&&)` je move constructor, pozri poznámku pre fajnšmekrov na strane 88. Na prístup k dátam treba dve funkcie, aby sme mohli mať aj konštantné aj nekonštantné premenné. Funkcia `addMultiple` pripočíta `delta`-násobok matice `b`. Funkcia `addRow` dostane ako parameter riadok `b` rozmerov  $1 \times m$  a pripočíta ho ku každému riadku matice. Podobne `addColumn` so stĺpcom. Funkcia `fill` vyplní maticu funkciou `f`. Parameter `f` je lambda (alebo funktor alebo funkcia), ktorá má dva parametre typu `int` a na všetky prvky matice sa zavolá `(*this)(i, j) = f(i, j)`. Podobne pri `apply` má `f` jeden parameter, ktorý ako vstup dostane hodnotu `(*this)(i, j)` a modifikuje ju `(*this)(i, j) = f((*this)(i, j))`.

Okrem toho chceme samostatné funkcie

```

1 void multInto(const Matrix& a, const Matrix& b, Matrix& res); // uloží a.b do res
2 Matrix operator*(const Matrix& a, const Matrix& b);
3 Matrix operator+(const Matrix& a, const Matrix& b);

```

Keď už máme vyriešenú prácu s maticami, môžeme začať pracovať na neurónovej sieti. Základ bude vrstva neurónov, na začiatok urobme niečo jednoduché

## Neurónové siete

```
1 struct Layer {
2     int n1;          // počet neurónov vo vrstve
3     int n0;          // veľkosť predchádzajúcej vrstvy (vstupu)
4     Matrix W, b;   // W: n1 x n0, kde w[i,j] = váha i-teho neurónu k j-temu vstupu
5                 // b: stĺpec n1 x 1
6     Layer(int _n0, int _n1);           // konštruktor
7     Matrix eval(const Matrix& X);    // vstup n0 x s, výstup n1 x s hodnoty neurónov
8 }
```

Vrstva bude mať `n1` neurónov a bude napojená na `n0` vstupných hodnôt. Váhy neurónov budú zapamätané v matici `W`. Funkcia `eval` má ako parameter maticu, v ktorej sú v stĺpcoch uložené vstupy a vráti maticu, kde sú v stĺpcoch uložené výsledné hodnoty neurónov. Ako to čo najjednoduchšie spraviť? Ak vyrátame  $W \cdot X$ , dostaneme maticu rozmerov  $n_1 \times s$ , kde v riadku  $i$  a stĺpci  $j$  je  $\mathbf{w}_i \cdot \mathbf{x}_j$ . Od každého stĺpca potrebujeme odrátať vektor `b`, čím dostaneme pre každý neurón a každý vstup hodnotu, ktorú treba porovnať s nulou. Môžeme preto písť (v `b` chceme mať hodnoty  $-b_i$  pre každý neurón)

```
1 Matrix Layer::eval(const Matrix &X) {
2     Matrix res = W * X;
3     res.addColumn(b);
4     res.apply([](double x) { return (x > 0) ? 1.0 : 0.0; });
5     return res;
6 }
```

Zápis `(x > 0) ? 1.0 : 0.0` som tu ešte nespomíнал, ale je to skrátený zápis podmienky v jednom výraze. `(♣) ? ♥ : ♦` je výraz, ktorého hodnota je ♥, ak ♣ je `true` a ♦ inak. Tu som to použil namiesto dlhšieho `if (x > 0) return 1.0; else return 0.0;`

**Úloha 141.** Zober si neurónovú sieť z úvodného príkladu. Napiš program, ktorý na vstupe dostane číslo  $n$  a potom  $n$  celých čísel. Program vytvorí neurónovú sieť s dvoma vrstvami (v prvej bude  $n$  neurónov, v druhej jeden), ktorá zistí, či na vstupe existuje číslo, ktoré je väčšie ako súčet zvyšných. Túto sieť spustí na vstupných hodnotách a výsledok vypíše.

Na pohodlnnejšiu prácu s jednotlivými vrstvami si ich môžeme zabaliť do jedného typu takto:

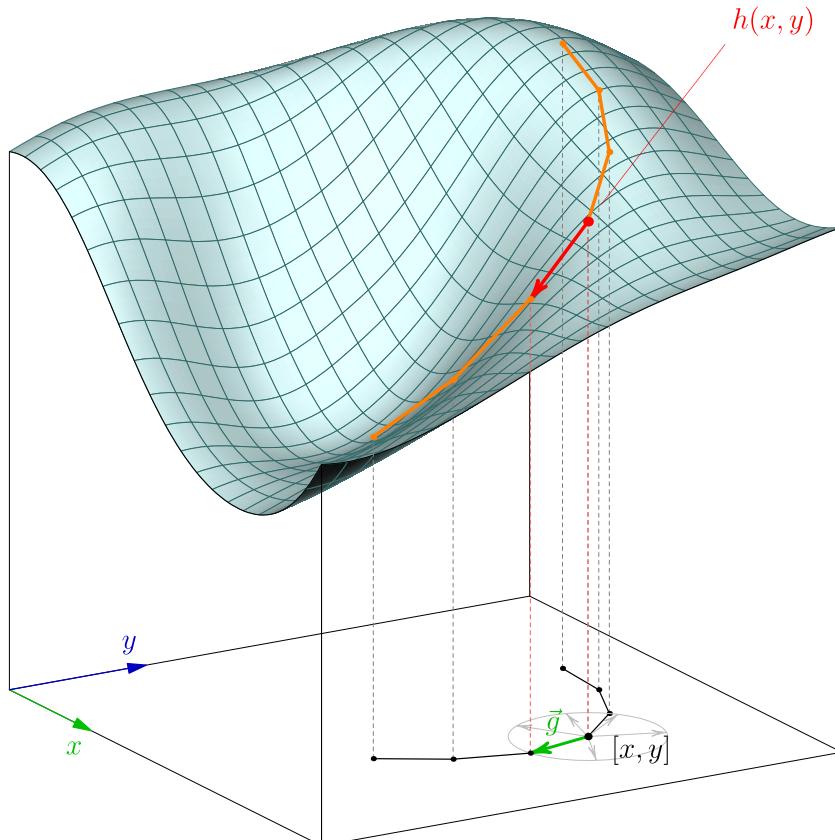
```
1 struct Network {
2     const unsigned long int h; // počet vrstiev
3     const vector<int> n;      // veľkosť vrstiev, n[0] je veľkosť vstupu
4     vector<Layer> layers;    // vrstvy
5     Network(initializer_list<int>); // konštruktory
6     Network(const vector<int>&);
7     Matrix feed(const Matrix &);
8 }
9
10 Network::Network(initializer_list<int> v) : h{v.size() - 1}, n{v} {
11     for (int i = 0; i < h; i++) layers.emplace_back(n[i], n[i + 1]);
12 }
13
14 Network::Network(const vector<int>& v) : h{v.size() - 1}, n{v} {
15     for (int i = 0; i < h; i++) layers.emplace_back(n[i], n[i + 1]);
16 }
17
18 Matrix Network::feed(const Matrix &x) {
19     Matrix res(x);
20     for (auto &l : layers) res = l.eval(res);
21     return res;
22 }
```

Premenná typu `Network` si pamäta počet vrstiev `h`, ich veľkosti `n` a pole vrstiev `layers`. Hlavná metóda je `feed`, ktorá zoberie maticu vstupných hodnôt a postupne ju “prebubble” cez všetky vrstvy. V tomto zápisе som použil dve nové veci, ktoré s neurónovými siefami nesúvisia. Prvou z nich je typ `initializer_list` definovaný v rovnomennej knižnici (`#include<initializer_list>`). Nemá veľa funkcionality (v podstate má len veľkosť `size()` a iterátory `begin()` a `end()`), ale dá sa v konštruktore použiť na to, aby sme premennú mohli inicializovať zoznamom konštant v kučeravých zátvorkách. Takže napr. volanie `Newtork net{n, n, 1}`; vyrobí sieť so vstupom veľkosti `n` a dvoma vrstvami s `n` a jedným neurónom. V konštruktore sa najprv nastaví `h` (je tam `v.size()-1`, lebo `v` obsahuje aj veľkosť vstupu) a initializer list `v` sa pošle do konštruktora vektora `n`. Druhá nová funkcia je metóda vektora `emplace_back`. Rovnako ako `push_back` pridá prvok na koniec vektora, ale kým `push_back` by najprv vyrobil premennú typu `Layer` a potom ju presunul do vektora, `emplace_back` dostane parametre, ktoré sa použijú v konštruktore na vyrobenie premennej priamo na mieste (takže je to spravidla trochu efektívnejšie).

Keď už vieme, ako neurónovú sieť spustiť, podme sa pozrieť na to, ako vyrobiť neurónovú sieť pre nejakú úlohu. Začnime skromne. Chcem mať neurónovú sieť s ôsmimi vstupmi, ktoré budú reprezentovať zápis 8-bitového čísla  $N$  v dvojkovej súštave. Sieť má mať jeden výstup, ktorý hovorí, či  $N$  je prvočíslo. Namiesto toho, aby sme sieť ručne zostavovali, si iba povieme, kolko vrstiev a aké počty neurónov v nich chceme a budeme sa snažiť napisať program, ktorý váhy neurónov doráta tak, aby sieť čo najlepšie fungovala.

Na to si treba v prvom rade premyslieť, ako povedať, či sieť dobre funguje. Keďže máme iba 256 možných vstupov, je to jednoduché: spustíme sieť na všetkých vstupoch a spočítame, kolkokrát sa pomýlila.

Druhá vec je, ako vyrátať váhy neurónov. Budeme to robiť podobne, ako keď sme v kapitole 31 simulovali eróziu. Mali sme tam výškovú mapu, ktorá nám pre bod  $[x, y]$  v rovine určovala výšku terénu v tomto bode  $h(x, y)$ . Pre kvapku sme si pamätali jej polohu  $[x, y]$ .



## Neurónové siete

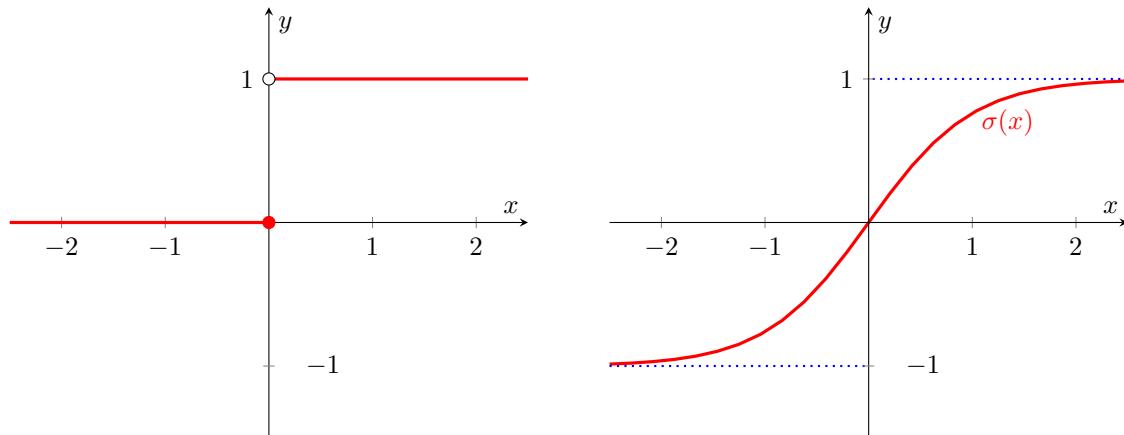
Pohybovať sme sa vedeli v rovine, t.j. z bodu  $[x, y]$  sme sa mohli pohnúť smerom  $\vec{\Delta} = [\Delta_x, \Delta_y]$  do bodu  $[x + \Delta_x, y + \Delta_y]$ . Vždy sme si vybrali gradient, t.j. smer, ktorým hodnota  $h(x, y)$  najviac klesá, a pohnúť sme sa o kúsok tým smerom.

Keby sme mali najjednoduchšiu neurónovú sieť, ktorá by mala iba jeden neurón s jedným vstupom, mohli by sme si predstaviť rovnaký obrázok. Bod  $[x, y]$  v rovine by zodpovedal sieti, v ktorej je váha neurónu  $x$  a porovnávacia hodnota  $b$  je  $y$ . Výška v bode  $[x, y]$  by udávala chybu siete, t.j. na kolkátiach vstupoch sa pomýli. Keď sa hýbeme bodom v rovine, hýbeme sa v priestore všetkých možných kombinácií váh pre našu sieť. Teraz si zober neurónovú sieť zo začiatku tejto kapitoly: má 4 neuróny a každý z nich má 4 parametre (3 vstupné váhy a hranicu  $b_i$ ), takže dokopy viem celú sieť opísť 16 číslami. Tie si viem prestaviť ako súradnice bodu  $[x_1, x_2, \dots, x_{16}]$  v 16-rozmernom priestore. Takýto obrázok už neviem nakresliť, ale je v podstate podobný: namiesto dvojrozmernej roviny sa budem hýbať v 16-rozmernej, a pre každý bod mi bude moja výšková mapa udávať chybu siete. Pohnúť sa nejakým smerom znamená, že ku každej súradnici pripočítam nejaké  $\Delta_i$ .

Celý plán na nájdienie váh neurónov by vyzeral takto: začneme v nejakom náhodnom bode (t.j. so sieťou, ktorá má náhodné váhy) a potom opakujeme cyklus: zistíme, ktorým smerom chyba siete klesá a pohneme sa o kúsok tým smerom. Keď bude sieť dosť dobrá, tak môžeme skončiť. Ako zistíť smer klesania chyby? Najjednoduchšie je zobrať si nejaký náhodný smer (t.j. každý parameter siete posunúť o náhodný malý kúsok) a zistíť, ako dobrá je sieť. Ak sme sa zlepšili, tak sa tam presunieme, ak nie, ostaneme stáť na mieste.

Toto má jeden zásadný problém: to, aká dobrá je sieť, meriame počtom vstupov, na ktorých sa pomýli. To je celé číslo, ktoré má v našom prípade iba 256 možností. Keď sa pohnem iba o malý kúsok, je veľká šanca, že sa počet chyb nezmení. Ak sa pohnem o veľký kus, je veľká šanca, že dolinu preskočím a ocitnem sa na náhodnom mieste. Inými slovami, naša výšková mapa nevyzerá tak, ako na predchádzajúcom obrázku, ale skladá sa z veľa rovných plošín. No a keď sme na takej plošine, nevieme zistíť, ktorým smerom sa vydať.

Problém je už v samotnom neuróne, konkrétnie v jeho rozhodovacej funkcií, ktorá vráti 1 ak je hodnota  $\mathbf{w} \cdot \mathbf{x} - b > 0$  a 0 inak. Ak vähy  $\mathbf{w}$  zmeníme iba veľmi málo, aj  $\mathbf{w} \cdot \mathbf{x} - b$  sa zmení len o málo, takže je len malá šanca, že sa hodnota neurónu zmení – sme na plošine. Rozhodovacia funkcia je na obrázku vľavo: pre kladné hodnoty je výsledok 1, pre záporné 0. Ak by sme namiesto toho zobražovali rozhodovaciu funkciu ako tá na obrázku vpravo a neurón by ráhal hodnotu  $\sigma(\mathbf{w} \cdot \mathbf{x} - b)$ , mali by sme podobnú vlastnosť: pre kladné hodnoty  $\mathbf{w} \cdot \mathbf{x} - b$  je výsledok takmer 1 a pre záporné takmer -1. Funkcia sa navyše vždy zvažuje smerom k nule, takže vieme, ktorým smerom sa treba pohnúť.



Vpravo som použil funkciu, ktorá sa volá *hyperbolický tangens*<sup>4</sup> a je dostupný z knižnice `cmath` ako `tanh(x)`. Teraz výsledok každého neurónu, a tým pádom aj celej siete, nebude 0 alebo 1, ale číslo z intervalu (-1, 1). V programe to znamená iba toľko, že funkciu `Layer::eval` zmením takto:

<sup>4</sup> je to funkcia

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1},$$

```

1 Matrix Layer::eval(const Matrix &X) {
2     Matrix res = W * X;
3     res.addColumn(b);
4     res.apply([](double x) { return tanh(x); });
5     return res;
6 }
```

Výsledok celej siete budem interpretovať tak, že záporné hodnoty beriem ako odpoveď *nie* a kladné ako *áno*.

Ked' budem vyhodnocovať, ako dobrá je sieť, nebudem počítať počet vstupov, na ktorých sa pomýli, ale budem brať do úvahy kompletné hodnoty výstupov (t.j. ako čísla z intervalu  $(-1, 1)$ ). Pre každú hodnotu vo výslednej matici si zrátam jej chybu, t.j. rozdiel oproti správnej odpovedi. Rozdiel ale môže byť kladný aj záporný, čo môže robiť problémy (dve veľké chyby sa navzájom odčítajú). Preto<sup>5</sup> namiesto rozdielu zoberiem druhú mocninu rozdielu (je vždy kladná a ráta sa s ňou lepšie, ako napr. s absolútou hodnotou). Tomuto sa vraví *stredná kvadratická odchýlka (mean square error)*: ak mám čísla  $\mathbf{a} = [a_0, a_1, \dots, a_{n-1}]$  a  $\mathbf{b} = [b_0, b_1, \dots, b_{n-1}]$ , tak  $\text{mse}(\mathbf{a}, \mathbf{b}) = \sum_{i=0}^{n-1} (a_i - b_i)^2$ .

V programe si spravím funkciu, ktorá bude porovnávať dve matice (aby výsledok nezávisel od veľkosti matice, nakoniec ho vydelím rozmermi):

```

1 double mse(const Matrix &answer, const Matrix &truth) {
2     // obe matice musia mať rovnaké rozmery
3     const int n = truth.n, m = truth.m;
4     double err = 0;
5     for (int i = 0; i < n; i++) {
6         for (int j = 0; j < m; j++) {
7             double tmp = truth(i, j) - answer(i, j);
8             err += tmp * tmp;
9         }
10    err /= (double)(n * m);
11    return err;
12 }
```

Teraz mám všetko pripravené na to, aby sme skompletizovali celý učiaci sa program. Najprv si vyrobíme maticu `data`, ktorá bude mať 8 riadkov a 256 stĺpcov: každý stĺpec bude mať hodnoty  $-1$  a  $1$  podľa toho, ako sú nastavené byty v binárnom zápisе čísla  $i$ . Budem mať aj maticu `truth`, ktorá bude mať jeden riadok a 256 stĺpcov: pre každé číslo tam bude 1 alebo  $-1$  podľa toho, či je alebo nie je prvočíslo. Ak si napíšeš jednoduchý test `prime()`, program by mohol vyzeráť napr. takto

```

1 int n = 8;
2 Matrix data(n, 1 << n), truth(1, 1 << n);
3
4 for (int i = 0; i < (1 << n); i++) {
5     truth(0, i) = prime(i) ? 1.0 : -1.0;
6     for (int j = 0; j < n; j++) {
7         data(j, i) = (i & (1 << j) > 0) ? 1.0 : -1.0;
8     }
}
```

Potom si vyrobím sieť s ôsmimi vstupmi. Dajme tomu, že za vstupom budem mať ešte tri vrstvy so 16, 8 a jedným neurónom (to bude výstup). Kedže budem potrebovať k sieti pridávať náhodný šum a potom ho možno

kde  $e \approx 2.71828$  je iracionálne číslo, podobne ako  $\pi$ . Podobne ako  $\pi$ , aj  $e$  sa prirodzene vynorí, keď začneš rátať nejaké veci, ale to teraz nie je dôležité. Namiesto tanh by sme mohli zobrať ľavia iných podobných funkcií, napr. `sigmoid`

$$y = \frac{1}{1 + e^{-x}}$$

<sup>5</sup>Sú na to aj iné, hlbšie, dôvody, ale tie tu nepotrebujueme riešiť.

## Neurónové siete

---

zobrať naspäť, najjednoduchšie je k triede `Network` pridať metódu `addNoise`, ktorá na vstupe dostane okrem iného seed pre náhodný generátor; takto viem ľahko urobiť tú istú postupnosť náhodných čísel (stačí poslať rovnaký seed). Spravil som to takto:

```
1 void Network::addNoise(unsigned int seed, int sgn, double val) {
2     mt19937 rnd;
3     uniform_real_distribution<> dis(-val, val);
4     rnd.seed(seed);
5     for (auto &l : layers)
6         for (auto m : {&l.W, &l.b})
7             m->apply([&](double x) { return x + sgn * dis(rnd); });
8 }
```

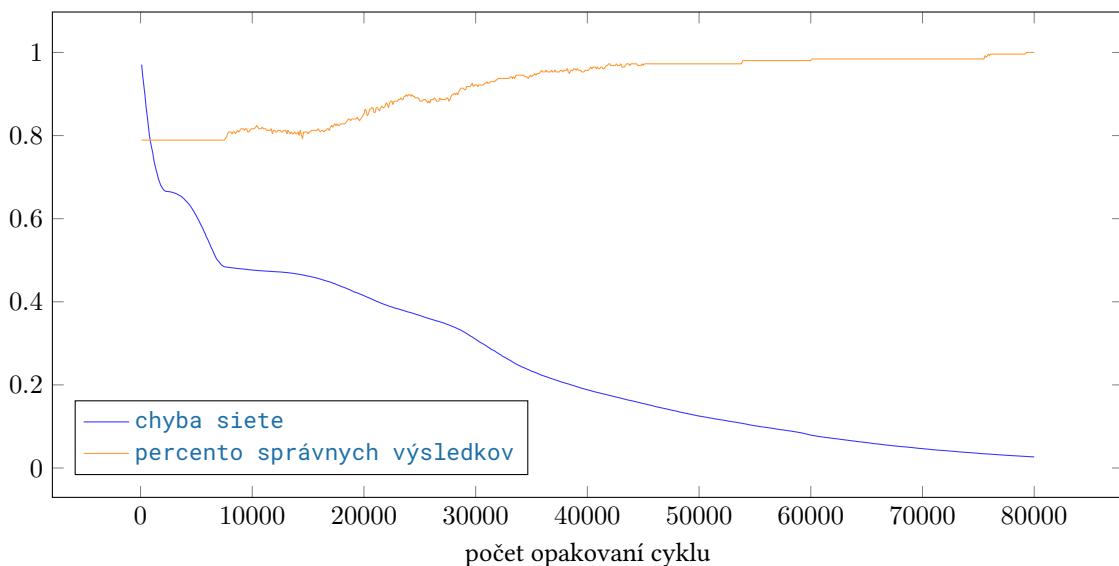
Na začiatku nastavím sieť na náhodný bod:

```
1 Network net{1, 16, 8, 1};
2 net.addNoise(random_device{}(), 1, 0.01);
```

V hlavnom cykle si vyrobím náhodné číslo. To použijem ako seed, s pomocou ktorého pridám do siete náhodný šum. Vypočítam si odpoveď siete na všetkých vstupoch a zrátam si chybu. Ak je väčšia ako doterajšia, odrátam od siete ten istý šum (použijem rovnaký seed, ale opačné znamienko).

```
1 mt19937 rnd(random_device{}());
2 double last_err = 1e50;
3 Matrix answer;
4 int cnt = 0;
5 do {
6     auto seed = rnd();
7     net.addNoise(seed);
8     answer = net.feed(data);
9     auto err = mse(answer, truth);
10    if (err < last_err)
11        last_err = err;
12    else
13        net.addNoise(seed, -1);
14    cout << cnt++ << " " << last_err << endl;
15 } while (last_err > 5e-3);
```

Ked' som to dal celé dokopy a vyskúšal, typicky som dostal niečo takéto:



Vidno, že chyba siete postupne klesá, ale treba na to pomerne veľa iterácií: v mojom prípade zrhuba 80000, kým sa sieť naučila všetky čísla správne. Zároveň vidno, že prvočíslo je pomerne málo, takže mať 80% úspešnosť dokáže aj úplne hlúpa sieť.

**Úloha 142.** Vyskúšaj si, ako sa zmení trénovanie, keď meníš veľkosť siete. Čo sa stane, ak je primalá? A čo ak je privelká?

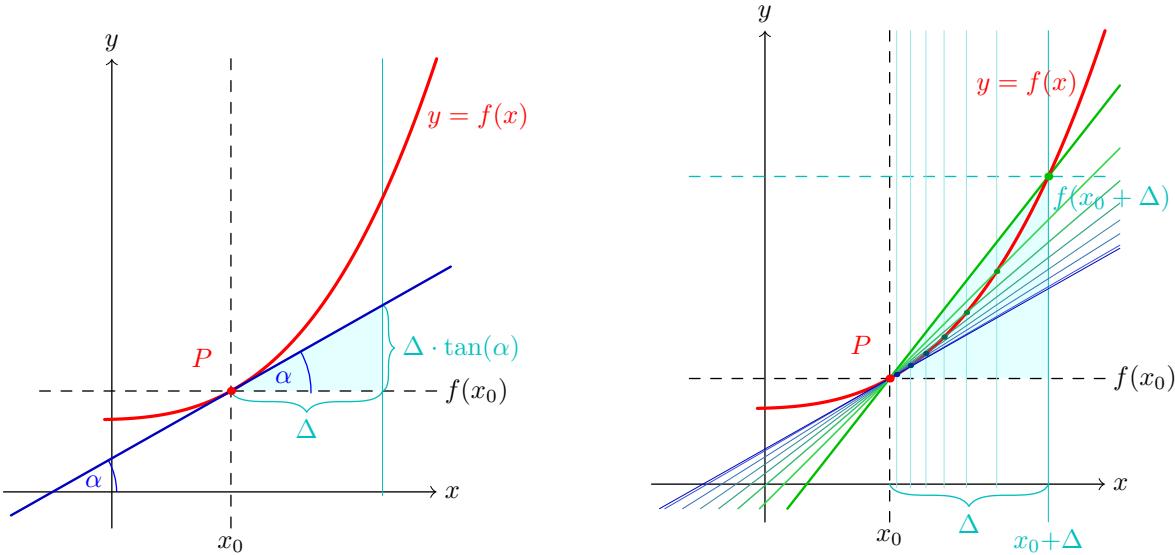
Toto bola, samozrejme, trochu hračkárska úloha, keďže sme sa starali iba o 256 rôznych vstupov. Chcelo by to skúsoť si väčšie siete, ale nás spôsob učenia, keď si vždy vyberáme náhodný smer, je veľmi pomalý. Preto skôr, ako pokročíme smerom k zaujímavejším úlohám, najprv vymyslíme spôsob, ako siete trénovať rýchlejšie. Konkrétnie budeme chcieť zrátať smer, ktorým výšková mapa klesá (a teda ktorým sa máme posunúť). Na to ale potrebujeme ešte jednu matematickú odbočku.

### Matematická odbočka: o deriváciách

Predstav si, že si nakreslím graf nejakej funkcie  $y = f(x)$ . Keď si zoberiem nejaké číslo  $x_0$ , bod  $P = [x_0, f(x_0)]$  leží na grafe funkcie. Teraz ma zaujíma dotyčnica<sup>6</sup> ku grafu v bode  $P$ . Konkrétnie, chceme vedieť, aký uhol  $\alpha$  zviera dotyčnica s osou  $x$ . Keby som ho poznal, tak viem, že ak sa na osi  $x$  pohnem o nejaký hodinieký kúsok  $\Delta$ , tak priamka stúpne o  $\Delta \cdot \tan(\alpha)$ . To  $\tan(\alpha)$  je nejaké číslo<sup>7</sup>, ktoré závisí iba od uhla  $\alpha$  (lebo pre všetky volby  $\Delta$  sú tyrkysové trojuholníky z ľavého obrázka podobné). Dotyčnica sa v domácnosti hodí: keby som pozeral iba veľmi blízko  $x_0$ , mohol by som sa tváriť, že namiesto (možno zložitej) funkcie  $f(x)$  mám jednoduchú priamku.

<sup>6</sup>t.j. priamka, ktorá má s grafom spoločný iba jeden bod

<sup>7</sup>Volá sa *tangens* a patrí medzi goniometrické funkcie, podobne ako sin a cos z kapitoly 17. Tangens je pomer protiľahlej a priľahlej strany pravouhlého trojuholníka, inými slovami  $\tan(\alpha) = \frac{\sin(\alpha)}{\cos \alpha}$ .



Ak neviem zrátať priamo dotyčnicu, môžem si skúsiť zrátať niečo, čo sa jej dosť podobá. Zoberiem si malý kúsok  $\Delta$  a spravím si zelenú priamku z pravého obrázka, ktorá prechádza bodom  $P$  a bodom  $[x_0 + \Delta, f(x_0 + \Delta)]$ . O tejto priamke ľahko zistím tangens jej uhla: je to  $\frac{f(x_0 + \Delta) - f(x_0)}{\Delta}$ . Čím menšie  $\Delta$  si zvolím, tým bude výsledná priamka vernejšie reprezentovať dotyčnicu.

Podme si to skúsiť zrátať napr. pre funkciu  $f(x) = x^2$ . Pre zvolené  $\Delta$  bude  $\tan(\alpha)$

$$\tan(\alpha) = \frac{(x_0 + \Delta)^2 - x_0^2}{\Delta}.$$

Ak sa bude  $\Delta$  blížiť k nule, budem v čitateľi aj menovateľ dostávať menšie a menšie čísla a o výsledku neviem povedať nič. Napíšem si to isté inak:

$$\tan(\alpha) = \frac{(x_0 + \Delta)^2 - x_0^2}{\Delta} = \frac{x_0^2 + 2x_0\Delta + \Delta^2 - x_0^2}{\Delta} = 2x_0 + \Delta.$$

Teraz je jasné, že čím bližšie bude  $\Delta$  k nule, tým bližšie bude  $\tan(\alpha)$  k  $2x_0$ . Môžem teda celkom pokojne povedať, že dotyčnica k  $x^2$  v bode  $x_0$  má sklon  $2x_0$ . Sklon dotyčnice v nejakom bode sa volá *derivácia*, takže sme práve zrátali, že derivácia funkcie  $x^2$  v bode  $x_0$  je  $2x_0$ . Na deriváciu sa môžem pozrieť ako na funkciu, ktorá pre dané  $x$  vráti deriváciu v bode  $x$ : ak mám pôvodnú funkciu  $f(x) = x^2$ , tak jej derivácia je funkcia  $2x$ . Označovať sa zvykne rôzne. Lagrange deriváciu označoval čiarkou, napr. ak  $f(x) = x^2$ , tak  $f'(x) = 2x$ . Keď napr. chceme zdôrazniť, podľa akej premennej derivujeme, môžeme písť aj  $\frac{dx^2}{dx} = 2x$ . V každom prípade, derivácia nám hovorí, ako rýchlo v danom bode  $x_0$  funkcia rastie: ak sa pohneme o malý kúsok  $\Delta$ , funkcia narastie<sup>8</sup> zhruba o  $\Delta \cdot f'(x_0)$ .

Podme si zopár derivácií zrátať. Vieme spraviť  $x^2$ , tak skúsme  $f(x) = x^a$  pre nejaké celé číslo  $a > 2$ . Roznásobme si zátvorky v  $(x_0 + \Delta)^a$ . Napr. pre  $a = 3$  to bude vyzeráť

$$\begin{aligned} (x_0 + \Delta)^3 &= (x_0 + \Delta)(x_0 + \Delta)(x_0 + \Delta) = \\ &= x_0(x_0 + \Delta)(x_0 + \Delta) + \Delta(x_0 + \Delta)(x_0 + \Delta) = \\ &= x_0x_0(x_0 + \Delta) + x_0\Delta(x_0 + \Delta) + \Delta x_0(x_0 + \Delta) + \Delta\Delta(x_0 + \Delta) = \\ &= x_0x_0x_0 + x_0x_0\Delta + x_0\Delta x_0 + x_0\Delta\Delta + \Delta x_0x_0 + \Delta x_0\Delta + \Delta\Delta x_0 + \Delta\Delta\Delta \end{aligned}$$

Z tohto by malo byť jasné, že ak roznásobíme  $(x_0 + \Delta)^a$ , dostanem súčet  $2^a$  sčítancov tak, že z každej zátvorky si vyberiem buď  $x_0$  alebo  $\Delta$ . Takže budem mať jeden sčítanec  $x_0^a$ , a sčítancov  $\Delta \cdot x_0^{a-1}$  a všetky ostatné budú

<sup>8</sup>Ak je derivácia kladná, funkcia rastie, ak je záporná, tak funkcia klesá. Ak je nulová, tak nerastie ani neklesá.

obsahoval člen  $\Delta^2$ . Preto  $(x_0 + \Delta)^a = x_0^a + a\Delta x_0^{a-1} + \Delta^2\heartsuit$ , kde  $\heartsuit$  je niečo zložené z  $x_0$  a  $\Delta$ , čo ma teraz príliš nezaujíma. Idem rátať sklon priamok podobne ako pred chvíľou:

$$\tan(\alpha) = \frac{(x_0 + \Delta)^a - x_0^a}{\Delta} = \frac{x_0^a + a\Delta x_0^{a-1} + \Delta^2\heartsuit - x_0^a}{\Delta} = \frac{a\Delta x_0^{a-1} + \Delta^2\heartsuit}{\Delta} = ax_0^{a-1} + \Delta\heartsuit$$

Keď beriem menšie a menšie  $\Delta$ ,  $\Delta\heartsuit$  bude viac a viac bližšie k nule. Preto  $(x^a)' = ax^{a-1}$ . Platí to aj pre iné hodnoty  $a$ ? Ak  $a = 0$ ,  $x^0 = 1$  a uhol je zjavne 0. Fajn. Čo tak  $a = -1$ ? Vtedy mám<sup>9</sup>  $x^{-1} = \frac{1}{x}$ , takže si zrátam:

$$\tan(\alpha) = \frac{\frac{1}{x_0+\Delta} - \frac{1}{x_0}}{\Delta} = \frac{\frac{x_0 - (x_0 + \Delta)}{x_0(x_0 + \Delta)}}{\Delta} = \frac{x_0 - (x_0 + \Delta)}{x_0(x_0 + \Delta)\Delta} = -\frac{1}{x_0(x_0 + \Delta)}$$

takže keď je  $\Delta$  veľmi blízko nuly, je  $\tan(\alpha)$  blízko  $-\frac{1}{x_0^2}$ , takže  $(x^{-1})' = -\frac{1}{x^2} = -1 \cdot x^{-2}$ . Skvelé. A čo ak  $a = \frac{1}{2}$ ? Vtedy  $x^{\frac{1}{2}} = \sqrt{x}$ , takže mám

$$\tan(\alpha) = \frac{\sqrt{x_0 + \Delta} - \sqrt{x_0}}{\Delta} = \frac{\sqrt{x_0 + \Delta} - \sqrt{x_0}}{\Delta} \cdot \frac{\sqrt{x_0 + \Delta} + \sqrt{x_0}}{\sqrt{x_0 + \Delta} + \sqrt{x_0}} = \frac{x_0 + \Delta - x_0}{\Delta (\sqrt{x_0 + \Delta} + \sqrt{x_0})} = \frac{1}{\sqrt{x_0 + \Delta} + \sqrt{x_0}}$$

a teda keď je  $\Delta$  menšie a menšie dostávam  $(\sqrt{x})' = \frac{1}{2\sqrt{x}} = \frac{1}{2}x^{-\frac{1}{2}}$ . S trochou námahy by sa dalo vidieť, že vzťah  $(x^a)' = ax^{a-1}$  platí pre hocjaké  $a$ . To sa nám ešte bude hodíť.

Ešte sa chvíľu s deriváciami hrajme: ak mám funkcie  $f(x)$  a  $g(x)$  a spravím si funkciu  $h(x) = f(x) + g(x)$ . Viem zistiť  $h'(x)$ ? Napíšem si ako vždy:

$$\tan(\alpha) = \frac{(f(x_0 + \Delta) + g(x_0 + \Delta)) - (f(x_0) + g(x_0))}{\Delta} = \frac{f(x_0 + \Delta) - f(x_0)}{\Delta} + \frac{g(x_0 + \Delta) - g(x_0)}{\Delta}$$

Preto vidíme, že  $h'(x) = f'(x) + g'(x)$ . Podobne sa dá vidieť, že ak mám funkciu  $f(x)$  a spravím si  $g(x) = c \cdot f(x)$ , tak  $g'(x) = c \cdot f'(x)$ . S násobením je to trošku zložitejšie. Ak mám funkcie  $f(x)$  a  $g(x)$  a urobím si  $h(x) = f(x) \cdot g(x)$ , tak dostávam

$$\tan(\alpha) = \frac{(f(x_0 + \Delta)g(x_0 + \Delta)) - (f(x_0)g(x_0))}{\Delta}$$

Čo s tým? Spravím fintu, že do čitateľa si pridám rafinované napísanú nulu

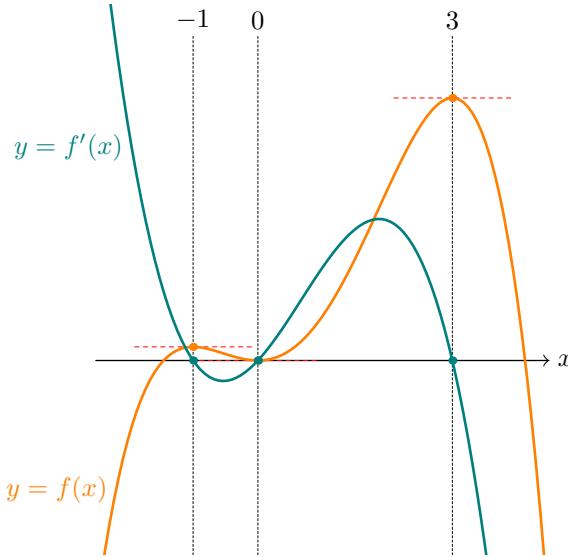
$$\begin{aligned} \tan(\alpha) &= \frac{f(x_0 + \Delta)g(x_0 + \Delta) - f(x_0)g(x_0) + f(x_0 + \Delta)g(x_0) - f(x_0 + \Delta)g(x_0)}{\Delta} = \\ &= f(x_0 + \Delta) \left( \frac{g(x_0 + \Delta) - g(x_0)}{\Delta} \right) + g(x_0) \left( \frac{f(x_0 + \Delta) - f(x_0)}{\Delta} \right) \end{aligned}$$

Keď je  $\Delta$  strašne maličké, tak  $f(x_0 + \Delta)$  je čoraz bližšie k  $f(x_0)$  a tie veľké zátvorky sa blížia k dotyčniciam pre  $g(x_0)$  a  $f(x_0)$ . Preto vidíme, že  $h'(x) = f(x)g'(x) + f'(x)g(x)$ .

Na lepsiu ilustráciu si zoberme funkciu  $f(x) = 8x^3 - 3x^4 + 18x^2$  (na obrázku oranžová) a pýtajme sa, akú najväčšiu hodnotu môže nadobudnúť. Vieme si ľahko zrátať jej deriváciu: najprv ju rozdelíme na súčet troch

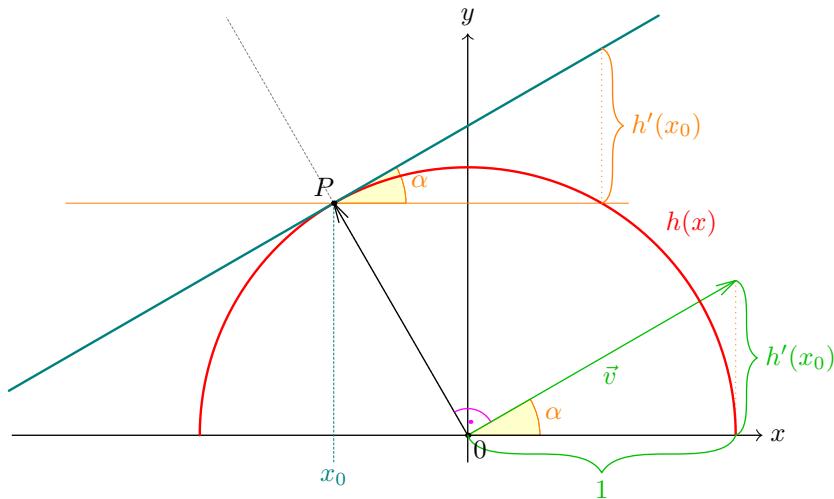
<sup>9</sup>pripomeň si poznámku pod čiarou na str. 90

členov, z ktorých každý má tvar  $cx^a$  a tie už zderivovať vieme. Takže dostaneme, že  $f'(x) = 24x^2 - 12x^3 + 36x$  (na obrázku modrá funkcia).



Ak má mať  $f(x)$  v nejakom bode  $x_0$  maximum, tak v  $x_0$  nesmie rásť, a teda  $f'(x_0) = 0$ . Takže stačí zistiť, kde všade je  $f'(x) = 0$  a v jednom z tých bodov bude maximum. Napíšem si  $f'(x)$  v tvare  $f'(x) = -12x(x^2 - 2x - 3) = -4x(x+1)(x-3)$ , z ktorého vidno, že  $f'(x) = 0$  v bodoch  $-1, 0$  a  $3$ . A je to.

Ešte jedna vec sa bude hodif pre počítanie s deriváciami, a to derivácia zloženej funkcie. Zoberme si kružnicu so stredom v bode  $[0, 0]$  a polomerom 1 a podme k nej spraviť dotyčnicu v nejakom bode  $P$ . Narysoval by som to ľahko: urobím polpriamku  $\overrightarrow{OP}$  a na ňu kolmicu v bode  $P$ . Ako to ale zrátať? Kružnicu tvoria body so vzdialenosou 1 od stredu, preto pre ne platí  $x^2 + y^2 = 1^2$ . Keď si z toho vyjadrim  $y$ , dostanem, že kružnica (presnejšie jej horná polovica) je grafom funkcie  $h(x) = \sqrt{1 - x^2}$ . Vektor  $\vec{v} = [1, h'(x_0)]$  mi preto dáva dotyčnicu, ktorú chceme. Keď zrátam  $h'(x_0)$ , budem môcť overiť, že vektor  $\vec{v}$  je kolmý na  $\overrightarrow{OP}$ .



Ako zrátam deriváciu  $\sqrt{1 - x^2}$ ? Tá funkcia síce vyzerá hrozivo, ale keď sa lepšie prizriem, vidíme, že je zložená z dvoch funkcií, ktoré už derivovať viem. Keď označím  $g(x) = 1 - x^2$  a  $f(x) = \sqrt{x}$ , tak  $\sqrt{1 - x^2} = f(g(x))$  (t.j. najprv sa zráta  $g(x)$  a výsledok sa použije ako vstup do funkcie  $f(\cdot)$ ).

Podme sa teda pozrieť zblížšia na deriváciu  $f(g(x))$ .

$$\tan(\alpha) = \frac{f(g(x_0 + \Delta)) - f(g(x_0))}{\Delta} \cdot \frac{g(x_0 + \Delta) - g(x_0)}{g(x_0 + \Delta) - g(x_0)} = \frac{f(g(x_0 + \Delta)) - f(g(x_0))}{g(x_0 + \Delta) - g(x_0)} \cdot \frac{g(x_0 + \Delta) - g(x_0)}{\Delta}$$

V prvom kroku som si napísal rafinovanú jednotku. To môžem urobiť, len ak  $g(x_0 + \Delta) \neq g(x_0)$ ; v opačnom prípade by bolo treba trochu inú úvahu (ale dopadlo by to rovnako). V tom, čo som dostał, je  $\frac{g(x_0 + \Delta) - g(x_0)}{\Delta}$  pre veľmi malé  $\Delta$  v podstate  $g'(x_0)$ . Čo sa dá pre malé  $\Delta$  povedať o

$$\frac{f(g(x_0 + \Delta)) - f(g(x_0))}{g(x_0 + \Delta) - g(x_0)}?$$

Hodnota  $g(x_0 + \Delta)$  je približne  $g(x_0) + \Delta g'(x_0)$ . Keďže  $g'(x_0)$  je nejaké konkrétné číslo, keď zmenšujem  $\Delta$  na veľmi malé, je aj  $\Delta \cdot g'(x_0)$  veľmi malé. Keď si označím  $y_0 = g(x_0)$  a  $\Delta' = \Delta g'(x_0)$ , budem mať

$$(f(g(x_0)))' = \frac{f(\Delta' + y_0) - f(y_0)}{\Delta'} \cdot g'(x_0)$$

a teda vidíme, že  $(f(g(x)))' = f'(g(x))g'(x)$ . Tento vzťah je možno zrozumiteľnejší, keď ho zapíšem takto:

$$\frac{dh}{dx} = \frac{dh}{dg} \cdot \frac{dg}{dx}$$

To znamená, že keď mám nejakú funkciu  $h$  (napr.  $h = \sqrt{1 - x^2}$ ), označím si nejakú časť ako  $g$  (napr.  $g(x) = 1 - x^2$ ). Teraz sa na  $g$  môžem pozrieť ako na novú premennú, takže  $h$  závisí od  $g$  a  $g$  závisí od  $x$ . Preto keď trochu zmením  $x$ ,  $g$  sa zmení o  $\frac{dg}{dx}$  (a.k.a.  $g'(x)$ ) a zaujíma ma, ako sa zmení  $h(g)$ .

V našom príklade  $g(x) = 1 - x^2$ , preto  $\frac{dg}{dx} = g'(x) = -2x$ . Vonkajšia funkcia  $f(g(x)) = h(g) = \sqrt{g}$ , preto  $\frac{dh}{dg} = \frac{1}{2}g^{-\frac{1}{2}}$ . Preto

$$\frac{d\sqrt{1-x^2}}{dx} = \frac{d\sqrt{g}}{dg} \cdot \frac{d(1-x^2)}{dx} = \frac{1}{2}g^{-\frac{1}{2}} \cdot (-2x) = \frac{1}{2(1-x)^{\frac{1}{2}}} \cdot (-2x) = \frac{-x}{\sqrt{1-x^2}}$$

Teraz si môžem overiť, že vektor  $\vec{v} = \left[1, -\frac{x_0}{\sqrt{1-x_0^2}}\right]$  je kolmý na  $\overrightarrow{OP} = [x_0, \sqrt{1-x_0^2}]$ . Na to sa hodí, čo sme si hovorili o skalárnom súčine na str. 167: vyrátam si

$$\vec{u} \cdot \overrightarrow{OP} = x_0 - \frac{x_0}{\sqrt{1-x_0^2}} \cdot \sqrt{1-x_0^2} = x_0 - x_0 = 0$$

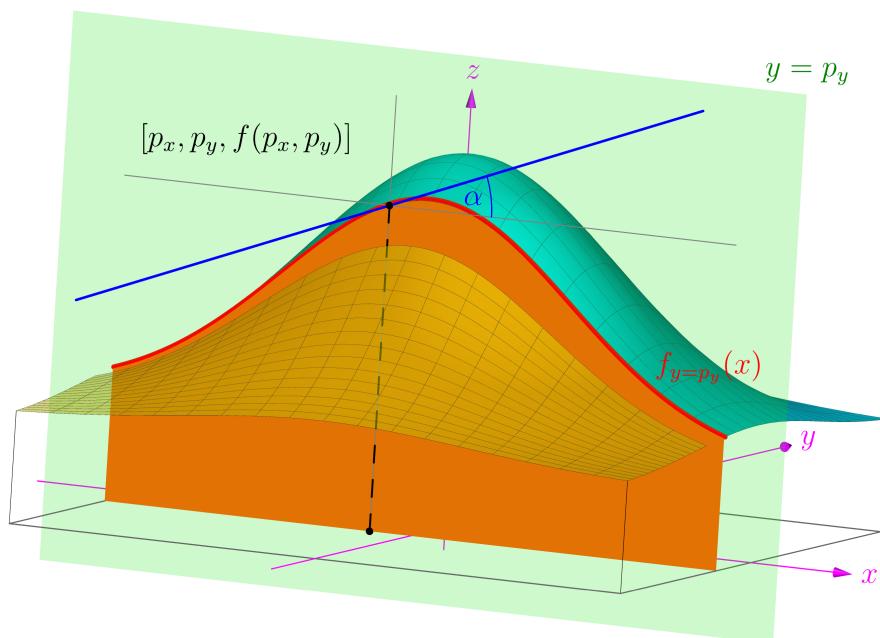
preto vidíme, že keď skalárny súčin je 0, kosínus uhla medzi vektormi musí byť 0, a teda  $\overrightarrow{OP}$  a  $\vec{v}$  sú na seba kolmé.

Zhrňme si, čo vieme o deriváciách:

- 1.  $(x^a)' = ax^{a-1}$
- 2.  $(c \cdot f(x))' = c \cdot f'(x)$
- 3.  $(f(x) + g(x))' = f(x)' + g'(x)$
- 4.  $(f(x) \cdot g(x))' = f'(x)g(x) + f(x)g'(x)$
- 5.  $(f(g(x)))' = f'(g(x)) \cdot g'(x)$

Aby som sa príliš nerozkokošil s deriváciami, treba si pripomenúť, načo to celé rozprávam: hľadáme spôsob, ako upraviť parametre siete tak, aby chyba čo najviac klesla. Zoberme si opäť najjednoduchšiu možnú sieť s jediným neurónom, ktorý má parametre  $w$ ,  $b$  a pre vstup  $x$  vyráta  $\tanh(wx + b)$ . Dajme tomu, že chceme riešiť úlohu s jediným vstupom  $x_1$ , ktorého správny výstup je  $y_1$ . Odpoveď siete na vstupe  $x_1$  je  $\tanh(wx_1 + b)$ , chyba siete (stredná kvadratická odchýlka) je funkcia parametrov  $w$  a  $b$ :  $\text{err}(w, b) = (\tanh(wx_1 + b) - y_1)^2$ . Ak mám "siet" s parametrami  $w$ ,  $b$ , chceme zistiť, ktorým smerom sa pohnúť, aby sa chyba zmenšila. Na to nám poslúži gradient funkcie<sup>10</sup>.

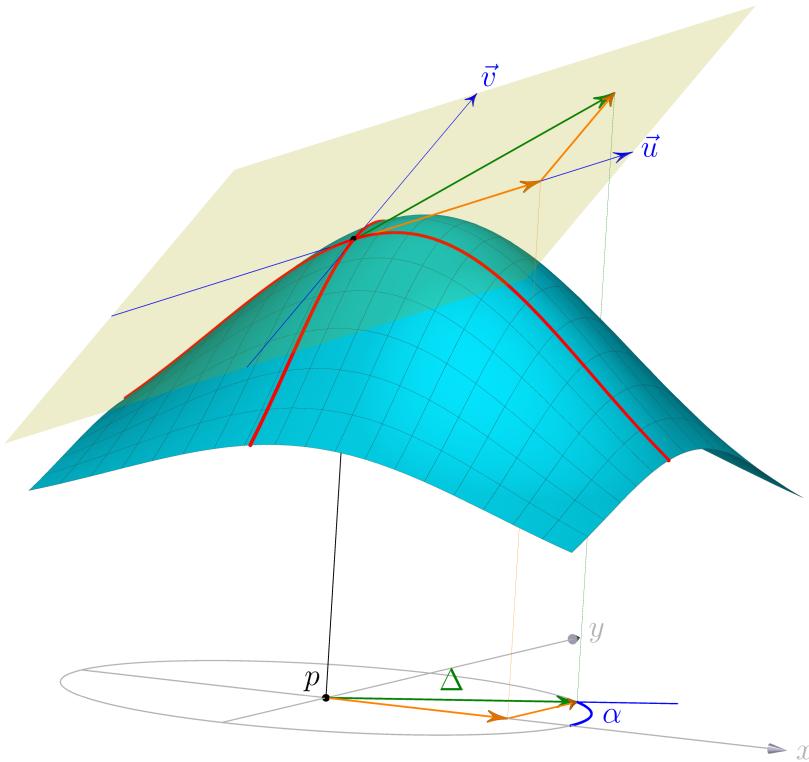
Mám nejakú funkciu dvoch premenných  $f(x, y)$  a bod  $p = [p_x, p_y]$ . Ked' si zafixujem hodnotu  $y = p_y$  a dosadím do  $f(x, y)$ , dostanem funkciu jednej premennej  $f_{y=p_y}(x)$ . Body, v ktorých je  $y = p_y$  tvoria v 3D rovinu. Funkcia  $f_{y=p_y}(x)$  je prienik 3D výskovej mapy s touto rovinou:



Derivácia funkcie  $f_{y=p_y}(x)$  v bode  $p_x$  mi preto určuje, ako rastie výšková mapa v bode  $p$  v smere osi  $x$ . Toto je často používané číslo, preto dostalo aj vlastný názov. Volá sa *parciálna derivácia* a zvykne sa označovať  $\frac{\partial f}{\partial x}(p)$ . To znamená, že keby som sa z bodu  $p$  pohol o veľmi malé  $\Delta$  v smere osi  $x$ , tak hodnota  $f(x, y)$  za zväčší o  $\Delta \cdot \frac{\partial f}{\partial x}(p)$ .

Ked' to podobne spravíme pre os  $y$ , dostanem dva vektoru  $\vec{u} = \left[1, 0, \frac{\partial f}{\partial x}(p)\right]$  a  $\vec{v} = \left[0, 1, \frac{\partial f}{\partial y}(p)\right]$ , ktoré mi určujú dotykovú rovinu ku grafu funkcie  $f$  v bode  $[p_x, p_y, f(p_x, p_y)]$ .

<sup>10</sup>Už sme ho spomenuli, v časti o erózii na str. 175, ale vtedy nám stačila iba zjednodušená predstava. Teraz to budeme potrebovať spraviť poriadnejšie.



Povedzme, že sa teraz pohnem z bodu  $p$  o malý kúsok  $\Delta$  pod uhlom  $\alpha$ . Zaujíma ma, ako sa zmení hodnota funkcie, ktorú si, ak je  $\Delta$  dosť malé, môžem nahradieť dotykovou rovinou. Najprv sa pohnem v smere osi  $x$  o  $\Delta \cos(\alpha)$ . Kedže v smere osi  $x$  dotyková rovina rastie podľa vektora  $\vec{u}$ , hodnota narastie o  $\Delta \cos(\alpha) \frac{\partial f}{\partial x}(p_x, p_y)$ . Potom sa pohnem v smere osi  $y$  o  $\Delta \sin(\alpha)$  a analogicky mi hodnota narastie o  $\Delta \sin(\alpha) \frac{\partial f}{\partial y}(p_x, p_y)$ . Kedže to zrátam, dostonem, že keď som sa pohol o  $\Delta$  pod uhlom  $\alpha$ , hodnota narastla o  $\Delta \left( \cos(\alpha) \frac{\partial f}{\partial x}(p_x, p_y) + \sin(\alpha) \frac{\partial f}{\partial y}(p_x, p_y) \right)$ . Ak si označím  $\vec{w} = [\cos(\alpha), \sin(\alpha)]$  vektor smeru o ktorý som sa pohol a  $\vec{\nabla} = \left[ \frac{\partial f}{\partial x}(p_x, p_y), \frac{\partial f}{\partial y}(p_x, p_y) \right]$  vektor parciálnych derivácií, vidím, že hodnota narastla o  $\Delta (\vec{w} \cdot \vec{\nabla})$ . A keď si pripomienim, čo sme rozprávali o skalárnom súčine na str. 167, vidím, že hodnota narastie o  $\Delta \cdot \|\vec{w}\| \cdot \|\vec{\nabla}\| \cdot \cos(\varphi)$ , kde  $\varphi$  je uhol medzi vektormi  $\vec{w}$  a  $\vec{\nabla}$ .

Akým smerom sa má pohnúť, ak chcem, aby hodnota narastla čo najviac? Hodnoty  $\Delta$ ,  $\|\vec{w}\|$ ,  $\|\vec{\nabla}\|$  sú stále rovnaké, jediné, čo sa mení, je  $\cos(\varphi)$ . To znamená, že hodnota narastie najviac vtedy, keď  $\cos(\varphi)$  bude 1, t.j. keď  $\varphi$  bude 0. Inými slovami, hodnota narastie najviac vtedy, keď sa pohnem v smere vektora  $\vec{\nabla}$ .

Táto úvaha by sa dala zovšeobecniť aj na veľa rozmerov<sup>11</sup>, takže by sme dostali:

Funkcia  $n$  premenných  $f(x_1, \dots, x_n)$  rastie najrýchlejšie v smere vektora gradientu  $\nabla = \left[ \frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right]$

Na záver matematickej odbočky si ešte raz všimni predchádzajúci obrázok. Máme tam funkciu  $f(x, y)$  a jej dotykovú plochu v bode  $p$ . Čo by sa stalo, ak by  $x$  a  $y$  neboli premenné, ale nejaké funkcie parametra  $t$ , takže by nás zaujímal funkcia jednej premennej  $h(t) = f(x(t), y(t))$ ? Ako by vyzerala derivácia  $\frac{dh}{dt}$ ? Ak parametrom

<sup>11</sup>To nie je úplne samozrejmé. Je veľa vecí, ktoré napr. platia v dvoch rozmeroch, ale v troch už nie (napr. že dve priamky, ktoré nie sú rovnobežné, sa pretínajú), takže by bolo treba spraviť poriadny dôkaz pre veľa rozmerov. Tu ho ale neurobíme; išlo len o to ukázať intuiciu v pozadí.

$t$  pohnem o veľmi malé  $\Delta$ , zmení sa hodnota  $x(t)$  na  $x(t + \Delta)$ . Keďže  $\Delta$  je maličké, môžem si funkciu  $x(t)$  nahradieť jej dotyčnicou a dostanem  $x(t + \Delta) \approx x(t) + \Delta \frac{dx}{dt}(t)$ . Hodnota  $y(t)$  sa podobne zmení na  $y(t + \Delta) \approx y(t) + \Delta \frac{dy}{dt}(t)$ . To znamená, že v dotykovej rovine k funkcií  $f(x, y)$  sa pohnem o  $\Delta \frac{dx}{dt}(t)$  v smere osi  $x$  a hodnota funkcie narastie o  $\frac{\partial f}{\partial x}(x(t), y(t)) \Delta \frac{dx}{dt}(t)$ . Potom sa pohnem o  $\Delta \frac{dy}{dt}(t)$  v smere osi  $y$  a hodnota funkcie narastie o ďalších  $\frac{\partial f}{\partial y}(x(t), y(t)) \Delta \frac{dy}{dt}(t)$ . Dokopy teda ak pohnem parametrom  $t$  o  $\Delta$ , hodnota  $h(t)$  narastie o  $\Delta \left( \frac{\partial f}{\partial x}(x(t), y(t)) \frac{dx}{dt}(t) + \frac{\partial f}{\partial y}(x(t), y(t)) \frac{dy}{dt}(t) \right)$ . Zasa by sa to dalo rozšíriť aj na viac rozmerov a dostali by sme analógiu derivácie zloženej funkcie pre viaceru rozmerov:

Ak máme funkciu  $f(x_1, \dots, x_n)$  a  $h(t) = f(g_1(t), g_2(t), \dots, g_n(t))$ , potom

$$\frac{dh}{dt}(t) = \sum_{i=1}^n \frac{\partial f}{\partial x_i}(g_1(t), \dots, g_n(t)) \cdot \frac{dg_i}{dt}(t)$$

Koniec matematickej odbočky, vráťme sa k našej jednoneurónovej "sieti". Povedali sme, že pre parametre  $[w, b]$  je chyba siete

$$\text{err}(w, b) = (\tanh(wx_1 + b) - y_1)^2.$$

Akým smerom treba pohnúť parametre, aby sa chyba čo najviac zmenšila? Teraz už vieme, že treba ísť v smere  $-\nabla$ , kde  $\nabla$  je gradient  $\left[ \frac{\partial \text{err}}{\partial w}, \frac{\partial \text{err}}{\partial b} \right]$ . Podľme teda rátať deriváciu. Funkciu  $\tanh(\cdot)$  príliš nepoznáme, ale dá sa vyhľadať, že jej deriváciu už niekto zratal a

$$\tanh(x)' = \frac{1}{\cosh(x)^2}$$

$\cosh(\cdot)$  je nejaká iná funkcia, ktorá je dostupná z knižnice `cmath` a to nám nateraz môže stačiť. Funkciu err si napíšeme ako zloženú funkciu takto

$$\begin{aligned} \text{err}(w, b) &= (y(w, b) - y_1)^2 \\ y(w, b) &= \tanh(u(w, b)) \\ u(w, b) &= wx_1 + b \end{aligned}$$

Použijeme, čo vieme o derivovaní zložených funkcií a napíšme si

$$\frac{\partial \text{err}}{\partial w} = \frac{\partial \text{err}}{\partial y} \cdot \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial w} \quad \frac{\partial \text{err}}{\partial b} = \frac{\partial \text{err}}{\partial y} \cdot \frac{\partial y}{\partial u} \cdot \frac{\partial u}{\partial b}$$

S tým, čo už vieme, zderivujeme jednotlivé funkcie ľahko:

$$\frac{\partial \text{err}}{\partial y} = 2(y - y_1) \quad \frac{\partial y}{\partial u} = \frac{1}{\cosh(u)^2} \quad \frac{\partial u}{\partial w} = x_1 \quad \frac{\partial u}{\partial b} = 1$$

Keď to poskladáme dokopy, dostaneme

$$\frac{\partial \text{err}}{\partial w}(w, b) = \frac{2x_1(\tanh(wx_1 + b) - y_1)}{\cosh(wx_1 + b)^2} \quad \frac{\partial \text{err}}{\partial b}(w, b) = \frac{2(\tanh(wx_1 + b) - y_1)}{\cosh(wx_1 + b)^2}$$

To znamená, že ak máme jednoneurónovú "sieť", tak pre akúkoľvek kombináciu  $w, b$  vieme vyrátať, ktorým smerom treba hodnoty  $w$  a  $b$  pohnúť, aby sa chyba siete čo najrýchlejšie zmenšovala.

Sieť s jedným neurónom nie je nič, čo by nás príliš uspokojilo, tak to podme skúsiť zrátať pre zaujímavejšie siete. V našom programe máme triedu **Network**, ktorá obsahuje pole vrstiev typu **Layer**. Označme si vrstvy siete  $L^{(0)}, L^{(1)}, \dots, L^{(h-1)}$ . Každá vrstva obsahuje **Matrix**  $W, b$ , tie vo vrstve  $t$  si označme  $W^{(t)}, b^{(t)}$ . Veľkosti vrstiev máme zapamätané vo **vector<int>**  $n$ , takže  $n_0$  je počet hodnôt na vstupe a  $t$ -ta vrstva  $L^{(t)}$  má  $n_{t+1}$  neurónov, takže  $W^{(t)}$  je rozmerov  $n_t \times n_{t+1}$  a  $b^{(t)}$  je stĺpec  $n_t \times 1$ . Tak, ako v predchádzajúcom hračkárskom príklade boli parametre siete  $w$  a  $b$ , teraz sú parametre všetky hodnoty vo všetkých maticiach  $W^{(t)}, b^{(t)}$ . Podobne ako predtým, aj teraz potrebujeme zrátať chybu siete a potom gradient, t.j. deriváciu podľa každého z parametrov. Priponiem, že vyhodnocovanie vrstvy máme naprogramované takto:

```

1 Matrix Layer::eval(const Matrix &X) {
2     Matrix res = W * X;
3     res.addColumn(b);
4     res.apply([](double x) { return tanh(x); });
5     return res;
6 }
```

Povedzme, že naša úloha má  $s$  vstupov, takže vstupná matica  $\mathbf{X}$  má rozmery  $n_0 \times s$ . Prvá vrstva najprv vyráta  $\mathbf{W}^{(0)} \cdot \mathbf{X}$  a potom ku každému stĺpcu výsledku priráta  $\mathbf{b}^{(0)}$ . Tento medzivýsledok si označíme  $\mathbf{U}^{(0)}$ . Ak si označíme  $\mathbf{1}^s$  riadok obsahujúci s jednotiek, ľahko sa overí, že  $\mathbf{U}^{(0)} = \mathbf{W}^{(0)} \cdot \mathbf{X} + \mathbf{b}^{(0)} \cdot \mathbf{1}^s$ . Nakoniec na každý prvok  $\mathbf{U}^{(0)}$  aplikujem  $\tanh(\cdot)$ . Túto operáciu budem označovať ako  $\sigma(\mathbf{U}^{(0)})$  a výsledok označíme  $\mathbf{Y}^{(0)}$ . Ak vstupom pre prvú vrstvu je vstupná matica  $\mathbf{X} = \mathbf{X}^{(0)}$ , výstup z prvej vrstvy je vstupom pre druhú, t.j.  $\mathbf{X}^{(1)} = \mathbf{Y}^{(0)}$ . Pri neurónových sieťach sa spravidla rozhodovacia funkcia neaplikuje na poslednú vrstvu, takže mám

$$\begin{aligned}\mathbf{X}^{(0)} &= \mathbf{X} \\ \mathbf{U}^{(t)} &= \mathbf{W}^{(t)} \cdot \mathbf{X}^{(t)} + \mathbf{b}^{(t)} \cdot \mathbf{1}^s \\ \mathbf{Y}^{(t)} &= \sigma(\mathbf{U}^{(t)}) \\ \mathbf{X}^{(t+1)} &= \mathbf{Y}^{(t)}\end{aligned}$$

a dokopy sief ráta zloženú funkciu

$$\mathbf{U}^{(h-1)} = \mathbf{W}^{(h-1)} \cdot \sigma \left( \mathbf{W}^{(h-2)} \cdot \sigma \left( \dots \sigma \left( \mathbf{W}^{(0)} \cdot \mathbf{X} + \mathbf{b}^{(0)} \cdot \mathbf{1}^s \right) \dots \right) + \mathbf{b}^{(h-2)} \cdot \mathbf{1}^s \right) + \mathbf{b}^{(h-1)} \cdot \mathbf{1}^s$$

Ak si označíme  $\widehat{\mathbf{Y}}$  maticu (rozmerov  $h_{n-1} \times s$ ) správnych odpovedí, tak chyba siete, t.j. stredná kvadratická odchýlka, čiže priemer druhých mocnín rozdielov výstupných hodnôt a skutočných hodnôt, je

$$\text{err} \left( \mathbf{W}^{(0)}, \dots, \mathbf{W}^{(h-1)}, \mathbf{b}^{(0)}, \dots, \mathbf{b}^{(h-1)} \right) = \frac{1}{s \cdot n_{h-1}} \sum_{j=0}^{n_{h-1}} \sum_{k=0}^{s-1} \left( \widehat{\mathbf{Y}}_{j,k} - \mathbf{U}_{j,k}^{(h-1)} \right)^2$$

$\text{err}(\cdot)$  je veľmi zložitá funkcia, v ktorej ako neznáme vystupujú všetky hodnoty z matíc  $\mathbf{W}^{(t)}$  a  $\mathbf{b}^{(t)}$ . Aby sme zistili, ktorým smerom chyba najviac klesá, potrebujeme zrátať derivácie  $\frac{\partial \text{err}}{\partial \mathbf{W}_{j,k}^{(t)}}$  a  $\frac{\partial \text{err}}{\partial \mathbf{b}_j^{(t)}}$  pre všetky prípustné kombinácie  $j, k, t$ . Začneme s tým, že si pre všetky prípustné hodnoty  $j, k, t$  vyrátame  $\frac{\partial \text{err}}{\partial \mathbf{U}_{j,k}^{(t)}}$ . Pôjdeme pritom odzadu<sup>12</sup>. Pre fixné  $j, k$  zrátame hodnotu  $\frac{\partial \text{err}}{\partial \mathbf{U}_{j,k}^{(h-1)}}$  tak, že si predstavíme err ako funkciu jedinej premennej  $\mathbf{U}_{j,k}^{(h-1)}$  a zrátame deriváciu. Keď si rozpíšem zápis so sumami, dostanem

<sup>12</sup>preto sa táto metóda zvykne volať *back propagation*

$$\text{err} = \frac{1}{s \cdot n_{h-1}} \left( \widehat{\mathbf{Y}}_{0,0} - \mathbf{U}_{0,0}^{(h-1)} \right)^2 + \dots + \frac{1}{s \cdot n_{h-1}} \left( \widehat{\mathbf{Y}}_{j,k} - \mathbf{U}_{j,k}^{(h-1)} \right)^2 + \dots + \frac{1}{s \cdot n_{h-1}} \left( \widehat{\mathbf{Y}}_{n_{h-1},s-1} - \mathbf{U}_{n_{h-1},s-1}^{(h-1)} \right)^2$$

Už vieme, že derivácia konštantnej funkcie je 0 a  $(f(x) + g(x))' = f'(x) + g'(x)$ . V predchádzajúcom výraze je jediný člen, ktorý závisí od  $\mathbf{U}_{j,k}^{(h-1)}$ , preto všetko ostatné sa zderivuje na nulu a ostane mi

$$\frac{\partial \text{err}}{\partial \mathbf{U}_{j,k}^{(h-1)}} = \frac{-2}{s \cdot n_{h-1}} \left( \widehat{\mathbf{Y}}_{j,k} - \mathbf{U}_{j,k}^{(h-1)} \right)$$

Predstavme si teraz, že už máme pre nejaké  $t$  zrátané  $\frac{\partial \text{err}}{\partial \mathbf{U}_{j,k}^{(t+1)}}$  pre všetky  $j, k$  a chceme z nich vyrátať  $\frac{\partial \text{err}}{\partial \mathbf{U}_{j,k}^{(t)}}$ .

Pretože  $\mathbf{Y}_{j,k}^{(t)} = \sigma(\mathbf{U}_{j,k}^{(t)})$  vždy aplikuje  $\tanh(\cdot)$  na jednotlivé prvky v matici  $\mathbf{U}$ , predstavíme si to ako zloženú funkciu a môžem si vyjadriť

$$\frac{\partial \text{err}}{\partial \mathbf{U}_{j,k}^{(t)}} = \frac{\partial \text{err}}{\partial \mathbf{Y}_{j,k}^{(t)}} \cdot \frac{d\mathbf{Y}_{j,k}^{(t)}}{d\mathbf{U}_{j,k}^{(t)}}$$

pričom  $\frac{d\mathbf{Y}_{j,k}^{(t)}}{d\mathbf{U}_{j,k}^{(t)}}$  je derivácia  $\tanh(\cdot)$ , teda

$$\frac{d\mathbf{Y}_{j,k}^{(t)}}{d\mathbf{U}_{j,k}^{(t)}} = \frac{1}{\cosh(\mathbf{U}_{j,k}^{(t)})^2}$$

Teraz chceme zrátať pre premennú  $\mathbf{Y}_{j,k}^{(t)}$  hodnotu  $\frac{\partial \text{err}}{\partial \mathbf{Y}_{j,k}^{(t)}}$ , ak už máme zrátané všetky hodnoty  $\frac{\partial \text{err}}{\partial \mathbf{U}_{\ell,r}^{(t+1)}}$ . Na to použijeme rámček o zložených funkciach zo strany 266: Najprv si predstavíme, že err je funkcia premenných  $\mathbf{U}_{\ell,r}^{(t+1)}$  a potom si poviem, že v skutočnosti  $\mathbf{U}_{\ell,r}^{(t+1)}$  závisia od premenných  $\mathbf{Y}_{j,k}^{(t)}$ , preto

$$\frac{\partial \text{err}}{\partial \mathbf{Y}_{j,k}^{(t)}} = \sum_{\ell < n_{t+1}} \sum_{r < s} \frac{\partial \text{err}}{\partial \mathbf{U}_{\ell,r}^{(t+1)}} \cdot \frac{\partial \mathbf{U}_{\ell,r}^{(t+1)}}{\partial \mathbf{Y}_{j,k}^{(t)}}$$

Treba nám preto zistiť, ako závisí  $\mathbf{U}_{\ell,r}^{(t+1)}$  od  $\mathbf{Y}_{j,k}^{(t)}$ . Vieme, že  $\mathbf{U}^{(t+1)} = \mathbf{W}^{(t+1)} \cdot \mathbf{Y}^{(t)} + \mathbf{b}^{(t+1)} \cdot \mathbf{1}^s$ . Keď si nakreslíme príslušné matice aj s ich rozmermi, bude to vyzeráť takto:

$$n_{t+1} \begin{array}{|c|} \hline s \\ \hline \mathbf{U}^{(t+1)} \\ \hline \end{array} = n_{t+1} \begin{array}{|c|} \hline n_t \\ \hline \mathbf{W}^{(t+1)} \\ \hline \end{array} \cdot \begin{array}{|c|} \hline n_t \\ \hline \mathbf{Y}^{(t)} \\ \hline \end{array} + s \begin{array}{|c|} \hline 1 \\ \hline \mathbf{b} \\ \hline \end{array} \cdot \begin{array}{|c|} \hline s \\ \hline \mathbf{1} \\ \hline \end{array}$$

To znamená, že

$$\mathbf{U}_{\ell,r}^{(t+1)} = \sum_{i=0}^{n_t-1} \mathbf{W}_{\ell,i}^{(t+1)} \mathbf{Y}_{i,r}^{(t)} + \mathbf{b}_{\ell}^{(t+1)}$$

Kedže nás zaujíma iba to, ako  $\mathbf{U}_{\ell,r}^{(t+1)}$  závisí od  $\mathbf{Y}_{j,k}^{(t)}$ , vidíme, že ak  $r \neq k$ , tak  $\mathbf{Y}_{j,k}^{(t)}$  v tomto výraze nevystupuje a pre  $r = k$  si môžeme napísat

$$\mathbf{U}_{\ell,k}^{(t+1)} = \mathbf{W}_{\ell,j}^{(t+1)} \mathbf{Y}_{j,k}^{(t)} + \text{bias}$$

pričom bias od  $\mathbf{Y}_{j,k}^{(t)}$  nezávisí. Preto  $\frac{\partial \mathbf{U}_{\ell,k}^{(t+1)}}{\partial \mathbf{Y}_{j,k}^{(t)}} = \mathbf{W}_{\ell,j}^{(t+1)}$ . Keď sa vrátíme o krok späť, dostaneme

$$\frac{\partial \text{err}}{\partial \mathbf{Y}_{j,k}^{(t)}} = \sum_{\ell < n_{t+1}} \frac{\partial \text{err}}{\partial \mathbf{U}_{\ell,k}^{(t+1)}} \cdot \mathbf{W}_{\ell,j}^{(t+1)}$$

Keď sa pozornejšie prizrieš, čo sme zrátali, pripomína to násobenie matíc. Ak si označíme  $\widehat{\mathbf{U}}^{(t+1)}$  maticu rozmerov  $n_{t+1} \times s$ , ktorá bude obsahovať hodnoty  $\frac{\partial \text{err}}{\partial \mathbf{U}_{\ell,r}^{(t+1)}}$  a  $\widehat{\mathbf{Y}}^{(t)}$  maticu rozmerov  $n_t \times s$ , ktorá bude obsahovať hodnoty  $\frac{\partial \text{err}}{\partial \mathbf{Y}_{j,k}^{(t)}}$ , tak

$$\widehat{\mathbf{Y}}^{(t)} = \mathbf{W}^T \cdot \widehat{\mathbf{U}}^{(t+1)}$$

Uff. Tak toto bolo trochu viac rátania, ale o to jednoduchšie budeme mať teraz programovanie. Triedu `Layer` si prerobíme tak, že si bude okrem matíc  $\mathbf{W}$  a  $\mathbf{b}$  pamätať aj  $\mathbf{U}$ ,  $\mathbf{Y}$  a  $d\mathbf{U}$ , čo budú naše matice  $\mathbf{U}^{(t)}$ ,  $\mathbf{Y}^{(t)}$  a  $\widehat{\mathbf{U}}^{(t)}$ .

```

1 struct Layer {
2     int n1;           // počet neurónov vo vrstve
3     int n0;           // veľkosť predchádzajúcej vrstvy (vstupu)
4     Matrix W, b;    // W: n1 x n0, kde w[i,j] = váha i-teho neurónu k j-temu vstupu
5                 // b: stĺpec n1 x 1
6     Matrix U, Y;   // n1 x s : výstupné hodnoty neurónov pred a po aktivácii
7     Matrix dU;      // derivácia chyby podľa U
8
9     Layer(int _n0, int _n1); // konštruktor
10
11    void feed(const Matrix&);           // spracuj vstup, nastav U
12    void activate();                   // nastav Y podľa už nastaveného U
13    void backPropagation(const Layer& nxt); // nastav dU podľa nasledujúcej vrstvy
14 }
```

Jednotlivé metódy sa napíšu priamočiaro:

```

1 void Layer::feed(const Matrix &x) {
2     if (x.m != s) {
3         s = x.m;
4         U = Matrix(n1, s);
5     }
6     multInto(W, x, U);
7     U.addColumn(b);
8 }
```

```

1 void Layer::activate() {
2     Y = U;
3     Y.apply([](double x) { return tanh(x); });
4 }
```

## Neurónové siete

```

1 void Layer::backPropagation(const Layer &nxt) {
2     Matrix Z = (nxt.W.transposed()) * nxt.dU;
3     dU = U;
4     dU.apply([this](Num x) {
5         Num tmp = cosh(x);
6         return 1.0 / (tmp * tmp);
7     });
8     dU.fill([this, &Z](int i, int j) { return dU(i, j) * Z(i, j); });
9 }
```

Metóda `backPropagation` vyráta `dU` v jednej vrstve na základe už vyrátanej `dU` z nasledujúcej vrstvy. Aby to celé mohlo odštartovať, potrebujeme zrátať `dU` v poslednej vrstve. V nej rátame chybu pomocou `mse`, takže si spravíme takúto funkciu, ktorá vyráta  $\frac{\partial \text{err}}{\partial U_{j,k}^{(h-1)}}$  a výsledok uloží v matici `D`:

```

1 void diffMse(const Matrix &data, const Matrix &truth, Matrix &D) {
2     const int n = truth.n, m = truth.m;
3     if (D.n != n || D.m != m) D = Matrix(n, m); // prerob ak nemá správne rozmery
4     for (int i = 0; i < n; i++) {
5         for (int j = 0; j < m; j++)
6             D(i, j) = -2.0 / (double)(n * m) * (truth(i, j) - data(i, j));
7 }
```

Celá sieť bude vstup spracovávať v metóde `feed`, ktorá výsledok uloží v `layers[h - 1].U`:

```

1 Matrix& Network::feed(const Matrix &input) {
2     const Matrix *x = &input;
3     for (int i = 0; i < h - 1; i++) {
4         layers[i].feed(*x);
5         layers[i].activate();
6         x = &layers[i].Y;
7     }
8     layers[h - 1].feed(*x);
9     return layers[h - 1].U;
10 }
11
12 Matrix& Matrix::output() { return layers[h - 1].U; }
13
14 Num Network::error(const Matrix &output, const Matrix &truth) {
15     return mse(output, truth);
16 }
```

Volanie `Network::backPropagation` najprv nechá vstupnú maticu prejsť vrstvu po vrstve cez celú sieť metodou `feed`, ktorá v každej vrstve nastaví hodnoty `U` a `Y`. Potom v opačnom poradí nastaví v každej vrstve `dU`.

```

1 void Network::backPropagation(const Matrix &input, const Matrix &truth) {
2     feed(input);
3     diffMse(layers[h - 1].U, truth, layers[h - 1].dU);
4     for (int i = h - 2; i >= 0; i--) layers[i].backPropagation(layers[i + 1]);
5 }
```

Stále ešte ale nemáme zrátaný gradient, t.j. hodnoty  $\frac{\partial \text{err}}{\partial W_{j,k}^{(t)}}$  a  $\frac{\partial \text{err}}{\partial b_j^{(t)}}$ . Keď už ale máme zrátané  $\frac{\partial \text{err}}{\partial U_{j,k}^{(t)}}$ , tento zvyšok dorátame ľahko. Vieme, že

$$U^{(t)} = W^{(t)} \cdot Y^{(t-1)} + b^{(t)} \cdot \mathbf{1}^{(s)}$$

Pozrime sa najprv na tú ľažšiu vec, na  $\frac{\partial \text{err}}{\partial \mathbf{W}_{j,k}^{(t)}}$ . Podobne, ako pred chvíľou, predstavíme si err ako zloženú funkciu premenných  $\mathbf{U}_{\ell,r}^{(t)}$ , ktoré v skutočnosti závisia od  $\mathbf{W}_{j,k}^{(t)}$  a dostaneme

$$\frac{\partial \text{err}}{\partial \mathbf{W}_{j,k}^{(t)}} = \sum_{\ell < n_t} \sum_{r < s} \frac{\partial \text{err}}{\partial \mathbf{U}_{\ell,r}^{(t)}} \cdot \frac{\partial \mathbf{U}_{\ell,r}^{(t)}}{\partial \mathbf{W}_{j,k}^{(t)}}.$$

Pretože

$$\mathbf{U}_{\ell,r}^{(t)} = \sum_{i=0}^{n_t-1} \mathbf{W}_{\ell,i}^{(t)} \mathbf{Y}_{i,r}^{(t-1)} + \mathbf{b}_{\ell}^{(t)}$$

tak viem, že  $\mathbf{U}_{\ell,r}^{(t)}$  závisí od môjho  $\mathbf{W}_{j,k}^{(t)}$  iba vtedy, keď  $\ell = j$ , a potom

$$\mathbf{U}_{j,r}^{(t)} = \mathbf{W}_{j,k}^{(t)} \mathbf{Y}_{k,r}^{(t-1)} + \mathbf{b}_j^{(t)}$$

$$\frac{\partial \text{err}}{\partial \mathbf{W}_{j,k}^{(t)}} = \sum_{r < s} \frac{\partial \text{err}}{\partial \mathbf{U}_{j,r}^{(t)}} \cdot \frac{\partial \mathbf{U}_{j,r}^{(t)}}{\partial \mathbf{W}_{j,k}^{(t)}} = \sum_{r < s} \frac{\partial \text{err}}{\partial \mathbf{U}_{j,r}^{(t)}} \cdot \mathbf{Y}_{j,k}^{(t-1)}$$

Opäť, ak si označím  $\widehat{\mathbf{W}}^{(t)}$  maticu, kde budú hodnoty  $\frac{\partial \text{err}}{\partial \mathbf{W}_{j,k}^{(t)}}$ , tak

$$\widehat{\mathbf{W}}^{(t)} = \widehat{\mathbf{U}}^{(t)} \cdot \mathbf{Y}^{(t-1)}$$

S hodnotami  $\frac{\partial \text{err}}{\partial \mathbf{b}_j^{(t)}}$  je to ešte jednoduchšie, lebo mám výraz

$$\frac{\partial \text{err}}{\partial \mathbf{b}_j^{(t)}} = \sum_{\ell < n_t} \sum_{r < s} \frac{\partial \text{err}}{\partial \mathbf{U}_{\ell,r}^{(t)}} \cdot \frac{\partial \mathbf{U}_{\ell,r}^{(t)}}{\partial \mathbf{b}_j^{(t)}}$$

a  $\mathbf{U}_{\ell,r}^{(t)} = \mathbf{b}_{\ell}^{(t)} + \mathbf{W}_{\ell,r}^{(t)}$ .

Takže to môžeme rovno doprogramovať. Urobíme to tak, že potom, čo sa zavolá `Network::backPropagation()` a nastavia sa `dU` vo všetkých vrstvách, zavoláme `Network::gradient()`, ktorý vyráta matice `dW` a `db` pre každú vrstvu. Môžem si to spraviť tak, že každá vrstva vráti matice `dW` a `db` v jednom type `LayerData`. Pridané funkcie budú vyzerať nejak takto:

```

1 struct LayerData {
2     Matrix dW, db;
3 };
4
5 using Gradient = std::vector<LayerData>;

```

```

1 LayerData Layer::gradient(const Matrix &Y) {
2     LayerData res{dU * Y.transposed(), Matrix(n1, 1)};
3     for (int j = 0; j < n1; j++) {
4         res.db(j, 0) = 0;
5         for (int k = 0; k < s; k++) res.db(j, 0) += dU(j, k);
6     }
7     return res;
8 }
```

## Neurónové siete

```
1 Gradient Network::gradient(const Matrix &input) {
2     Gradient res(h);
3     for (int i = h - 1; i > 0; i--)
4         res[i] = std::move(layers[i].gradient(layers[i - 1].Y));
5     res[0] = std::move(layers[0].gradient(input));
6     return res;
7 }
```

```
1 void Network::apply(Num a, const Network::Gradient &g) {
2     for (int t = 0; t < h; t++) {
3         layers[t].W.addMultiple(-a, g[t].dW);
4         layers[t].b.addMultiple(-a, g[t].db);
5     }
6 }
```

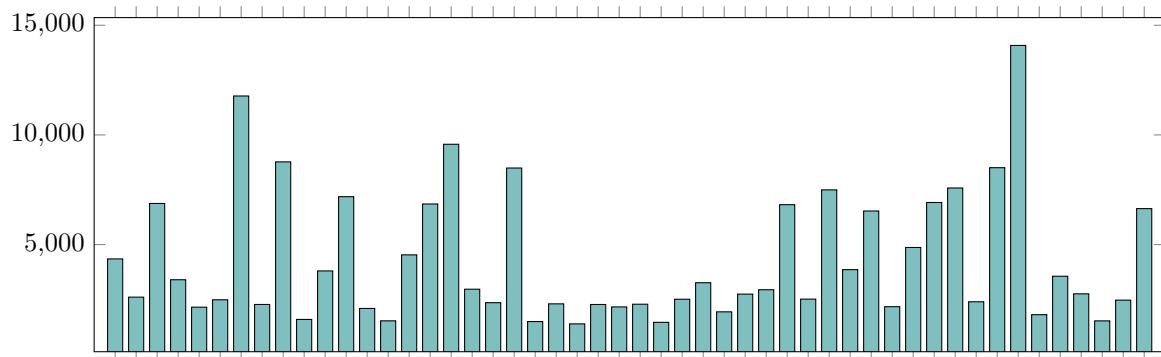
Pri trénovaní siete to potom vyzerá takto:

```
1 net.backPropagation(data, truth);
2 auto err = net.error(net.output(), truth);
3 auto g = net.gradient(data);
4 net.apply(step, g);
```

V riadku 2 som si zrátal chybu, ktorú môžem použiť napríklad na to, aby som vedel, kedy s trénovaním prestať. Parameter **step** mi hovorí, kolko z gradientu mám prirátať, t.j. ako ďaleko sa v smere gradientu pohnem. Zvolíš ho je celkom umenie: ak je príliš veľký, pôjdem v smere gradientu príďaleko a chyba siete narastie. Ak je príliš malý, trénovanie bude trvať pridlho. Môžeš si skúsiť rozmyslieť spôsob, ako **step** automaticky meniť (napr. ak chyba klesla, tak ho trochu zväčšíš, ak stúpla, tak ho zmenšíš).

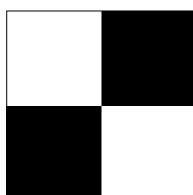
**Úloha 143.** Naprogramuj sieť, ktorá sa učí pomocou back propagation a otestuj ju na našom príklade s prvočíslami.

Ked som spúšťal túto vylepšenú sieť na príklade s 8-bitovými prvočíslami, kde náhodné učenie trvalo okolo 80000 iterácií, tak back propagation bolo spravidla veľmi rýchle, ale občas, ak náhodné nastavenie siete na začiatku dopadlo nejak zle, učenie trvalo dlho. Preto som si povedal, že ak sieť počíta viac ako 5000 iterácií a chyba je stále privelká, sieť zahodím a skúsim to celé od začiatku. Vo výsledku mi stačilo v priemere 4300 iterácií. To je oveľa menej, ako 80000 iterácií pri náhodnom učení, a pre väčšie siete by ten rozdiel bol ešte väčší. Zároveň je vidieť, že čas stále dosť závisel od toho, ako sa na začiatku sieť náhodne nastavila.

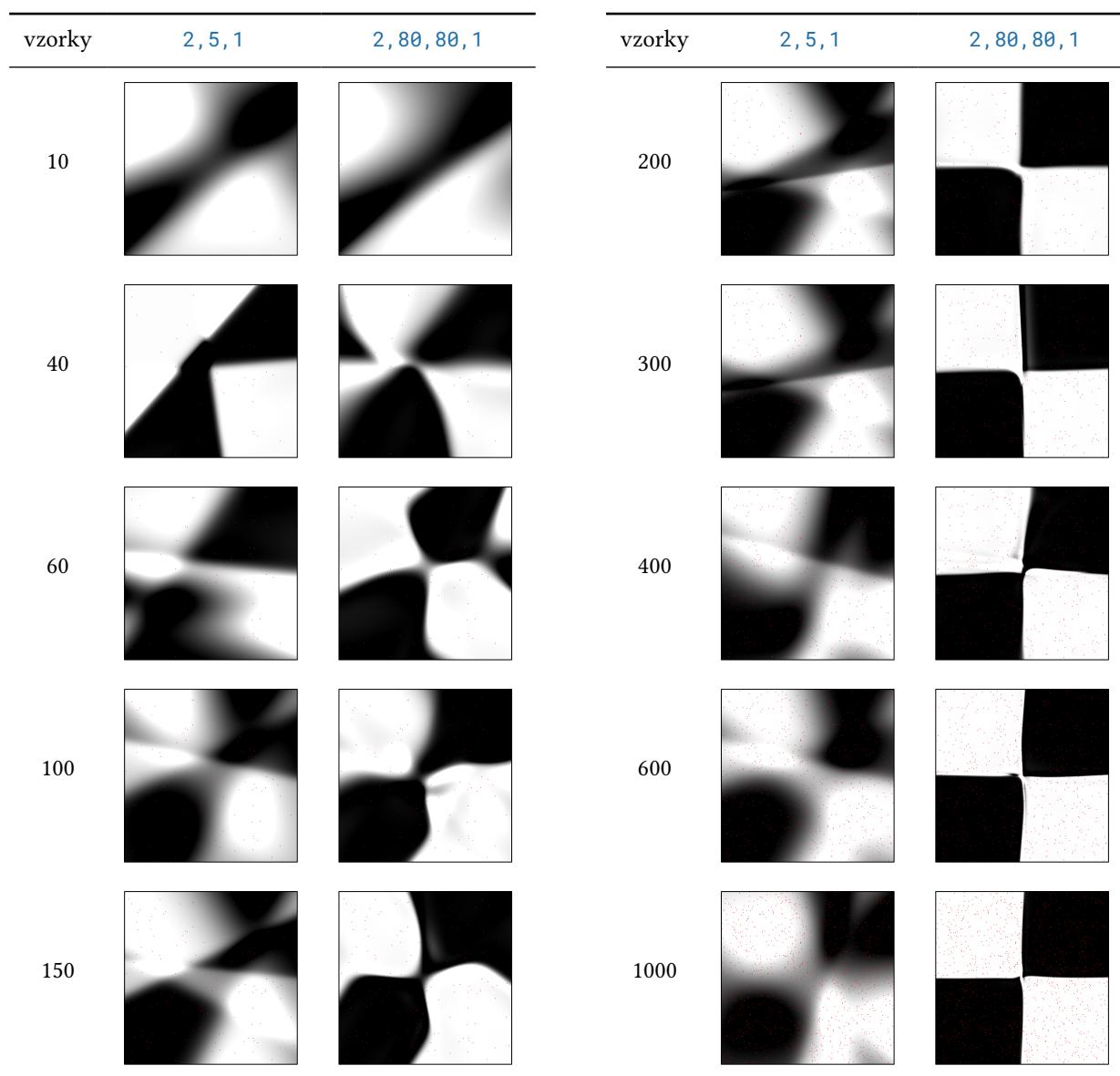


Doteraz sme sieť používali tak, že sme na nájdenie váh neurónov (a.k.a. *trénovanie*) použili všetky možné vstupy a výstupy úlohy. Čo ale s úlohami, ktoré majú priveľa možných vstupov? Urobíme to tak, že si vyberieme niekoľko vstupov a natrénujeme sieť iba na nich. Týmto nedostaneme presné riešenie úlohy, lebo o vstupoch,

ktoré sme na trénovanie nepoužili, nevieme nič. Môžeme sa ale spoliehať na to, že veľa úloh má takú vlastnosť, že vstupy, ktoré sú v istom zmysle podobné, budú mať aj podobné riešenie. A rovnakú vlastnosť má aj neurónová sieť: na podobné vstupy dáva podobnú odpoveď. Ako dobre to môže fungovať závisí od konkrétnej úlohy. Podľme si to vyskúšať na jednoduchom príklade. Dajme tomu, že chceme mať sieť s dvoma vstupnými neurónmi, ktoré budú mať hodnoty medzi  $-1$  a  $1$ . Tieto vstupné hodnoty sú súradnice bodu v rovine a mojím cieľom je povedať, akú farbu má bod s danými súradnicami v tomto vzore:

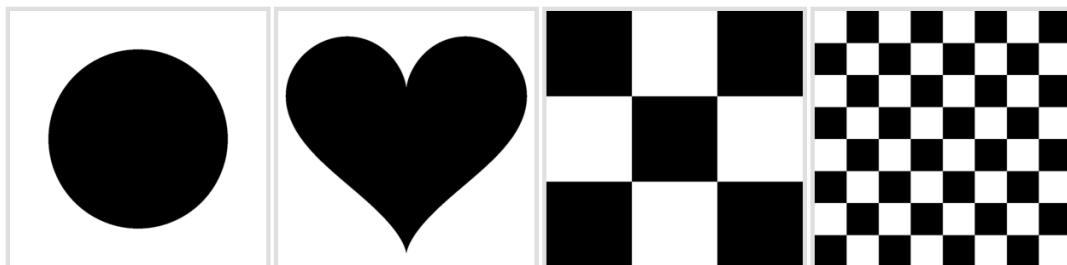


Zobral som si dve siete. Jedna mala tri vrstvy s  $2, 5, 1$  a jedným neurónom a druhá tri vrstvy s  $2, 80, 80, 1$  a jedným neurónom. Potom som pre rôzne počty náhodne vybratých bodov natrénoval obe siete a následne sa pozrel, aké odpovede dávali pre ostatné body.

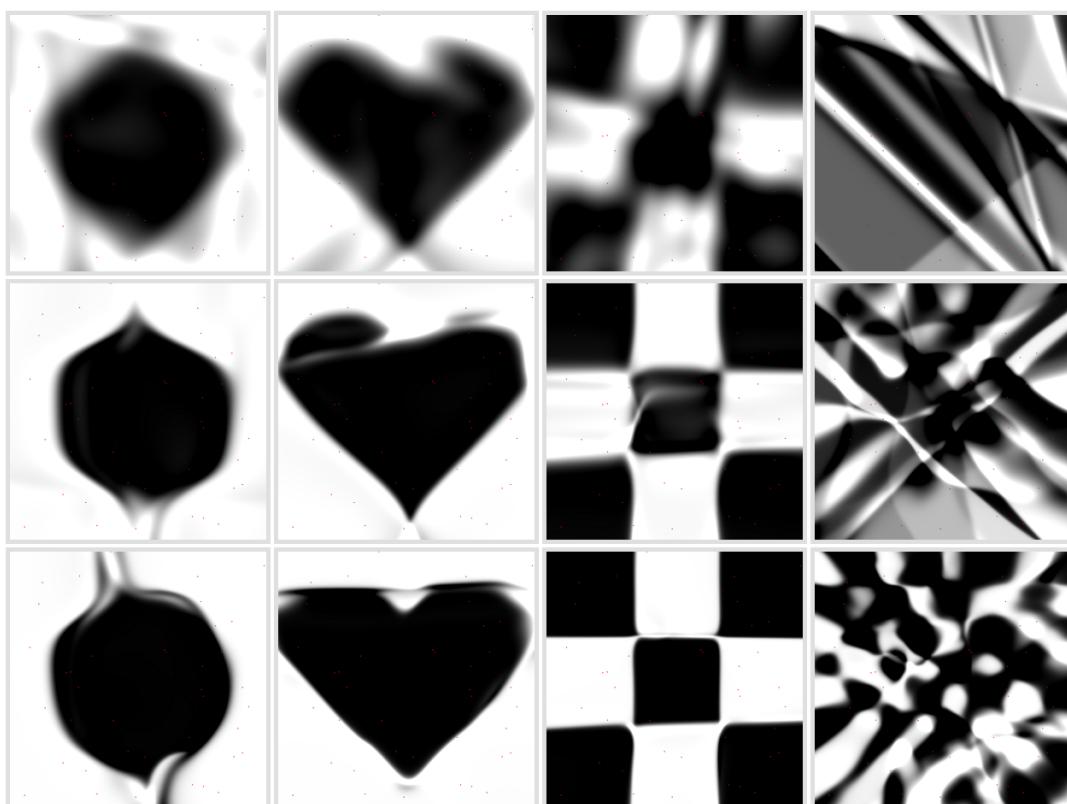


## Neurónové siete

Tu je dobre vidno, že malá sieť je príliš malá na to, aby sa vedela vzor naučiť, aj keď bolo na vstupe veľa bodov. Veľká sieť ale už pri 200 vzorkách dáva celkom slušné výsledky. Takže ak máme šťastie a úloha, ktorú riešime, má vhodnú štruktúru, môže nám stačiť pomerne málo testovacích príkladov na to, aby sa ju sieť naučila celkom rozumne riešiť. Na druhej strane, to "pomerne málo" môže byť pre zložitejšie úlohy dosť veľa a nevieme ho dopredu poriadne odhadnúť. Môžeme preto použiť nasledovný prístup: vyberieme si nejaký počet náhodných testovacích príkladov a urobíme niekoľko krokov back-propagation. Potom vyberieme iné náhodné príklady a zase na nich urobíme niekoľko krokov. V nasledujúcim experimente som si zobraľ tieto 4 vzory:



Zobraľ som tri siete: prvá mala vrstvy **2, 25, 1**, druhá **2, 100, 20, 1** a tretia **2, 850, 150, 10, 1**. V jednej epoche som vybraľ 40 náhodných bodov a spravil 200 iterácií back-propagation. Po 900 epochách to vyzeralo takto:

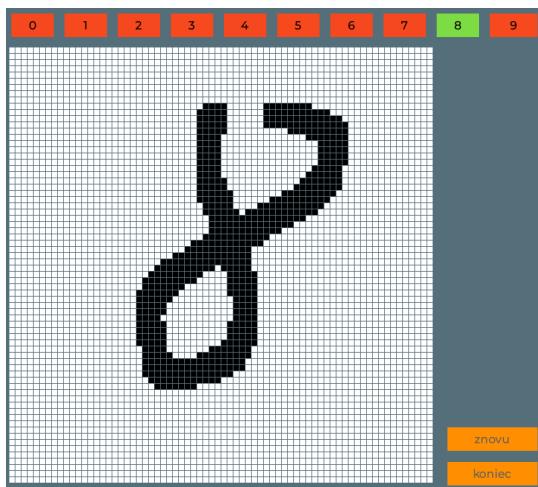


**Úloha 144.** Pozri si video<sup>13</sup> z predchádzajúceho experimentu. Čo sa z neho dá vidieť? Naprogramuj podobný experiment. Ako by sa dalo trénovanie urýchliť lepšou voľbou parametrov (veľkosť siete, veľkosť vzorky, dĺžka epochy)?

<sup>13</sup><https://github.com/pocestny/programovanie/raw/master/materialy/evolution.mp4>

## Projekt: rozpoznávanie písaných číslíc

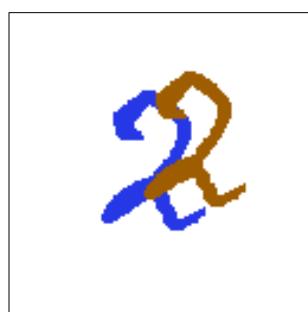
Výsledkom tohto projektu bude program, v ktorom môžeš myšou nakresliť číslicu, program ju rozpozná a zasveti príslušné číslo. Vyzerať by to malo nejak takto<sup>14</sup>:



Namiesto toho, aby som sa snažil naprogramovať tvary jednotlivých číslíc, natrénujem na rozpoznávanie neurónovú sieť: každý pixel zo vstupu bude jeden vstupný neurón s hodnotou od  $-1$  pre bielu po  $1$  pre čiernu. Na výstupnej vrstve bude 10 neurónov a každý bude reprezentovať jednu číslicu (čím väčšia hodnota neurónu, tým viac si sieť myslí, že je na vstupe daná číslica). Aby som sieť mohol trénovať, potrebujem veľa príkladov napísaných číslíc. Americký NIST (National Institute of Standards and Technology) má verejnú databázu<sup>15</sup> rukou písaných písmen a číslíc, ktorá sa dá použiť. Tá časť, čo ma zaujíma, sú rukou písané číslice, každá je čierno-biely obrázok rozmerov  $128 \times 128$  vo formáte png, napr.:



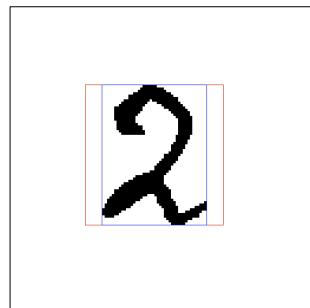
Na použitie v neurónovej sieti si ich potrebujem pripraviť. Jednak mať  $128 \times 128 = 16384$  vstupných neurónov by bolo priveľa a jednak by sieť bola citlivá na posunutie. Napr. tieto dve verzie dvojky sú takmer rovnaké, ale aktivované by boli úplne rôzne vstupné neuróny:



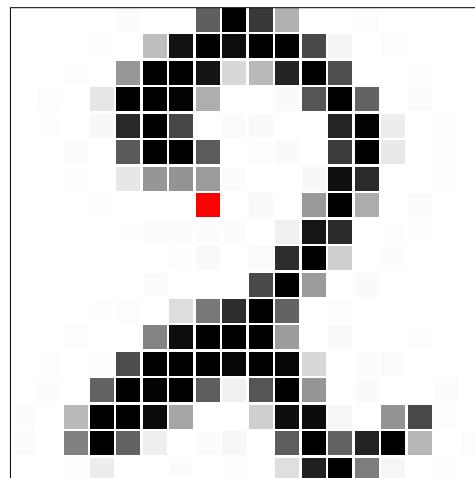
Povedal som si, že mi bude stačiť  $26 \times 26 = 676$  vstupných neurónov. Predspracovanie obrázka robím takto: zistím si bounding box, t.j. najmenší obdĺžnik, v ktorom sú všetky čierne pixely. Potom ho upravím na štvorec podľa dlhšej strany:

<sup>14</sup>Výsledný program si môžeš vyskúšať tu: <https://beda.dcs.fmph.uniba.sk/mnist/digits.html>

<sup>15</sup><https://www.nist.gov/srd/nist-special-database-19>



Štvorec vyrežem, zmenším na rozmer  $18 \times 18$  a nájdeme ťažisko čiernych (sivých) pixelov:



Napokon výsledok vycentrujem tak, aby ťažisko bolo v strede štvorca  $26 \times 26$ . Týmto spôsobom dostanem všetky číslice rovnako veľké a rovnako umiestnené. Ak budem tým istým spôsobom spracovávať aj nakreslené vzory v programe, nebudem mať problémy s rôzne veľkými a rôzne posunutými číslami. Výsledok spracovania niektorých čísel z trénovacieho datasetu vyzerá takto (všimni si drobné rozdiely v typickom písaní niektorých číslíc oproti Slovensku):

2	8	2	7	1	3	5	0	3	6	0	1	9	8	8	5	8	5	2	6	8	7	4	2	4	7	6	7	4	2	5	1	0	3	6	
7	8	7	6	4	3	9	6	7	7	1	3	3	9	6	7	5	9	9	4	0	1	1	5	9	2	4	2	7	4	2	3	8	8	2	
9	2	3	1	3	4	8	1	9	9	7	4	5	6	9	6	1	0	3	3	2	6	9	9	0	3	0	0	2	9	2	1	5	0	1	
6	5	8	7	9	5	3	3	2	0	1	0	0	5	2	5	8	8	9	8	6	6	6	3	5	2	9	5	8	0	7	1	8	3	8	
8	9	2	9	0	0	2	7	2	7	4	0	4	6	3	5	4	8	2	0	5	8	2	2	8	0	5	2	6	3	8	9	1	2	8	
7	7	1	2	5	1	5	9	4	9	6	5	3	2	2	3	3	9	1	6	8	0	0	2	5	9	5	6	9	4	2	5	2	2	1	
3	1	9	7	2	9	2	1	5	9	8	5	3	5	7	1	4	5	9	3	6	9	6	6	2	8	1	8	8	0	9	6	7	1	2	
4	8	5	7	2	4	7	0	0	4	9	3	7	8	2	3	2	2	6	3	0	2	3	5	4	5	9	1	8	8	3	7	9	8	7	1
6	3	2	4	1	5	4	1	7	8	6	8	1	5	7	3	8	8	5	4	1	2	3	3	1	1	0	2	6	6	5	2	0	2	4	
7	3	7	8	5	6	9	9	3	6	7	5	2	8	9	3	3	4	3	6	8	5	8	8	0	3	4	4	3	4	1	6	4	3	8	
2	3	1	6	5	2	3	6	2	9	5	2	9	1	6	2	1	3	6	0	3	6	2	8	5	2	0	3	2	4	7	8	4	3	1	
8	5	3	8	5	3	3	7	1	2	7	3	3	8	6	1	4	0	7	3	7	1	7	7	8	4	5	6	7	3	3	1	5	4	5	
6	5	3	9	2	3	1	4	7	4	6	0	2	6	2	9	7	5	3	6	6	5	9	2	7	7	4	6	0	4	4	7	8	4	9	
0	8	6	9	3	5	2	6	1	8	8	0	6	5	5	0	7	8	9	8	5	8	6	7	9	2	3	3	0	3	6	5	9	5	7	
1	5	7	0	4	0	6	9	4	5	1	7	1	0	6	1	8	5	6	5	8	9	1	3	9	4	8	9	3	1	8	9	0	3	6	

Vieme, čo chceme dosiahnuť, tak to môžeme začať programovať. Najprv si spravíme pomocné triedy pre pixel a obdĺžnik

```

1 struct Point {
2     int _p[2];
3     Point &operator=(const Point &a);
4     int &x() { return _p[0]; }
5     int &y() { return _p[1]; }
6     int x() const { return _p[0]; }
7     int y() const { return _p[1]; }
8     int &operator[](int i) { return _p[i]; }
9     int operator[](int i) const { return _p[i]; }
10    Point &operator+=(const Point &p);
11    Point &operator*=(double a);
12 };

```

```

1 struct Box {
2     Point p[2]; // Ľavý horný a pravý dolný okraj
3     Point &operator[](int i) { return p[i]; }
4     Point operator[](int i) const { return p[i]; }
5     int x() const { return p[1].x() - p[0].x(); }
6     int y() const { return p[1].y() - p[0].y(); }
7     Box &expand(const Point &q); // rozšír obdĺžnik tak, aby obsahoval bod q
8     Point center() const;
9 };

```

Doprogramovať zvyšné metódy by malo byť priamočiare. Ďalej si urobím pomocnú triedu na uloženie štvorcového obrázka rozmerov  $d \times d$ :

```

1 using u8 = uint8_t;
2
3 struct Bytes {
4     std::vector<u8> a;
5     int d, n;
6     u8 sentinel;
7     Bytes(int _d) : d(_d), n(d * d) {
8         a.resize(n, 255); // miesto na štvorec  $d \times d$ , celý biely
9     }
10    u8 &operator()(int x, int y);
11    u8 operator()(int x, int y) const;
12    Point pos(int i) const {
13        return Point{i % d, i / d}; // súradnice i-teho pixelu
14    }
15    Point cog() const; // tăžisko
16    Box bbox() const; // bounding box
17 };

```

Premennú `sentinel` som použil na to, aby som mohol kontrolovať, či parametre v `operator()` nie sú mimo rozsahu. Keďže vracam referenciu, potrebujem vrátiť niečo, kam sa dá beztrestne zapisovať.

```

1 u8 &Bytes::operator()(int x, int y) {
2     if (x < 0 || y < 0 || x >= d || y >= d) return sentinel;
3     return a[x + y * d];
4 }

```

Násť bounding box je jednoduché

## Neurónové siete

```
1 Box Bytes::bbox() const {
2     Box b{Point{n, n}, Point{0, 0}};
3     for (int i = 0; i < n; i++) {
4         if (a[i] < 255) b.expand(pos(i));
5     }
6 }
```

Ďalej budem potrebovať výrez naškálovať na rozmer  $18 \times 18$ . Aj keď sa to nezdá, škálovanie je celkom zložitý problém, ak ho chceš spraviť poriadne. Ak by som chcel iba napr. zmenšíť obrázok na polovicu, tak jednému pixelu výsledného obrázka prislúchajú štyri pixely (štovrec  $2 \times 2$ ) z pôvodného. Stačí mi preto spraviť priemer z tých štyroch a je to. Ak ale rozmer nového ovrázka nie je deliteľom pôvodného, začnú problémy. Pre naše účely to nie je zase také dôležité, ale rozhodol som sa spraviť výnimku a použiť externú knižnicu, konkrétnie škálovaciu knižnicu [Avir](#)<sup>16</sup>. Používa sa ľahko. Keďže všetko v nej je šablóna, stačí nakopírovať súbor [avir.h](#)<sup>17</sup> do pracovného adresára a použiť `#include "avir.h"`. Potom môžeš vyrobiť premennú typu<sup>18</sup> `avir::CImageResizer<>`. Konštruktor dostane ako parameter počet bitov na farbu, v našom prípade 8. Celú prácu potom urobí metóda `resizeImage`

```
1 avir::CImageResizer<>::resizeImage(
2     const u8 *src,                                // pointer na dátu pôvodného obrázka
3     const int src_w, const int src_h,               // rozmery pôvodného obrázka
4     int lineSize,                                 // veľkosť riadka, môže vždy zostať 0
5     u8 *dst,                                     // pointer na dátu výsledného obrázka
6     const int dst_w, const int dst_h,               // výsledné rozmery
7     const int numChan,                            // počet kanálov na pixel, pre nás 1
8                                         // (napr. pre RGB by bolo 3, pre RGBA 4)
9     const double k                               // krok algoritmu, tu vždy môže zostať 0
10 );
```

Posledná vec, ktorú potrebujem, je nájsť ťažisko. To spravím tak, že zrátam vážený priemer pixelov (biely má váhu 0, čierny váhu 255):

```
1 Point Bytes::cog() const {
2     Point p{0, 0};
3     int s = 0;
4     for (int i = 0; i < d * d; i++) {
5         u8 v = 255 - a[i];
6         Point q = v * pos(i);
7         s += v;
8         p += q;
9     }
10    for (int i = 0; i < 2; i++)
11        p[i] = (int)((double)(p[i]) / (double)s);
12    return p;
13 }
```

Celá funkcia na spracovanie obrázku vyzerá takto:

<sup>16</sup><https://github.com/avaneev/avir>

<sup>17</sup><https://github.com/avaneev/avir/raw/master/avir.h>

<sup>18</sup>tie prázdne zátvorky <> znamenajú, že `CImageResizer` je šablóna s default parametrom, podobne, ako keď majú default parametre funkcie. Konkrétnie `avir::CImageResizer<>` je to isté ako `avir::CImageResizer< avir::fpclass_def<float> >`

```

1 void processImage(Bytes &in, Bytes &out) {
2     Box bb = in.bbox();
3     int d = 1 + max(bb.x(), bb.y()); // dlhšia strana bounding boxu
4
5     // o je ľavý horný pixel vyrezaného štvorca
6     Point o{bb[0].x() - (d - bb.x()) / 2, bb[0].y() - (d - bb.y()) / 2};
7     Bytes crop(d); // sem uložím vyrezaný štvorec d x d
8     int t = 0;
9     for (int i = 0; i < d; i++)
10        for (int j = 0; j < d; j++) {
11            in.sentinel = 255; // ak som mimo rozsahu, prečítam bielu
12            crop.a[t++] = in(o.x() + j, o.y() + i);
13        }
14
15    Bytes rsz(sc); // sem uložím naškálovaný výsledok
16    imageResizer.resizeImage(crop.a.data(), d, d, 0, rsz.a.data(), sc, sc, 1, 0);
17
18    Point c = rsz.cog(); // tažisko
19    o = Point{dim / 2 - c.x(), dim / 2 - c.y() - 1}; // okraj vycentrovaného štvorca
20
21    for (int i = 0; i < sc; i++)
22        for (int j = 0; j < sc; j++)
23            out(o.x() + j, o.y() + i) = rsz(j, i);
24 }
```

Takto som spracoval celý dataset. Výsledok je v súbore `mnist.zip`<sup>19</sup> kde je pre každú číslicu jeden dlhý binárny súbor. V ňom je najprv počet obrázkov ako 32-bitový integer a za ním idú jednotlivé  $26 \times 26$  obrázky za sebou jeden byte na každý pixel, takže prečítať to viem takto:

```

1 vector<vector<Bytes>> dataset(10); // pre každú číslicu je zoznam obrázkov
2
3 for (int num = 0; num < 10; num++) {
4     stringstream ss;
5     ss << "./mnist/" << num << ".bin";
6     ifstream f(ss.str(), ios::binary);
7     uint32_t x;
8     f.read((char *)&x, 4); // prečítam 32 bitov (4 byty) počet obrázkov
9     while (x-- > 0) dataset[num].emplace_back(dim); // vytvorím miesto pre obrázky
10    for (auto &d : dataset[num])
11        f.read((char *)d.a.data(), dim * dim); // postupne všetky prečítam
12 }
```

**Úloha 145.** Natrénuj neurónovú sieť na datsete číslíc. Do triedy `Network` pridaj metódy `save` a `load` a natrénovanú sieť ulož do súboru.

Ja som použil sieť s vrstvami so **676, 300, 100, 10** neurónmi a trénoval som po epochách 1000 iterácií vždy na vzorke 200 náhodne vybratých číslíc.

Keď už je natrénovaná sieť uložená v súbore, ostáva spraviť program, ktorý ju používa a v ktorom sa dajú kresliť číslice. Na to sa dajú použiť widgety z kapitoly 38. Hlavné okno bude `VLayout mainWindow`, v ktorom budú dva `HLayouty`: `topBar` a `mainArea`. V `topBar` desať widgetov typu `Bulb`, ktoré zobrazujú jednotlivé číslice. V `mainArea` bude vľavo widget typu `Canvas`, ktorý sa stará o kreslenie a vpravo `VLayout buttonArea` s gombíkmi:

<sup>19</sup><https://github.com/pocestny/programovanie/raw/master/materialy/mnist.zip>

## Neurónové siete

```
1 Canvas *canvas = nullptr;
2 Bulb *bulbs[10];
3 ...
4
5 int main() {
6     ...
7     VLayout mainWindow;
8     canvas = new Canvas();
9     auto topBar = new HLayout();
10    topBar->fixedHeight = 52;
11    auto mainArea = new HLayout();
12    auto buttonArea = new VLayout();
13    buttonArea->fixedWidth = 150;
14    Button *quitButton = new Button("koniec");
15    Button *clearButton = new Button("znovu");
16
17    for (int i = 0; i < 10; i++) {
18        bulbs[i] = new Bulb(to_string(i));
19        (*topBar) << bulbs[i];
20    }
21
22    mainWindow << topBar << mainArea;
23    (*mainArea) << canvas << buttonArea;
24    (*buttonArea) << new Widget() << clearButton << quitButton;
25
26    ...
27 }
```

Widget **Bulb** môžeme spraviť modifikáciou **Button**: bude vždy disabled, aby sa naňho nedalo klikáť. Navyše bude mať metódu **set(double d)**, ktorá nastaví pozadie od červenej po zelenú. Na to môžem použiť **Gradient** z úlohy 119.

```
1 struct Bulb : Button {
2     static const Gradient g;
3     Bulb(const std::string &_txt) ;
4     void set(double val) ;
5 };
6
7 const Gradient Bulb::g("gradient.gpf");
```

Nakoniec treba dorobiť triedu **Canvas**. V nej budem mať **Bytes img** pre obrázok (napr.  $70 \times 70$ ), v metóde **irender** ho vykreslím zo štvorčekov. Zároveň budem v metódach **onMouseDown** a **onMouseUp** sledovať polohu myši a kresliť. Keďže myš sa môže pohybovať dosť rýchlo, je lepšie vždy vykresliť čiaru od posledného zapamätaného miesta po súčasné. Na to môžem použiť algoritmus kreslenia čiary z úlohy 59.

```
1 struct Canvas : Widget {
2     const int dim = 70;
3     int pxdim, spacing; // rozmer štvorčeka a medzeru medzi nimi
4                     // si budem nastavovať v resize
5     Bytes img;
6     bool painting;    // či je stlačený gombík myši
7     SDL_Point lastMouse; // posledná poloha myši, odtiaľ kreslím čiaru
8
9     Canvas();          // konštruktor
10    ~Canvas();
```

```

12 void clear();
13 void line(SDL_Point,SDL_Point); // vykresli čiaru
14 void resize(SDL_Rect scr); // nastaví všetko potrebné pri zmene okna
15 void render(SDL_Renderer *renderer);
16
17 void onMouseDown(SDL_MouseButtonEvent& e);
18 void onMouseUp(SDL_MouseButtonEvent& e);
19
20 ... // pomocné metódy
21 };

```

V hlavnom programe v pravidelných intervaloch spustím sieť na aktuálnom vstupe:

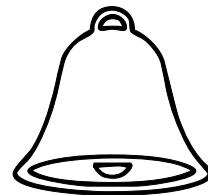
```

1 void run_net() {
2     auto &inp = canvas->img; // nakreslený obrázok
3     Matrix in(dim * dim, 1); // vstup pre sieť
4
5     Bytes tmp(inp.d); // pred zmenšením vstup
6                     // trochu rozmažem
7     int d = inp.d;
8     inp.sentinel = 255;
9     for (int i = 0; i < d; i++) {
10        for (int j = 0; j < d; j++) {
11            int v = 0;
12            for (int di : {-1, 0, 1})
13                for (int dj : {-1, 0, 1}) v += inp(i + di, j + dj);
14            tmp(i, j) = u8(v / 9);
15        }
16
17     Bytes img(26);
18     processImage(tmp, img); // zmenší a vycentruj
19
20     // vyrob vstup pre sieť a spusti ju
21     for (int i = 0; i < 26 * 26; i++)
22         in(i, 0) = 1.0 - 2.0 * (double)img.a[i] / 255.0;
23     auto &out = net.feed(res.in);
24
25     // nastav farby podľa výsledku
26     for (int i = 0; i < 10; i++) bulbs[i]->set(tanh(out(i, 0)));
27 }

```

**Úloha 146.** Naprogramuj GUI tak, aby zyeralo ako screenshot na začiatku kapitoly.

... a to je nateraz všetko. Je veľmi veľa vecí, ktoré sme tu nespomenuli, ale ako stručná odpoveď na otázku „*Ako sa programuje v C++?*“ to snáď stačí.



Programátorské projekty s otvoreným koncom, s ktorými sa dá hrať ďalej:

Celulárny automat .....	22
Offline grafický editor .....	57
Mandelbrotova množina .....	110
Kompresia textu .....	138
Mapy náhodných ostrovov .....	157
Hra Breakthrough .....	190
Rozpoznávanie číslic .....	275

Matematické odbočky:

Pytagorova veta, Euklidova vzdialenosť a kruhy .....	21
Goniometrické funkcie sin a cos .....	59
Komplexné čísla .....	108
Body, vektory a počítanie s nimi .....	164
Stredná (očakávaná) hodnota .....	182
Matice a vektory .....	251
Derviácie .....	259

Tu je zoznam kapitol a pojmov, čo sa v nich nachádzajú. Farebne sú odlíšené pojmy týkajúce sa **jazyka C++**, **všeobecne programovania** a **matematiky**.

## 1 Rýchly úvod 1

výraz a príkaz .....	1
výpis na cout .....	1
premenná .....	1
typ int .....	1
priradenie .....	1
vstup z cin .....	2
compile-time vs runtime .....	2

## 2 Podmienky 3

typ bool .....	3
základné aritmetické a logické operácie .....	3
príkaz if-else .....	3
zložený príkaz .....	3

## 3 Cykly 6

cyklus while .....	6
nedefinovaná hodnota .....	6

## Kde je čo

---

<i>zarážka (sentinel)</i>	6
<i>Collatzova hypotéza</i>	7
<i>Fibonacciho čísla, zlatý rez</i>	7
<i>vnorené cykly</i>	7
<b>4 Polia</b>	9
<i>pole</i>	9
<i>lenivé vyhodnocovanie podmienok</i>	9
<b>5 Ďalšie cykly</b>	11
<i>príkaz for</i>	11
<i>inkrement i++, i+=1</i>	11
<i>príkaz do-while</i>	11
<i>príkazy break, continue</i>	11
<b>6 Algoritmické úlohy a príkazový riadok</b>	13
<i>štandardný vstup/výstup</i>	13
<b>7 Čísla, znaky, reťazce</b>	14
<i>dvojková a šestnásťková sústava</i>	14
<i>adresa</i>	14
<i>typy long, unsigned, char</i>	15
<i>ASCII</i>	15
<i>string, t.j. reťazec znakov</i>	15
<i>whitespace</i>	16
<i>Segmentation fault</i>	16
<b>8 Dvojrozmerné polia</b>	18
<i>dvojrozmerné pole</i>	18
<b>9 Čiernobiele obrázky</b>	20
<i>Pythagorova veta</i>	21
<i>odmocnina</i>	21
<i>vzdialenosť bodov</i>	21
<i>bitové operácie</i>	23
<b>10 Funkcie</b>	25
<i>return</i>	25
<i>lokálne premenné</i>	25
<i>parametre funkcie</i>	26
<i>globálne premenné</i>	27
<i>typ void</i>	28
<i>viditeľnosť</i>	28
<i>modifikátor const</i>	29

---

<b>11</b>	<b>Rekurzia</b>	30
<i>najväčší spoločný deliteľ</i>	30	
<i>parsovanie výrazov</i>	30	
<i>noskipws</i>	31	
<i>Grayov kód</i>	33	
<i>Sierpiňského koberec</i>	34	
<b>12</b>	<b>Zložitosť</b>	35
<i>Insertion Sort</i>	36	
<i>CountSort</i>	36	
<i>lineárna, kvadratická a exponenciálna zložitosť</i>	37	
<i>parabola</i>	37	
<i>súčet prvých <math>n</math> prirodzených čísel</i>	38	
<b>13</b>	<b>Dátové štruktúry: zásobník</b>	42
<i>dátová štruktúra</i>	42	
<i>zásobník</i>	42	
<b>14</b>	<b>Memoizácia a dynamické programovanie</b>	45
<i>vedľajší efekt funkcie</i>	45	
<i>memoizácia</i>	46	
<i>dynamické programovanie</i>	47	
<b>15</b>	<b>Vlastné typy: struct</b>	53
<i>struct</i>	53	
<i>pretypovanie (konverzia)</i>	53	
<i>inicializačný zoznam (initializer list)</i>	53	
<b>16</b>	<b>(Ne)Reálne čísla</b>	55
<i>typy float a double</i>	55	
<i>vedecká notácia: mantisa a exponent</i>	55	
<i>typ float</i>	55	
<i>fixed</i>	55	
<b>17</b>	<b>Projekt: offline grafický editor</b>	57
<i>sklon priamky</i>	57	
<i>dolná celá časť</i>	57	
<i>sin, cos</i>	58	
<i>cmath</i>	59	
<i>korytnačia grafika</i>	60	
<b>18</b>	<b>Adresy a smerníky (pointre)</b>	63
<i>operátor &amp;</i>	63	
<i>typ pointer</i>	63	

## Kde je čo

---

<i>operátor *</i> .....	64
<i>pointrová aritmetika</i> .....	65
<i>pointer na pole a pole pointrov</i> .....	66
<b>19 Polia revealed</b>	67
<i>dynamická alokácia: new[ ] a delete[ ]</i> .....	69
<i>prvocíslo</i> .....	70
<i>MergeSort</i> .....	70
<b>20 Zásobník ako vlastný typ</b>	72
<i>zápis -&gt;</i> .....	73
<i>trieda, metódy, atribúty</i> .....	73
<i>konštruktor a deštruktor</i> .....	74
<i>preprocesor, direktíva #include</i> .....	75
<i>direktívky #define, #ifdef</i> .....	75
<b>21 STL: typ <code>vector</code> a <code>string</code></b>	77
<i>šablóna (template)</i> .....	77
<i>Standard Template Library (STL)</i> .....	78
<i>typ <code>vector</code></i> .....	78
<i>faktoriál</i> .....	78
<i>typ <code>string</code></i> .....	78
<i>stringstream</i> .....	79
<i>prikladanie ako výraz</i> .....	80
<b>22 Konštruktory, referencie, operátory a iný cukor</b>	81
<i>implicitný (default) konštruktor</i> .....	81
<i>dynamická alokácia: new a delete</i> .....	81
<i>poradie konštruktov, :</i> .....	82
<i>implicitné (default) parametre</i> .....	82
<i>referencia</i> .....	85
<i>operátor prikladania</i> .....	85
<i>copy constructor</i> .....	86
<i>predefinovanie operátorov</i> .....	87
<i>move constructor: lvalue, rvalue, xvalue, prvalue</i> .....	88
<i>modifikátor <code>const</code></i> .....	88
<b>23 Binárne vyhľadávanie a logaritmy</b>	90
<i>logaritmus</i> .....	90
<i>umocnenie na neceločíselný exponent</i> .....	90
<i>binárne vyhľadávanie</i> .....	91
<i>invariant</i> .....	92
<i>provádzelný rozklad</i> .....	92
<i>zložitosť MergeSortu</i> .....	96
<i>zápis zložitosti <math>O(f(n))</math></i> .....	97
<b>24 Algoritmy v STL</b>	99

<i>iterátory</i> .....	99
<i>vnořené typy</i> .....	99
<i>auto</i> .....	100
<i>type alias using</i> .....	101
<i>namespace</i> .....	101
<b>25</b> Funkcie ako parametre	103
<i>pointer na funkciu</i> .....	103
<i>funktor</i> .....	104
<i>lambda výrazy</i> .....	104
<i>šablóna function</i> .....	105
<i>captures</i> .....	105
<b>26</b> Matematické intermezzo: komplexné čísla	108
<i>odmocnia z 2 nie je racionálne číslo</i> .....	108
<i>Mandelbrotova množina</i> .....	110
<i>fraktál</i> .....	111
<i>optimalizácia -0</i> .....	111
<i>complex</i> .....	111
<i>hex-retazec farby</i> .....	112
<i>interpolácia</i> .....	112
<i>farebný gradient</i> .....	112
<i>fmod</i> .....	113
<b>27</b> Vyhľadávacie stromy: <i>&lt;set&gt;</i> a <i>&lt;map&gt;</i> v STL	115
<i>spájaný zoznam</i> .....	115
<i>binárny vyhľadávací strom</i> .....	118
<i>AA stromy</i> .....	120
<i>set</i> .....	125
<i>iterátory pre set a map</i> .....	126
<i>typ pair</i> .....	126
<i>for cyklus cez kontajner</i> .....	129
<b>28</b> Binárne súbory a kompresia	134
<i>fstream</i> .....	134
<i>konverzia stream na bool</i> .....	135
<i>seekg, tellg a podobné</i> .....	135
<i>read a write</i> .....	135
<i>bezprefixový kód</i> .....	139
<i>Huffmanovo kódovanie</i> .....	141
<i>halda (heap)</i> .....	142
<i>parametre z príkazového riadka</i> .....	145
<i>kompresia s prediktorm</i> .....	145
<i>Hutter prize</i> .....	146
<b>29</b> Náhoda a pravdepodobnosť	147
<i>pravdepodobnosť</i> .....	147

## Kde je čo

---

<i>porovnávanie súborov</i> .....	147
<i>počet prvočísel menších ako n</i> .....	148
<i>/dev/urandom</i> .....	149
<i>pseudonáhodný generátor, seed</i> .....	149
<i>int ako parameter šablóny</i> .....	149
<i>funkcia random()</i> .....	151
<i>knižnica random, random_device, mt19937</i> .....	151
<i>pravdepodobnosťné rozdelenia</i> .....	152
 <b>30 Meranie času, statické a privátne metódy</b>	153
<i>knižnica chrono</i> .....	153
<i>statické metódy</i> .....	153
<i>private</i> .....	154
<i>tiem_point::time_since_epoch()</i> .....	156
 <b>31 Projekt: mapy náhodných ostrovov</b>	157
<i>výšková mapa</i> .....	157
<i>inkrement i++ ako výraz</i> .....	158
<i>Perlin noise</i> .....	159
<i>bilineárna interpolácia</i> .....	161
<i>bod, vektor</i> .....	164
<i>vektor ako rozdiel bodov, sčítanie vektorov</i> .....	164
<i>normalizovaný vektor</i> .....	165
<i>skalárny súčin</i> .....	165
<i>vektorový súčin</i> .....	167
<i>normálne roviny</i> .....	167
<i>súbor gpf na farebné gradienty</i> .....	171
<i>Lambertov osvetľovací model</i> .....	172
<i>simulácia erózie</i> .....	175
 <b>32 Hešovanie: &lt;unordered_set&gt; a &lt;unordered_map&gt; v STL</b>	180
<i>hešovacia funkcia, kolízie</i> .....	180
<i>náhodná premenná</i> .....	182
<i>stredná hodnota</i> .....	182
<i>zápis sumy</i> .....	182
<i>súčet stredných hodnôt</i> .....	183
<i>indikátorové premenné</i> .....	184
<i>univerzálné hešovanie</i> .....	185
<i>unordered_set</i> .....	186
<i>špecializácia šablóny</i> .....	187
<i>struct std::hash</i> .....	188
 <b>33 Projekt Breakthrough</b>	190
 <b>34 Ako udržať poriadok vo veľkých projektoch</b>	191
<i>object code, linker</i> .....	191
<i>definícia vs. deklarácia</i> .....	192

<i>extern</i> .....	194
<i>inline metóda</i> .....	195
<i>šablóny a header súbory</i> .....	197
<i>Makefile</i> .....	199
<i>číselné typy int8_t, uint32_t a spol.</i> .....	202
<i>deklarácia funkcie bez názvu parametra</i> .....	204
<b>35 Prehľadávanie herných stromov</b>	205
<i>zero-sum hry, stratégia</i> .....	205
<i>MiniMax a NegaMax</i> .....	206
<i>orezávanie herných stromov</i> .....	208
<i>transpozičné tabuľky</i> .....	210
<i>murmur hash</i> .....	210
<i>evaluačná funkcia</i> .....	212
<i>iterative deepening a podobné vylepšenia</i> .....	213
<b>36 2D grafika: knižnica SDL</b>	214
<i>knižnica</i> .....	214
<i>SDL_Window, SDL_Renderer a vykresľovacie funkcie</i> .....	215
<i>priradenie do struct s menami položiek</i> .....	216
<i>udalosti, SDL_Event</i> .....	217
<i>typ union</i> .....	217
<i>typ std::variant</i> .....	218
<i>vstup z klávesnice</i> .....	219
<i>príkaz switch</i> .....	221
<i>SDL_GetTicks, SDL_Delay, SDL_PollEvent</i> .....	222
<i>SDL_Surface a SDL_Texture, SDL_Image</i> .....	224
<i>fonty a text</i> .....	226
<i>zvuk</i> .....	227
<b>37 Ako robiť viac vecí naraz</b>	231
<i>vlákno (thread)</i> .....	232
<i>thread, join, yield</i> .....	233
<i>vzájomné vylúčenie, mutex</i> .....	235
<i>lock_guard</i> .....	237
<i>counting_semaphore</i> .....	238
<i>producer-consumer</i> .....	238
<i>deque</i> .....	238
<i>deadlock</i> .....	239
<b>38 Dedičnosť</b>	241
<i>delenie</i> .....	242
<i>virtuálne metódy</i> .....	244
<i>volanie konštruktorov pri dedení</i> .....	245
<b>39 Neurónové siete</b>	250
<i>násobenie matíc</i> .....	251

## Kde je čo

---

<i>transponovaná matica</i> .....	252
<i>podmienkový výraz (bool)?A:B</i> .....	254
<i>trieda initializer_list</i> .....	254
<i>emplace_back()</i> .....	254
<i>funkcia tanh()</i> .....	256
<i>stredná kvadratická odchýlka</i> .....	257
<i>tangens</i> .....	259
<i>derivácia</i> .....	260
<i>derivácia mocniny, súčtu a súčinu</i> .....	260
<i>derivácia zloženej funkcie</i> .....	262
<i>gradient funkcie viacerých premenných</i> .....	263
<i>parciálna derivácia</i> .....	264
<i>derivácie zloženej funkcie vo viacerých rozmeroch</i> .....	265
<i>back propagation</i> .....	267
<i>šablóny s default parametrami</i> .....	278