

# Objektinis Programavimas

**2-oji praktinė užduotis: "Duomenų  
apdorojimas"**



# Duomenų apdorojimas

- Užduoties formuluotė
- Reikalavimai skirtingoms programos versijoms
- Versijos (v0.1) analizė
- Versiju (v0.2) ir (v0.3) analizė
- Versijos (v0.4) analizė
- Versijos (v0.5) analizė
- Versijos (v1.0) analizė

# Užduoties formuliuotė

- Parašykite programą, kuri nuskaito šiuos studentų duomenis:
  - **vardą** ir **pavardę**
  - **n** atliktų **namų darbų** (nd) rezultatus (10-baleje sistemoje), o taip pat **egzamino** (egz) rezultatai.
- Tuomet iš šių duomenų, suskaičiuoja galutinį balą (galutinis):

$$\text{galutinis} = 0.4 \times \frac{\sum_{i=1}^n \text{nd}_i}{n} + 0.6 \times \text{egz}$$

# Reikalavimai versijai (v0.1) (1)

Terminas: 2018-02-17 

- Pagal aprašytus užduoties reikalavimus realizuokite programą, kuri nuskaito vartotojų įvedamus reikiamus duomenis:
  - studento vardą ir pavardę;
  - namų darbų ir egzamino rezultatai;
- Baigus duomenų įvedimą, suskaičiuoja galutį balą ir juos pateikia į ekraną tokiu ar panašiu pavidalu (kur galutinis apskaičiuotas balas pateikiamas dviejų skaičių po kablelio tikslumu):

-----  
Pavardė      Vardas      VidGalutinis

-----  
Arvydas      Sabonis      x.xx

Rimas      Kurtinaitis      y.yy

...

- Papildykite programą, kad vietoj **vidurkio** galutinio balo skaičiavimui būtų galima naudoti ir **medianą**. Tuomet išvedimas (output'as):

-----  
Pavardė      Vardas      VidGalutinis      MedGalutinis

-----  
Arvydas      Sabonis      x.xx      x.xx

Rimas      Kurtinaitis      y.yy      y.yy

...

# Reikalavimai versijai (v0.1) (2)

Terminas: 2018-02-17 

- Papildykite programą taip, kad ji veiktu ir tokiu atveju, kai namų darbų skaičius (**n**) yra nežinomas iš anksto, t.y. tik įvedimo metu vartotojas nusprendžia kuomet jis jau įvedė visų namų darbų rezultatus. Šią dalį realizuoti reikytų dviem būdais, kur namų darbų rezultatus saugant j:
  - C masyvą.
  - std::vector tipo konteinerį.
- Atliekant šią užduotį galite sukurti du atskirus (\*.cpp) failus (arba du projektus), tačiau nuo versijos v0.2 naudosime tik vector'ius.
- Papildykite programą taip, kad būtų galimybė, jog mokinio gautieji balai už namų darbus bei egzaminą būtų **generuojami atsitiktinai**.

# Reikalavimai versijai (v0.2)

Terminas: 2018-02-22



- Papildykite programos versiją (v0.1) taip, kad būtų galima duomenis ne tik įvesti bet ir nuskaityti iš failo. Todėl sukurtite ir užpildykite failą kursiokai.txt, kurio (pleriminari) struktūra:

Pavardė	Vardas	ND1	ND2	ND3	ND4	ND5	Egzaminas
Vardas1	Pavardė1	8	9	10	6	10	9
Vardas2	Pavardė2	7	10	8	5	4	6
...							

- Papildykite programą taip, kad nuskaičiuos duomenis iš failo, išvedimas pleriminariai atrodytu taip:

Pavardė	Vardas	VidGalutinis	MedGalutinis
-----			
Arvydas	Sabonis	x.xx	x.xx
Rimas	Kurtinaitis	y.yy	y.yy
...			

**Reikalavimai output'ui:** studentai **turi būti surūšiuoti** pagal vardus (ar pavardes) ir visi stulpeliai būtų gražiai "išlygiuoti".

# Reikalavimai versijai (v0.3)

Terminas: 2018-03-01 

- Atlikite versijos (v0.2) kodo reorganizavimą (**refactoring'ą**):
  - Kur tikslina, programoje naudokite (jeigu dar nenaudojote) struct'ūras;
  - Funkcijas, naujas duomenų tipus (struct'ūras) perkelkite į antraštinius (angl. **header (\*.h)**) failus, t.y. tokiu būdu turėtumete projekte turēti kelis \*.cpp failus, kaip ir kelis \*.h failus.
- Kur tikslina, bent minimaliai panaudokite išimčių valdymą (angl. **Exception Handling**)<sup>eh</sup>

```
try { // išimtys yra apdorojamos žemiau
    // kodas, kuris atlieka tam tikras užduotis
} catch (std::exception& e) {
    // kodas, kuris apdoroja išimtis
}
```

**Kam viso to reikia?** O pvz. kas atsitiks, jeigu failas, kurį bandote atidaryti neegzistuoja; arba bandote gauti masyvo elementą, kuris neegzistuoja?

<sup>eh</sup> [https://www.tutorialspoint.com/cplusplus/cpp\\_exceptions\\_handling.htm](https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm)

# Reikalavimai versijai (v0.4)

Terminas: 2018-03-01



- Patobulinkite (jeigu reikia pagal v0.1 paskutinę užduotį turimą realizaciją) ir sugeneruokite **penkis** atsitiktinius studentų sąrašų failus, sudarytus iš: 10, 100, 1000, 10000, 100000 įrašų. Vardus ir Pavardes galite generuoti "šabloninius", kaip pvz. Vardas1 Pavarde1, Vardas2 Pavarde2 ir t.t.
- Sūrušiuokite (padalinkite) studentus į dvi kategorijas:
  - Studentai, kurių **galutinis balas < 5.0** galėtume vadinti "vargšiukai", "nuskriaustukai" ir pan.
  - Studentai, kurių **galutinis balas >= 5.0** galėtume vadinti "kietiakiai", "galvočiai" ir pan.
- **Atlikite programos veikimo spartos analizę:** t.y. išmatuokite (kuo tiksliau) **visos** programos (t.y. failų kūrimą irgi reikia matuoti) veikimo laiką testuojant su šiais penkiais skirtingo dydžio duomenų failais.

# Reikalavimai versijai (v0.5)

Terminas: 2018-03-08 

- **Konteinerių testavimas:** Išmatuokite programos veikimo spartą (be failų generavimo, nes visais atvejais turite naudoti tuos pačius **fiksuotus** sugeneruotus studentų duomenų failus) priklausomai nuo naudojamo vieno iš trijų konteinerių:
  - std::vector
  - std::list
  - std::deque
- Jeigu Jūs turite susikurę struktūrą **Studentai** (ar kaip jūs ją pavadinote) ir iki šiol naudojote std::vector<Studentai>, tai turite ištirti: ar pasikeistų ir kaip pasikeistų sparta, jei vietoje std::vector<Studentai> naudotumėte std::list<Studentai> ir std::deque<Studentai>.

# Reikalavimai versijai (v1.0) (1)

Terminas: 2018-03-15



- **Optimizuokite studentų rūšiavimo (dalijimo) į dvi kategorijas ("vargšiukų" ir "kietiakų") realizaciją:** t.y. visiems trims konteinerių tipams (vector, list ir deque) išmatuokite programos veikimo spartą priklausomai nuo studentų dalijimo į dvi kategorijas strategijos:
  - **1 strategija:** Bendro **studentai** konteinerio (vector, list ir deque tipų) skaidymas (rūšiavimas) **į du naujus to paties tipo konteinerius: "vargšiukų" ir "kietiakų".** Tokiu būdu tas pats studentas yra dvejuose konteineriuose: bendrame **studentai** ir viename iš suskaidytų (**vargšiukai** arba **kietiakai**). Nesunku pastebėti, kad tokia strategija yra neefektyvi užimamos atminties atžvilgiu (jsitikinkite tuo!), tačiau šiame žingsnyje svarbiausia yra patyrinėti, kaip programos veikimo sparta priklauso nuo konteinerio?
  - **2 strategija:** Bendro studentų konteinerio (vector, list ir deque) skaidymas (rūšiavimas) **panaudojant tik vieną naują konteinerį: "vargšiukai".** Tokiu būdu, jei studentas yra vargšiukas, jį turime įkelti į naujają **"vargšiukų"** konteinerį ir ištrinti iš bendro **studentai** konteinerio. Po šio žingsnio **studentai** konteineryje liks vien tik **kietiakai**. Atminties atveju tai efektyviau, tačiau dažni trynimai gali būti "**skausmingi**", ypač tam tikro tipo konteineriams.
- Jeigu Jūsų šiuo metu realizuota strategija nesutampa nė su viena iš šių dviejų aukščiau aprašytų strategijų, turėsite palyginti tris strategijas: Jūsų ir abi aukščiau aprašytas strategijas.

# Reikalavimai versijai (v1.0) (2)

Terminas: 2018-03-15 

- Programos efektyvumas gali stipriai priklausyti ne tik nuo naudojamo konteinerio tipo, bet ir nuo naudojamų algoritmų. Susipažinkite su žemiau pateiktais:
  - std::find()
  - std::find\_if()
  - std::search()
  - std::copy()
  - std::remove()
  - std::remove\_if()
  - std::remove\_copy()
  - std::remove\_copy\_if()
  - std::transform()
  - std::partition()
  - std::stable\_partition()
- Pabandykite iš jų atsirinkti ir pritaikyti tinkamus algoritmus studentų dalijimo procedūrai paspartinti (optimizuoti) ant vieno fiksuoto konteinerio - **vektorius**. Palyginkite programos veikimo spartą po šių pakeitimų.

# Reikalavimai versijai (v1.0) (3)

Terminas: 2018-03-15 

- Galutinėje versijoje v1.0 turi būti pateikta:
  - Tvarkinga **github** repozicija, kurioje būtų tik Jūsų kurti (source) failai, t.y. jokių naudojamo IDE "šiukšlių".
  - README.md faile aprašyti visi releasai, bei pakomentuoti gauti rezultatai.
  - Parengta naudojimosi instrukcija, t.y. pagrindiniai žingsniai aprašyti tame pačiame README.md faile.
  - Parengta įdiegimo instrukcija, t.y. paruoštas **make** Makefile (Unix OS atveju) arba **cmake** CMakeLists.txt (bet kokios OS atveju).

# Vertinimo kriterijai (1)

- Iki versijos (v0.5) svarbiausia, kad Jūsų programos atliktų tai, kas yra prašoma užduotyse.  
Jeigu yra prašomą kažką panaudoti, pvz. antraštės (**header**) failus, iššimtis (**exceptions**) ir pan.- vadinasi juos ir turite panaudoti Jūsų realizacijoje. Tačiau tikrai nebus baudžiama už tai, jeigu Jūsų realizacijos nebus labai efektyvios ar "modernios" - tą mes mokysimės ir tobulėsime viso kurso (kaip ir tolesnio gyvenimo) metu - svarbu daryti sąvarankiškai, tačiau diskutuoti su kolegomis ir dėstytojais drąsiai!
- Tuomet natūralus klausimas - o už ką gi bus baudžiama, jeigu yra svarbu tik kad veiktu? Ogi bus baudžiama už tai, jeigu veiks "bug'ovai" 😊 Pvz.:
  - Kas nutiktų, jeigu ten kur reikia "rankomis" suvesti duomenis aš vietoj balo įvedu kažkokį kitą simbolį, pvz. "s" raidę? Ar Jūsų programa "neužlūžtų"? Juk natūralu, kad taip neturėtų įvykti, o programa tiesiog turėtų informuotu apie situaciją ir pagal ją priimtu atitinkamą sprendimą.
  - Kas nutiktų, jeigu nejvesčiau nė vieno namų darbų? Ar nesigautų dalyba iš nulio?
  - Kas nutiktų, jeigu studentų duomenų failas neegzistuoja? Arba tame toje vietoje kur turi būti namų darbų/egzamino balai, būtų ne skaičius, o koks nors kitas simbolis, pvz. "s" raidė?

## **Vertinimo kriterijai (2)**

- Bus vertinamas pačio darbo ir rezultatų apipavidalinimas, t.y.:
  - Kiekvienai versijai turi būti padarytas atskiras relyzas, o įvairūs efektyvumo tyrimai būtų išsamiai aprašyti ir pakomentuoti Jūsų repozicijos README.md faile.
  - Galiausiai, bus atsižvelgiama į suplanuotų darbų atlikimo grafiko, t.y. bus baudžiama: už nustatytu terminu nesilaikymą.

# Versijos (v0.1) analizė (1)

## Rezultatų (realiųjų skaičių) pateikimas norimu tikslumu

- <iomanip> antraštės (header) faile deklaruotas **manipulatorius** - std::setprecision, leidžia kontroliuoti kiek reikšmingų skaitmenų būtų naudojama pateikiant (output'inant) rezultataą.
- Tačiau kai naudojame std::endl, kuris taip pat yra manipulatorius, mums nereikia įtraukti <iomanip> antraštės. **Kodėl?**

# Versijos (v0.1) analizė (2)

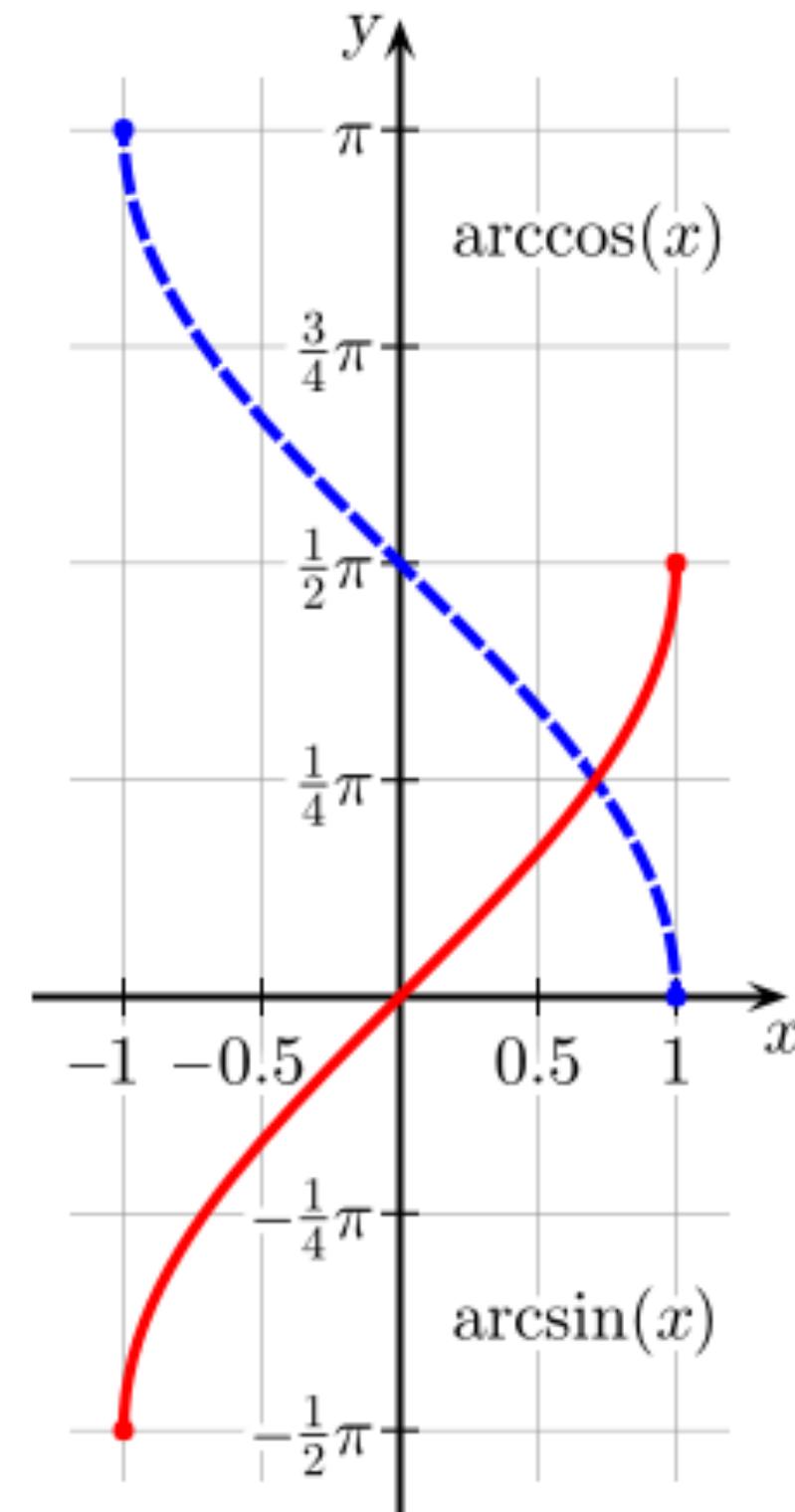
## std::setprecision()<sup>1</sup> pavyzdys

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <limits>

int main() {
    const long double pi = std::acos(-1.L); // pi = arccos(-1)
    std::cout << "default precision (6): " << pi << '\n'
        << "std::setprecision(10): " << std::setprecision(10) << pi << '\n'
        << "max precision: "
        << std::setprecision(std::numeric_limits<long double>::digits10 + 1)
        << pi << '\n';
}
```

```
default precision (6): 3.14159
std::setprecision(10): 3.141592654
max precision: 3.141592653589793239
```

<sup>1</sup> [http://en.cppreference.com/w/cpp/io/manip/  
setprecision](http://en.cppreference.com/w/cpp/io/manip/setprecision)



# Versijos (v0.1) analizė (3)

## Galutinio balo spausdinimas 2 skaičių po kablelio tikslumu

```
// išsaugome numatytaį (default) std::cout spausdinimo tikslumą
streamszie prec = cout.precision(); // C++11: auto prec
cout << "Galutinis balas yra " << setprecision(3)
      << 0.6 * egzaminas + 0.4 * suma / n // n - ND skaičius
      << setprecision(prec) << endl; // Grąžiname default tikslumą
```

# Versijos (v0.1) analizė (4)

## **EOF (end-of-file) signalas**

- Vienas iš būdų, kaip užbaigti duomenų įvedimą (cikle) yra **end-of-file (EOF)** signalo pasiuntimas.
- Skirtingos C++ realizacijos siūlo skirtingus būdus kaip tokį signalą į programą pasiūsti. Labiausiai paplitęs būdas yra:
  - pradėti naują eilutę (spaudžiame Enter)
  - tuomet spaudžiame Ctrl+z (Windows) arba Ctrl+d (Unix).

# Versijos (v0.1) analizė (5)

## Du EOF (end-of-file) naudojimo pavyzdžiai<sup>2</sup>

```
while(!cin.eof()) { // EOF tikrinimas čia
    cin >> x;          // kai nuskaityti nepavyksta, EOF tampa true
    // use x           // OUCH: naudojame x, nors nieko nenuskaitėme!
}
```

**VS.**

```
while(cin >> x) { // Bando nuskaityti į x, grąžina false kai nepavyksta
                  // Dabar ciklą vykdys tik kai nuskaityti pavyko!
}
```

<sup>2</sup> <https://stackoverflow.com/questions/4533063/how-does-ifstreams-eof-work>

# Versijos (v0.1) analizė (6)

**while (cin >> x) reiškinio analizė**

Norėdami iš std::cin **nuskaityti** reikšmę į **x** ir **patikrinti** ar pavyko nuskaitymas:

```
if (cin >> x) {/*...*/}
```

Tai yra ekvivalentu:

```
cin >> x;  
if (cin) { /* ... */ }
```

Todėl std::cin naudojimas **if** ar **while** sąlygose yra ekvivalentus sąlygai: ar paskutinis bandymas nuskaityti iš cin buvo sėkmingas.

# Versijos (v0.1) analizė (7)

## Kada skaitymas iš srauto std::cin yra nesėkminges?

1. Mes galėjome pasiekti įvesties failo pabaigą.
2. Galbūt susidūrėme su įvestimi, nesuderinama su kintamojo, j kurį bandome nuskaityti reikšmę, tipu.
  - Pvz., taip gali atsitikti, jei mes bandome nuskaityti į int tipo kintamąjį, bet nuskaitoma reikšmė yra ne skaičius.
3. Sistemoje gali būti aptikta įvesties įrangos gedimas.
  - Visais šiaisiai atvejais std::cin reikšmė bus false.

# **Versijos (v0.1) analizė (8)**

## **Srauto (stream) būsenos (1)**

- Srautas turi būseną (apibrėžtą konstantą), kuri nusako, ar I/O (**Input/Output**) buvo sėkminga, o jei ne – kokia to priežastis:

<b>Būsena</b>	<b>Reikšmė</b>
goodbit	Viskas OK
eofbit	Pasiekta failo pabaiga
failbit	Klaida; I/O operacija nebuvo sėkminga
badbit	Esminė klaida; neapibrėžta būsena

# **Versijos (v0.1) analizė (9)**

## **Srauto (stream) būsenos (2)**

- failbit - reiškia, kad operacija nebuvo tinkamai apdorota, bet srautas yra OK. Pavyzdžiui, tai nutinka, jei skaitmuo turėjo būti nuskaitomas, bet gautas simbolis yra raidė.
- badbit - reiškia, kad srautas yra sugadintas arba ryšys su srautu yra prarastas.
- eofbit - tradiciškai nutinka kartu su failbit, nes failo pabaiga (end-of-file) nustato eofbit, bet tuo pačiu ir failbit, nes nieko nuskaityti nepavyko.

# Versijos (v0.1) analizė (10)

## Nežinomo skaičiaus namų darbų (nd) nuskaitymas

```
int n = 0;          // nd skaitiklis
double suma = 0.;   // nd įverčių suma
double x;           // kintamasis iš kurių nuskaityti
/* invariantas: iki šiol nuskaitėme `n' nd rezultatų,
   ir jų įverčių suma lygi `suma' */
while (cin >> x) {
    ++n;            // reikalauja pirmoji invarianto dalis
    suma += x;       // reikalauja antroji invarianto dalis
}
```

# Versijos (v0.1) analizė (11)

## Nežinomo skaičiaus namų darbų (nd) nuskaitymas į vektorių

```
// Kažkur aukščiau: using std::vector
vector<double> nd; // double tipo vektorius ND rezultatams
double x;           // kintamasis į kurį nuskaityti ND reikšmes
// invariantas: `nd' kaupia visus iki šiol nuskaitytus ND
while (cin >> x) {
    nd.push_back(x); // Nuskaitytą reikšmę įdedame į nd vekt. pabaigą
}
```

# Versijos (v0.1) analizė (12)

## Apsauga nuo tuščio nd vektoriaus

```
// patikriname ar įvedė nd rezultatus
typedef vector<double>::size_type vecSize;
vecSize size = nd.size(); // C++11: auto size = nd.size();
if (size == 0) { // `ekvivalentu': if (nd.empty())
    cout << "Privalote įvesti ND rezultatus."
        << "Bandykite iš naujo.\n";
    return 1;
}
```

- Ką reiškia return 1; ?

# Versijos (v0.1) analizė (13)

## Tipų apibrėžimas naudojant **typedef** ir **using keyword's**us

A **typedef**-name can also be introduced by an alias-declaration. The identifier following the **using** keyword becomes a **typedef**-name and the optional attribute-specifier-seq following the identifier appertains to that **typedef**-name. It has the same semantics as if it were introduced by the **typedef** specifier. In particular, it does not define a new type and it shall not appear in the type-id.<sup>3</sup>

<sup>3</sup> <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf>

# Versijos (v0.1) analizė (14)

**typedef ir using pavyzdžiai**

```
typedef int MyInt;  
// ekvivalentu:  
using MyInt = int; // Nuo C++11  
// Tuomet:  
MyInt x = 5;  
  
// Ypač patogu, kai ilgi tipai:  
using IntVectorIt = std::vector<int>::iterator;  
// Ekvivalentu:  
typedef std::vector<int>::iterator IntVectorIt;
```

# Versijos (v0.1) analizė (15)

## Medianos apskaičiavimo žingsniai

- Norėdami apskaičiuoti medianą, privalome:
  1. Išsaugoti nuskaitytas reikšmes, nežinant iš anksto, kiek reikšmių bus.
  2. Kai visos reikšmės nuskaitytos – išrūšiuoti jas **nemažėjimo** tvarka.
  3. Rasti surūšiuotos (**variacinės**) eilutės vidutinę reikšmę jeigu reikšmių yra nelyginis skaičius ir dviejų "vidutinių" reikšmių vidurkį, jei reikšmių yra lyginis skaičius.

# Versijos (v0.1) analizė (16)

## Medianos radimas

```
// surūšiuojame vekt. nd naudojant `std::sort` iš <algorithm>
sort(nd.begin(), nd.end()); // nuo begin iki end
// apskaičiuojame medianą
vecSize vid = size/2; // vidurinis elementas
double mediana;
mediana = size % 2 == 0 ? (nd[vid-1] + nd[vid]) / 2
                         : nd[vid];
```

- Sąlyginis operatorius yra santrumpa **if-then-else** išraiškai ir dažnai vadinamas **? :** operatoriumi.

# Versijų (v0.2 & v0.3) analizė (1)

## Programų skaidymas į mažesnes dalis

- Kaip ir daugumoje programavimo kalbų, C++ siūlo du pagrindinius (didelių) programų organizavimo būdus:
  - panaudojant funkcijas, kartais dar vadinamas paprogramėmis.
  - panaudojant duomenų struktūras.
- Be to, C++ leidžia apjungti funkcijas ir duomenų struktūras į **klases**, su kuriomis "draugausime" nuo kito darbo.
- Galiausiai, C++ leidžia išskaidyti programą į mažesnius atskirai kompiliuojamus failus, kuriuos apjungia po kompiliavimo.

# Versijų (v0.2 & v0.3) analizė (2)

## Funkcija medianos radimui

```
// Apskaičiuojame medianą: funkcija nukopijuoją visą vektorių
double mediana(vector<double> vec) {
    typedef vector<double>::size_type vecSize;
    vecSize size = vec.size();
    if (size == 0) // std::domain_error deklaruota <stdexcept>
        throw std::domain_error("negalima skaičiuoti medianos tuščiam vektoriui");
    sort(vec.begin(), vec.end()); // surūšiuojame vektorių į variacinę eilutę
    vecSize vid = size / 2;      // vidurinis vektoriaus elementas
    return size % 2 == 0 ? (vec[vid] + vec[vid-1]) / 2 : vec[vid];
}
```

- Kadangi mediana() funkciją galime naudoti ne tik ND rezultatams, todėl **universaliau** informuojame (naudojant throw) kai vektorius vec yra tuščias.

# Versijų (v0.2 & v0.3) analizė (3)

## Išimtys (exceptions)

- Kai vektorius yra tuščias, paleidžiame (**throw**) išimtį (**exception**).
- Kai programa paleidžia išimtį, programos vykdymas baigiasi toje programos vietoje, kur throw buvo įvykdyta, ir **kartu su išimties objektu** pereina į **kitą** programos vietą.
- Šiuo atveju išimtis yra std::domain\_error tipo, kuris apibrėžtas <stdexcept> šablone, kurios tikslas informuoti, kad funkcijos argumentas yra **už leistinų ribų**.
- Mes sukuriame std::domain\_error išimtį, o string (const char\*) informuojame, kas blogo įvyko. Ši informacija vėliau gali būti panaudota, pvz. diagnostiniame pranešime ar programos logikoje.

# Versijų (v0.2 & v0.3) analizė (4)

## Funkcija vidurkio apskaičiavimui

```
// Apskaičiuojame vektoriaus elementų vidurki
// saugiau ir sparčiau būtų: double vidurkis(const vector<double>& vec) ?
double vidurkis(vector<double> vec) {
    if (vec.size() == 0) // apsaugo nuo dalybos iš 0!
        throw std::domain_error("negalima skaičiuoti vidurkio tuščiam vektoriui");
    // Deklaruota <numeric> header'je
    return std::accumulate(vec.begin(), vec.end(), 0.0) / vec.size();
}
```

std::accumulate - Computes the sum of the given value **init** and the elements in the range [**first**, **last**).<sup>accum</sup>

<sup>accum</sup> <http://en.cppreference.com/w/cpp/algorithm/accumulate>

# (v0.2) ir (v0.3) kūrimas (3)

## Persidengiančios (overloaded) funkcijos galutiniam balui

```
/* pagal egzamino ir namų darbų (nd) iverti (vidurki, medianą)
   apskaičiuoja galutinį balą */
double galBalas(double egzaminas, double nd) {
    return 0.6 * egzaminas + 0.4 * nd;
}

// pagal egzamino ir namų darbų rezultatus suskaičiuoja galutinį balą
// ši funkcija nekopijuoją vektoriaus; tą atlieka mediana()
double galBalas(double egzaminas, const vector<double>& nd) {
    if (nd.size() == 0) // patikriname, ar atliko bent vieną ND
        throw std::domain_error("studentas neatliko nė vieno namų darbo");
    return galBalas(egzaminas, mediana(nd)); // overloading: galBalas(double, double)
}
```

Čia `const vector<double>&` reiškia "reference to vector of const double":

# non-const nuorodos

## Žvilgsnis į prieitį 😱

- l-values - objektais, turintys dedikuotą atminties adresą (pvz., kintamieji).
- r-values - laikini neturi dedikuotos atminties objektais.
- Nekonstantinės nuorodos **inicializuojamos** sukūrimo metu ir tik j "non-const l-values"

```
int x = 10;          // non-const l-value
const int y = 20;    // const l-value
int &ref = x;        // OK, ref ir x - tas pats; x yra non-const l-value
int &wrongRef;      // negalima, nuoroda turi būti inicializuota!
int &ref2 = y;       // negalima!, y yra const l-value
int &ref3 = 10;       // negalima!, 10 yra r-value
```

# const nuorodos

- Konstantinės nuorodos **inicializuojamos** sukūrimo metu į "non-const l-value", "const l-values", ir "r-values":

```
int x = 10;
const int y = 20;
const int &ref = x;      // OK, x yra non-const l-value
const int &ref2 = y;     // OK, y yra const l-value
const int &ref3 = 10;    // OK, 10 yra r-value
```

# references vs. pointers

- Nuorodos ir rodyklės yra stipriai susijusios – nuorodos elgiasi, analogiškai **dereferenced const** rodyklėms.
- Įprastai nuorodos kompiliatorių realizuotos naudojant rodykles:

```
int x = 10;  
int &ref = x;  
int * const ptr = &x; // galime keisti tik *ptr reikšmę
```

- čia ref ir \*ptr elgiasi taip pat, t.y. ref == \*ptr.

# Įvairūs int \* const variantai

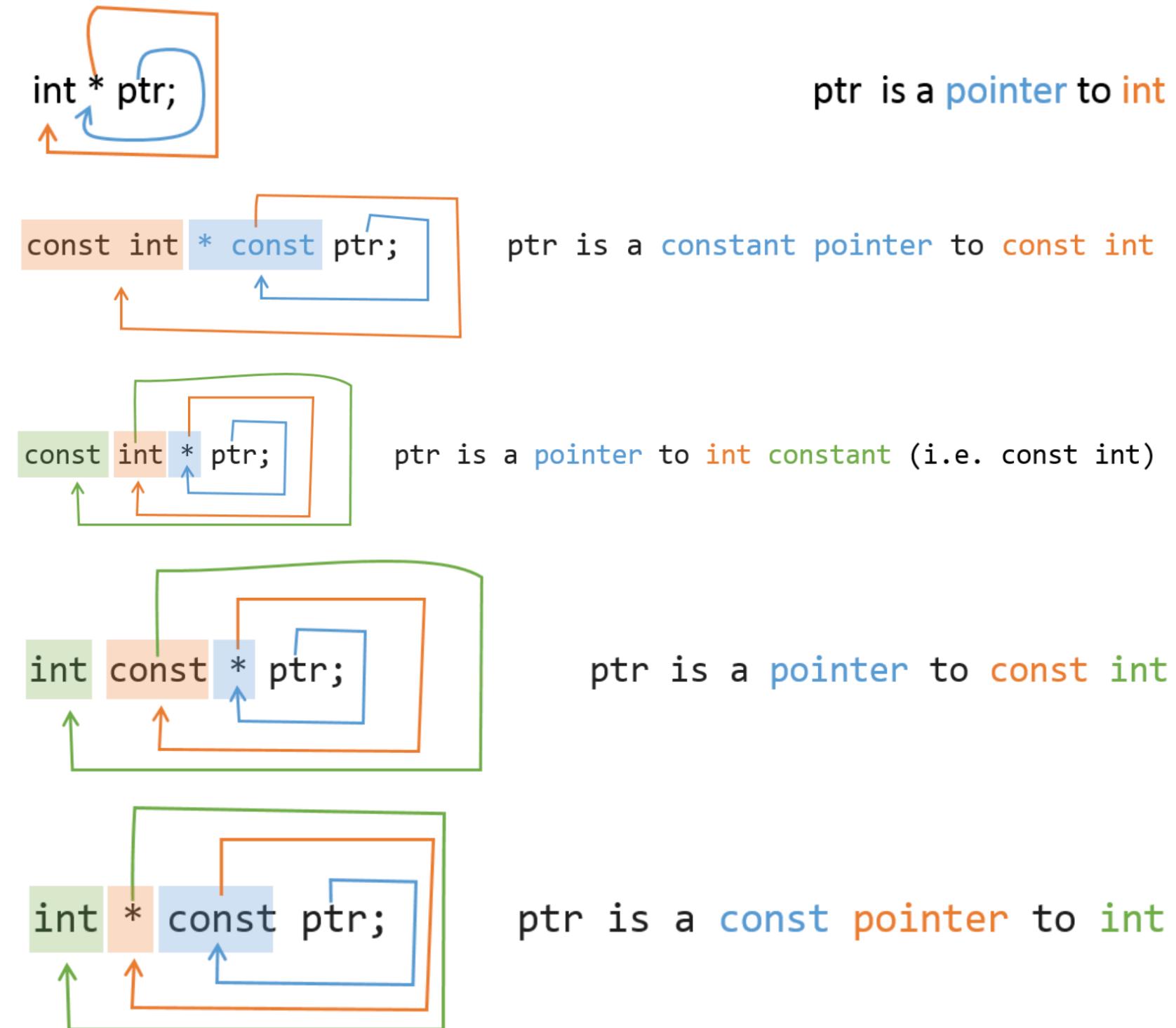
- Skaitome "žydiškai" (paremta **Clockwise/Spiral Rule**)<sup>4</sup>

<code>int * ptr</code>	- ptr is a pointer to <code>int</code>
<code>const int * ptr</code>	- ptr is a pointer to <code>const int</code>
<code>int const * ptr</code>	- ptr is a pointer to <code>const int</code>
<code>int * const ptr</code>	- ptr is a <code>const</code> pointer to <code>int</code>
<code>int const * const ptr</code>	- ptr is a <code>const</code> pointer to <code>const int</code>

- Pirmasis const gali būti abejose tipo pusėse:

```
const int *      == int const *
const int * const == int const * const
```

<sup>4</sup> <https://stackoverflow.com/questions/1143262/what-is-the-difference-between-const-int-const-int-const-and-int-const>



# Rodyklės į funkcijas (1)

```
int foo()      { return 5; }
int goo()      { return 6; }
int foo(int x) { return x; }

int main() {
    int (*fcnPtr)() = foo;      // fcnPtr nukreipta į funkciją foo()
    fcnPtr = goo;              // fcnPtr dabar nukreipta į funkciją goo()
    int (*fcnPtr2)(int) = foo; // fcnPtr nukreipta į funkciją foo(int)
    // Sintaksėje lengva susipainioti:
    std::cout << fcnPtr;       // Kompiliatoriui OK, bet atspausdina funkcijos adresą!
    std::cout << fcnPtr();     // OK: atspausdina 6
    std::cout << fcnPtr2(8);   // OK: atspausdina 8
    fcnPtr = goo();            // WRONG: priskiria funkcijos `goo()` reikšmę, vietoj adreso
    return 0;
}
```

The syntax for creating a non-const function pointer is one of the ugliest things you will ever see in C++<sup>learncpp</sup>

<sup>learncpp</sup> <http://www.learncpp.com/cpp-tutorial/78-function-pointers/>

# Rodyklės į funkcijas (2)

- **Tikslas**: patobulinti galBalas() funkciją, kad būtų galimybė perduoti funkciją - **kriterijų** (vidurkį ar medianą) namų darbų skaičiavimui.
- Kadangi abiejų funkcijų struktūra analogiška:

```
double mediana (vector<double> vec);  
double vidurkis (vector<double> vec);
```

- Todėl rodyklės į jas sintaksė yra:

```
double (*kriterijus)(vector<double>) = mediana;  
// Nuo C++11, galime naudoti: `std::function` iš <functional> header'io  
std::function<double(vector<double>)> kriterijus = mediana;
```

# (v0.2) ir (v0.3) kūrimas (4)

## Trečioji (overloaded) funkcija galutinio balo skaičiavimui

```
// pagal egzamino ir namų darbų rezultatus suskaičiuoja galutinį balą
// kriterijus: nusako medianą (DEFAULT) ar vidurkį naudosime ND skaičiavimui
double galBalas(double egzaminas, const vector<double>& nd,
                 double (*kriterijus)(vector<double>) = mediana) {
    if (nd.size() == 0) // patikriname, ar atliko bent vieną ND
        throw std::domain_error("studentas neatliko nė vieno namų darbo");
    return galBalas(egzaminas, kriterijus(nd)); // overloading: galBalas(double, double)
}
```

— Šia funkciją kreiptis galime vienu iš dviejų būdu:

```
galBalas(8.0, nd); // naudoja medianą (default) ND įverčiui
galBalas(8.0, nd, vidurkis); // naudoja vidurkį ND įverčiui
```

# (v0.2) ir (v0.3) kūrimas (5)

## Namų darbų rezultatų nuskaitymas

```
// iš įvedimo stream'o nuskaityti namų darbus į vector<double>
istream& readNd(istream& in, vector<double>& nd) {
    if (in) {          // jei stream būsena OK
        nd.clear();    // sunaikina vektoriaus elementus (jeigu buvo)
        double x;       // skaityti namų darbus
        while (in >> x) nd.push_back(x);
        in.clear();    // išvalo srautą, kad veiktų sekančiam studentui
    }
    return in;
}
```

`std::vector::clear` - Removes all elements from the container. Invalidates any references, pointers, or iterators referring to contained elements. Any past-the-end iterators are also invalidated. Leaves the `capacity()` of the vector unchanged.<sup>5</sup>

<sup>5</sup> <http://en.cppreference.com/w/cpp/container/vector/clear>

## **ND rezultatų nuskaitymas (2)**

- Srauto istream& readNd() gražinimas leidžia funkciją naudoti tokiu būdu:

```
if (readNd(cin, nd)) { /*...*/ }
```

```
// Priešingu atveju reikyt:
readNd(cin, nd);
if (cin) { /*...*/ }
```

**std::istream::clear** - Sets the stream error state flags by assigning them the value of state. By default, assigns std::ios\_base::goodbit which has the effect of clearing all error state flags.<sup>6</sup>

<sup>6</sup> <http://en.cppreference.com/w/cpp/container/vector/clear>

# (v0.2) ir (v0.3) kūrimas (6)

**Po šių patobulinimų, preliminari realizacija:**

```
/* PRIDĒTI: naudojamus `using`; reikiamus <header> failus; bei funkcijų:  
 * readNd(), mediana (vector<double>), galBalas(double, double),  
 * galBalas(double, const vector<double>&) realizacijas */  
int main() {  
    /* PRIDĒTI: realizaciją nuskaityti vardą, pavardę, egzamino rezultatą */  
    vector<double> nd;  
    readNd(cin, nd); // nuskaityti ND  
    try { // apskaičiuoti ir atspausdinti galutinių balų (jei įmanoma)  
        double galutinis = galBalas(egzaminas, nd);  
        streamsize prec = cout.precision();  
        cout << "Galutinis balas yra " << setprecision(3)  
            << galutinis << setprecision(prec) << endl;  
    } catch (std::domain_error) {  
        cout << endl << "Privalote įvesti studento rezultatus. "  
            "Bandykite iš naujo." << endl;  
        return 1;  
    }  
    return 0;  
}
```

# Kas gali throwinti?

- Reikia apgalvoti, kas gali paleisti (throw'inti) išimtj. Panagrinėkite tokj pvz.:

```
try { // pakeista galBalas() vieta
    streamszie prec = cout.precision();
    cout << "Galutinis balas yra " << setprecision(3)
        << galBalas(egzaminas, nd) << setprecision(prec);
}
```

- Jei galBalas() paleidžia išimtj, tai prieš tai tekstas netenka prasmės.
- Taip pat setprecision(3) liks neatstatytas į numatytaį (default).

# Duomenų organizavimas

- Tikslas patobulinti dabartinę realizaciją, taip, kad galėtume apdoroti visos studentų grupės duomenis pateiktus pvz. faile studentai.txt:

Pavardė	Vardas	Egz	ND1	ND2	ND3	ND4	ND5
Pavardė1	Vardas1	8	9	10	6	10	9
Pavardė2	Vardas2	7	10	8	5	4	6
...							

- Egz. balas prieš ND leidžia patogiau pritaikyti bet kokiam ND skaičiui.
- Reikia nuskaityti visų studentų duomenis, juos surūšiuoti ir išlygiuotai atvaizduoti. Tam patogu susikurti struktūrą, kurioje visi duomenys kartu.

## (v0.2) ir (v0.3) kūrimas (7)

### struct'ūros

- C++ kalboje tokią duomenų struktūra sukuriame tokiu būdu:

```
struct Studentas {  
    string vardas;  
    string pavarde;  
    double egzaminas;  
    vector<double> nd;  
}; // kabliataškis `;` yra būtinus!
```

- Struktūra "Studentas" yra **naujas tipas**, kuriame yra keturi duomenų nariai.
- Galime kurti "Studentas" tipo objektus (turinčius keturis duomenų narius); taip pat galime turėti konteinerius sudarytus iš Student'ų, pvz.: vector<Studentas> studVect;

# (v0.2) ir (v0.3) kūrimas (8)

## Adaptuotos (overloaded) funkcijos su Studentas tipo objektais

```
// nuskaityti duomenis iš streamo į `Studentas` objektą
istream& readStudent(istream& in, Studentas& s) {
    // nuskaityti ir išsaugoti vardą, pavardę ir egzamino rezultatą
    in >> s.vardas >> s.pavardė >> s.egzaminas;
    readNd(in, s.nd); // readNd(istream&, vector<double>&)
    return in;
}

// `Studentas` objektui apskaičiuoti galutinį balą pagal kriterijų
double galBalas(const Studentas& s,
                  double (*kriterijus)(vector<double>) = mediana) {
    return galBalas(s.egzaminas, kriterijus(s.nd)); // gali throw'inti
}
```

# **(v0.2) ir (v0.3) kūrimas (9)**

## **Studentų rūšiavimas (1)**

- Mes jau rūšiavome double tipo vektoriaus elementus mediana() funkcijoje:

```
vector<double> vec;  
sort(vec.begin(), vec.end());
```

- Analogija studentai vektoriui sudarytam iš Studentas narių yra negalima:

```
vector<Studentas> studentai;  
sort(studentai.begin(), studentai.end()); // blogai!
```

# (v0.2) ir (v0.3) kūrimas (10)

## Studentų rūšiavimas (2)

- Pirmiausia susikuriame Student'ų palyginimo funkciją, **predikatą**:

```
// rūšiuok pagal studentų vardus
bool compare(const Studentas& x, const Studentas& y) {
    return x.vardas < y.vardas;
}
```

- Tuomet studentai vektorių surūšiuojame naudodami tą pačią sort funkciją, tačiau su papildomu rūšiavimą nusakančiu argumentu compare:

```
sort(studentai.begin(), studentai.end(), compare);
```

# (v0.2) ir (v0.3) kūrimas (11)

## Studentų rūšiavimas (3)

- Galima turėti kelis rūšiavimo kriterijus (predikatus):

```
// rūšiuok pagal studentų pavardes
bool comparePagalPavarde(const Studentas& x, const Studentas& y) {
    return x.pavarde < y.pavarde;
}
// rūšiuok pagal studentų egzamino balus
bool comparePagalEgza(const Studentas& x, const Studentas& y) {
    return x.egzaminas < y.egzaminas;
}

// rūšiuojame pagal pavardes
sort(studentai.begin(), studentai.end(), comparePagalPavarde);
```

# (v0.2) ir (v0.3) kūrimas (12)

## Po naujausių patobulinimų, preliminari realizacija:

```
// Include visus reikiamus <header> failus, using'us
int main() {
    vector<Studentas> studentai;          // studentų vektorius
    Studentas student;                    // vienas student'as
    string::size_type ilgVardas = 0, ilgPavarde = 0; // ilgiausias vardas ir pavarde
    while (readStudent(cin, student)) { // nuskaityti visus įrašus
        ilgVardas = max(ilgVardas, student.vardas.size()); // max() iš <algorithm> header'io
        ilgPavarde = max(ilgPavarde, student.pavarde.size()); // max() narių tipai turi sutapti!
        studentai.push_back(student);
    }
    sort(studentai.begin(), studentai.end(), compare); // surūšiuojame pagal vardus
    for (vector<Studentas>::size_type i = 0; i != studentai.size(); ++i) {
        // parašyk vardą, užpildyk tuščia vieta priklausomai nuo ilg. vardo ir pavardės
        cout << studentai[i].vardas << string(ilgVardas + 1 - studentai[i].vardas.size(), ' ');
        cout << studentai[i].pavarde << string(ilgPavarde + 1 - studentai[i].pavarde.size(), ' ');
        try { // apskaičiuoti ir atspausdinti galutinių balus pagal du kriterijus (ND įverčiams)
            double galutinisMediana = galBalas(studentai[i], mediana);
            double galutinisVidurkis = galBalas(studentai[i], vidurkis);
            streamsize prec = cout.precision();
            cout << setprecision(3) << galutinisMediana << galutinisVidurkis << setprecision(prec);
        } catch (std::domain_error e) {
            cout << e.what(); // atspausdiname klaidos pranešimą ir pereiname prie kito studento
        }
    }
    return 0;
}
```

# (v0.2) ir (v0.3) kūrimas (13)

**Panaudojant foreach (C++11) vidinis for ciklas kompaktiškesnis**

```
for (const auto& stud : studentai) {
    // parašyk vardą, užpildyk tuščios vietas
    cout << stud.vardas << string(ilgVardas + 1 - stud.vardas.size(), ' ');
    cout << stud.pavarde << string(ilgPavarde + 1 - stud.pavarde.size(), ' ');
    try { // apskaičiuoti ir atspausdinti galutinių balus pagal du kriterijus (ND įverčiamas)
        double galutinisMediana = galBalas(stud, mediana);
        double galutinisVidurkis = galBalas(stud, vidurkis);
        streamszie prec = cout.precision();
        cout << setprecision(3) << galutinisMediana << galutinisVidurkis << setprecision(prec);
    } catch (std::domain_error e) {
        cout << e.what(); // atspausdiname klaidos pranešimą ir pereinam prie kitos
    }
}
```

# (v0.2) ir (v0.3) kūrimas (14)

## Source failai: mediana.{cpp, cc, cxx}

- C++, kaip ir daugelis programavimo kalbų, palaiko atskirą kompiliavimą, todėl programą galime išskaidyti į mažesnius atskirai kompiliuojamus failus.

```
// pirminis source failas (mediana.cpp) medianos funkcijai
#include <algorithm> // gauti deklaraciją sort()
#include <stdexcept> // gauti deklaraciją domain_error()
#include <vector> // gauti deklaraciją vector
#include "mediana.h"; // gauti deklaraciją mediana()

using std::domain_error; using std::sort; using std::vector;

double mediana(vector<double> vec) { // nukopijuojame vektorių
    typedef vector<double>::size_type vecSize;
    vecSize size = vec.size();
    if (size == 0) // std::domain_error deklaruota <stdexcept>
        throw std::domain_error("negalima skaičiuoti medianos tuščiam vektoriui");
    sort(vec.begin(), vec.end()); // surūšiuojame vektorių į variacinę eilutę
    vecSize vid = size / 2; // vidurinis vektoriaus elementas
    return size % 2 == 0 ? (vec[vid] + vec[vid-1]) / 2 : vec[vid];
}
```

# (v0.2) ir (v0.3) kūrimas (15)

## Header failai: mediana.{h, hh,.hpp}

- Header failuose turi būti naudojami tik **būtini** elementai nepamirštant header GUARD'ų:

```
// header failas: mediana.h
#ifndef GUARD_MEDIANA_H
#define GUARD_MEDIANA_H

#include <vector>
// mediana funkcijos deklaracija, realizacija: mediana.cpp
double mediana(std::vector<double>);

#endif
```

- Header failai suteikia galimybę visiems naudotis sukurtomis funkcijomis:

```
// main.cpp failas
#include "mediana.h"; // gauti deklaraciją mediana()
int main(){ /* Naudojame mediana() */ }
```

# (v0.2) ir (v0.3) kūrimas (16)

## Header failai: studentas.{h, hh, hpp}

```
// header failas (studentas.h)
#ifndef GUARD_STUDENTAS_H
#define GUARD_STUDENTAS_H

#include <iostream>
#include <string>
#include <vector>
#include "mediana.h"

struct Studentas {
    std::string vardas;
    std::string pavarde;
    double egzaminas;
    std::vector<double> nd;
}; // kabliataškis būtinis

bool compare(const Studentas&, const Studentas&);
bool comparePagalPavarde(const Studentas&, const Studentas&);
bool comparePagalEgza(const Studentas&, const Studentas&);
double galBalas(const Studentas&, double (*) (vector<double>) = mediana);
double galBalas(double, std::vector<double>);
std::istream& readStudent(std::istream&, Studentas&);

// TODO: Funkcijų realizacijos turi būti apibrėžtos studentas.cpp faile

#endif
```

## Viena iš (v0.4) užduočių formuluočių

- Sūrušiuokite (padalinkite) studentus į dvi kategorijas:
  - Studentai, kurių **galutinis balas < 5.0** galėtume vadinti "vargšiukai", "nuskriaustukai" ir pan.
  - Studentai, kurių **galutinis balas >= 5.0** galėtume vadinti "kietiakiai", "galvočiai" ir pan.

# Versijos (v0.4) kūrimas (1)

**Studentų skirstymas į dvi kategorijas:** išlaikiusius 💪 **ir** neišlaikiusius 😢

```
// predikatas patikrinantis ar studentas gavo skolą (neišlaikė)
bool gavoSkola(const Studentas& s,
                 double (*kriterijus)(vector<double>) = mediana) {
    return galBalas(s, kriterijus) < 5.0;
}
```

- Paprasčiausias problemos sprendimas būtu ištirti kiekvieno studento galutinį balą ir pagal jį studentą priskirti (nukopijuoti) į vieną iš dviejų naujų vektorių:
  - "kietujų", t.y., egzaminai išlaikiusių;
  - "minkštujų", t.y., gavusių skolą.

# Versijos (v0.4) kūrimas (2)

**Studentų skirstymas į dvi kategorijas:** kietus **ir** minkštus (**1 versija**)

```
// padalija studentus į du vektorius: kietus ir minkštus (gavusius skolą)
vector<Studentas> skirstykStudentus(vector<Studentas>& studentai) {
    vector<Studentas> kieti, minksti;
    for (vector<Studentas>::size_type i = 0; i != studentai.size(); ++i)
        if (gavoSkola(studentai[i])) // default ND kriterijus - mediana
            minksti.push_back(studentai[i]);
        else
            kieti.push_back(studentai[i]);
    studentai = kieti; // vektoriui 'studentai' priskiriame kietus
    return minksti; // grąžina vektorių iš studentų gavusių skolą
}
```

- Ši realizacija pakankamai sparti, tačiau tokiu būdu kiekvieno studento įrašas yra saugomas dvejuose vektoriuose!

# Versijos (v0.4) kūrimas (3)

**Studentų skirstymas į dvi kategorijas:** kietus **ir** minkštus (**1 versija su foreach iš C+ +11**)

```
// padalija studentus į du vektorius: kietus ir minkštus (gavusius skolą)
vector<Studentas> skirstykStudentus(vector<Studentas>& studentai) {
    vector<Studentas> kieti, minksti;
    for (const auto& stud : studentai)
        if (gavoSkola(stud))
            minksti.push_back(stud);
        else
            kieti.push_back(stud);
    studentai = kieti; // studentai turės tik išlaikiusius
    return minksti; // grąžina studentus gavusius skolą
}
```

# Versijos (v0.4) kūrimas (4)

**Studentų skirstymas į dvi kategorijas:** kietus **ir** minkštus (**2 versija**)

```
// minkštus studentus nukopijuoją į naują vektorių ir ištrina iš seno
vector<Studentas> raskMinkstus(vector<Studentas>& studentai) {
    vector<Studentas> minksti;
    vector<Studentas>::size_type i = 0;
    // invariantas: vektoriaus `studentai` elementai [0, i) yra "kieti"
    while (i != studentai.size()) {
        if (gavoSkola(studentai[i])) {
            minksti.push_back(studentai[i]);
            studentai.erase(studentai.begin() + i); // ištrinti i-ąjį stud.
        } else
            ++i; // pereiti prie kito studento
    }
    return minksti; // grąžina studentus gavusius skolą
}
```

# Elementų šalinimas (1)

## vector.erase() funkcija

- Vektoriaus tipo nario funkcija .erase() pašalina elementą(-us) iš vektoriaus.
- Argumentas nurodo, kurj(-iuos) elementą(-us) pašalinti:  
`studentai.erase(studentai.begin() + i);`
- Kadangi ne visi konteineriai palaiko indeksavimą (operatorius[]),  
todėl .erase() veikia tik per iteratorius, kuriuos palaiko visi konteineriai.

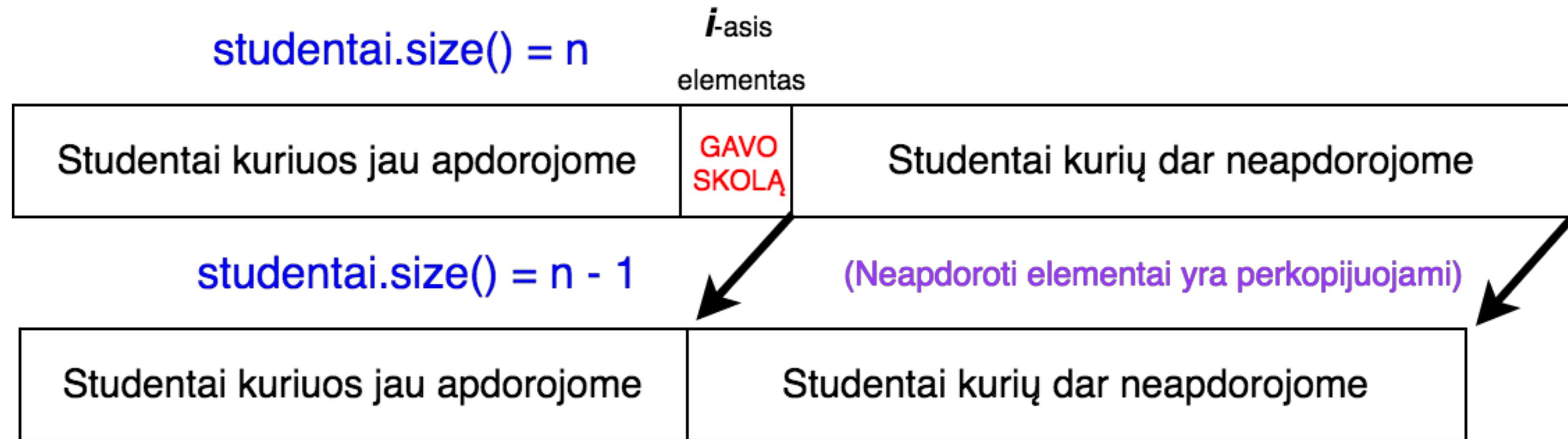
`std::vector::erase`: Removes specified elements from the container.

1) Removes the element at `pos`; 2) Removes the elements in the range `[first; last)`.  
Invalidates iterators and references at or after the point of the erase, including the `end()` iterator.  
<sup>v.erase</sup>

<sup>v.erase</sup> <http://en.cppreference.com/w/cpp/container/vector/erase>

# Elementų šalinimas (2)

`studentai.erase()` veikimo iliustracija



- Po ištrynimo,  $i$ -tasis bus sekantis neapdorotas elementas.

# Klaidinė optimizacija (1)

```
vector<Studentas>::size_type i = 0;
// išsaugome vektoriaus dydį prieš while ciklą
auto size = studentai.size(); // C++11
while (i != size) {
    if (gavoSkola(studentai[i])) {
        minksti.push_back(studentai[i]);
        studentai.erase(studentai.begin() + i);
    } else
        ++i; // pereiti prie kito studento
}
```

- Kai ištriname elementą iš vektoriaus, Jame yra vienu elementu mažiau, todėl dalis nuorodų studentai[i] būtų į neegzistuojančius elementus!

# Nuoseklus vs. atsitiktinis prieinamumas

- Abi studentus skirstančios funkcijos skirstykStudentus() ir raskMinkstus() iteruoja per konteinerių elementus **tik nuosekliai**.
- Tas nėra taip jau akivaizdu, kadangi iteruojame naudodami operator[], t.y., studentai[i], kuris gali kisti ir nenuosekliai.
- Kadangi elementus pasiekiame tik nuosekliai, todėl nėra tikslu naudoti indeksus (`operator[]), kuriuos palaiko ne visi konteineriai.
- Todėl būtų tikslinga funkcijų realizacijoje į tai atsižvelgti ir naudoti C++ standartinės bibliotekos tam skirtas priemones (iteratorius).

# Iteratoriai

Iteratoriai ("rodyklės"):

- Atpažįsta konteinerių ir jo elementų tipą.
- Leidžia naudoti (pasiekti) reikšmes į kuriuos iteratorius nukreiptas.
- Suteikia operacijas iteruoti (judėti) tarp įvairių tipų konteinerio elementų.
- Apriboja galimas operacijas pagal tai, ką konteineris gali efektyviai atlikti.

# Iteratorių tipai

- Kiekvienas standartinis konteineris (pvz. vector) apibrėžia du iteratorių tipus:

```
kontakteinerio-tipas::const_iterator    // read-only  
kontakteinerio-tipas::iterator          //
```

kontakteinerio-tipas apima konteinerio ir jo elementų tipą,  
pvz. vector<Studentas>.

# Iteratoriai vietoje indeksų (1)

Vietoje indeksų:

```
for (vector<Studentas>::size_type i = 0;  
     i != studentai.size(); ++i)  
    cout << studentai[i].vardas << endl;
```

Galime naudoti iteratorius:

```
for (vector<Studentas>::const_iterator it = studentai.begin();  
     it != studentai.end(); ++it) {  
    cout << (*it).vardas << endl;  
}
```

— `studentai.end()` - grąžina iteratorių į pirmą elementą už konteinerio!

# Iteratoriai vietoje indeksų (2)

```
for (vector<Studentas>::const_iterator it = studentai.begin();  
     it != studentai.end(); ++it) {  
    cout << (*it).vardas << endl; // it->vardas  
}
```

- Konteinerio elementus (lvalue) į kuriuos iteratorius it nukreiptas, pasiekiame per dereference operatorių \*.
- Kadangi operatorius . yra viršesnis negu operatorius \*<sup>prec.</sup>, todėl **būtina** apskliausti iteratorių (\*it). **Kas nutiktu jeigu neapskliaustum?**
- Todėl buvo sukurtas operatorius -> kurį naudojant ankstesnis reiškinys: (\*it).vardas yra ekvivalentus: it->vardas.

<sup>prec.</sup> [http://en.cppreference.com/w/cpp/language/operator\\_precedence](http://en.cppreference.com/w/cpp/language/operator_precedence)

# Versijos (v0.4) kūrimas (5)

## Studentų skirstymas į dvi kategorijas - naudojant vector iteratorius

```
vector<Studentas> raskMinkstus(vector<Studentas>& studentai) {
    vector<Studentas> minksti;
    vector<Studentas>::iterator it = studentai.begin();
    while (it != studentai.end()) {
        if (gavoSkola(*it)) {
            minksti.push_back(*it);    // dereference *it kad gauti elementą
            it = studentai.erase(it); // perduodame it, gauname it
        } else
            ++it; // pereiti prie kito studento
    }
    return minksti; // grąžina studentus gavusius skolą
}
```

- Kodėl ne: `vector<Studentas>::const_iterator it = studentai.begin();?`
- Kodėl `it = studentai.erase(it);` o ne tiesiog: `studentai.erase(it);?`

# Klaidinė optimizacija (2)

```
vector<Studentas> raskMinkstus(vector<Studentas>& studentai) {  
    vector<Studentas> minksti;  
    vector<Studentas>::iterator it      = studentai.begin();  
    vector<Studentas>::iterator end_it = studentai.end();  
    while (it != end_it) {  
        /* viskas kaip buvo ankščiau */  
    }  
    return minksti; // grąžina studentus gavusius skolą  
}
```

- Šis **while** ciklas beveik garantuotai "užluš". Kodėl?

# Kito tipo konteineriai

- Naudodami iteratorius, mes pašalinome konteinerių priklausomybę nuo indeksų.
- Poreikis įterpti ar ištrinti elementus iš bet kurią konteinerio vietą yra įprastas, todėl C++ turi tam optimizuotus konteinerius list (sarašas) apibrėžtus <list>.
- List'ų struktūrą sudėtingesnę, todėl jie yra lėtesni nei vektoriai, jei elementai pasiekiamas tik nuosekliai. Taip pat yra neparankūs cache'avimui.
- Sarašai ir vektoriai turi daug bendro, todėl galime programas naudojančias vectorius paprastai transformuoti į programas naudojančias listus.

# Versijos (v0.5) kūrimas (1)

## Studentų skirstymas į dvi kategorijas - naudojant list

```
// reikia: #include <list>
list<Studentas> raskMinkstus(list<Studentas>& studentai) {
    list<Studentas> minksti;
    list<Studentas>::iterator it = studentai.begin();
    while (it != studentai.end()) {
        if (gavoSkola(*it)) {
            minksti.push_back(*it);    // dereference *it kad gauti elementą
            it = studentai.erase(it); // perduodame it, gauname it
        } else
            ++it; // pereiti prie kito studento
    }
    return minksti; // grąžina studentus gavusius skolą
}
```

— **Interfeisas įdentiškas**, tačiau vietoje vector konteinerių naudojame listus!

# vector ir list skirtumai

- Sąrašo (listo) erase() ir ++it realizacijos gerokai skiriasi nuo vektorių.
- List'o erase ir push\_back nesugadina iteratorių į kitus konteinerio elementus.
- Kadangi listai nepalaiko atsitiktinio indeksavimo, todėl negalime naudoti standartinės bibliotekos sort() funkcijos, t.y:

```
// vector tipo konteinerio elementų rūšiavimas
vector<Studentas> studentai;
sort(studentai.begin(), studentai.end(), compare);
```

```
// list tipo konteinerio elementų rūšiavimas
list<Studentas> studentai;
studentai.sort(compare); // compare veikia su Studentas obj.!
```

# **typedef/using generalizavimas**

- Ar naudodamiesi `typedef` (`using`), galime parašyti vieną programos realizaciją, kuri lengvai leistų pasirinkti tarp skirtingų konteinerių tipų?

```
// typedef vector<Studentas> StudentuTipas;
typedef list<Studentas> StudentuTipas;

// C++11 alternatyva:
// using StudentuTipas = list<Studentas>;

// Tuomet
StudentuTipas studentai;
StudentuTipas::iterator it = studentai.begin();
// ... ir t.t. ir pan.
```

- Viskas čiki piki iki tol, kol ...?

# Atminties hierarchinė schema (1)

## Latency Numbers Every Programmer Should Know<sup>latency</sup>

### Latency Comparison Numbers

L1 cache reference	0.5	ns			
Branch mispredict	5	ns			
L2 cache reference	7	ns		14x L1 cache	
Mutex lock/unlock	25	ns			
Main memory reference	100	ns		20x L2 cache, 200x L1 cache	
Compress 1K bytes with Zippy	3,000	ns	3	us	
Send 1K bytes over 1 Gbps network	10,000	ns	10	us	
Read 4K randomly from SSD*	150,000	ns	150	us	~1GB/sec SSD
Read 1 MB sequentially from memory	250,000	ns	250	us	
Round trip within same datacenter	500,000	ns	500	us	
Read 1 MB sequentially from SSD*	1,000,000	ns	1,000	us	~1GB/sec SSD, 4X memory
Disk seek	10,000,000	ns	10,000	us	10 ms 20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000	ns	20,000	us	20 ms 80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000	ns	150,000	us	150 ms

### Notes

----  
1 ns =  $10^{-9}$  seconds

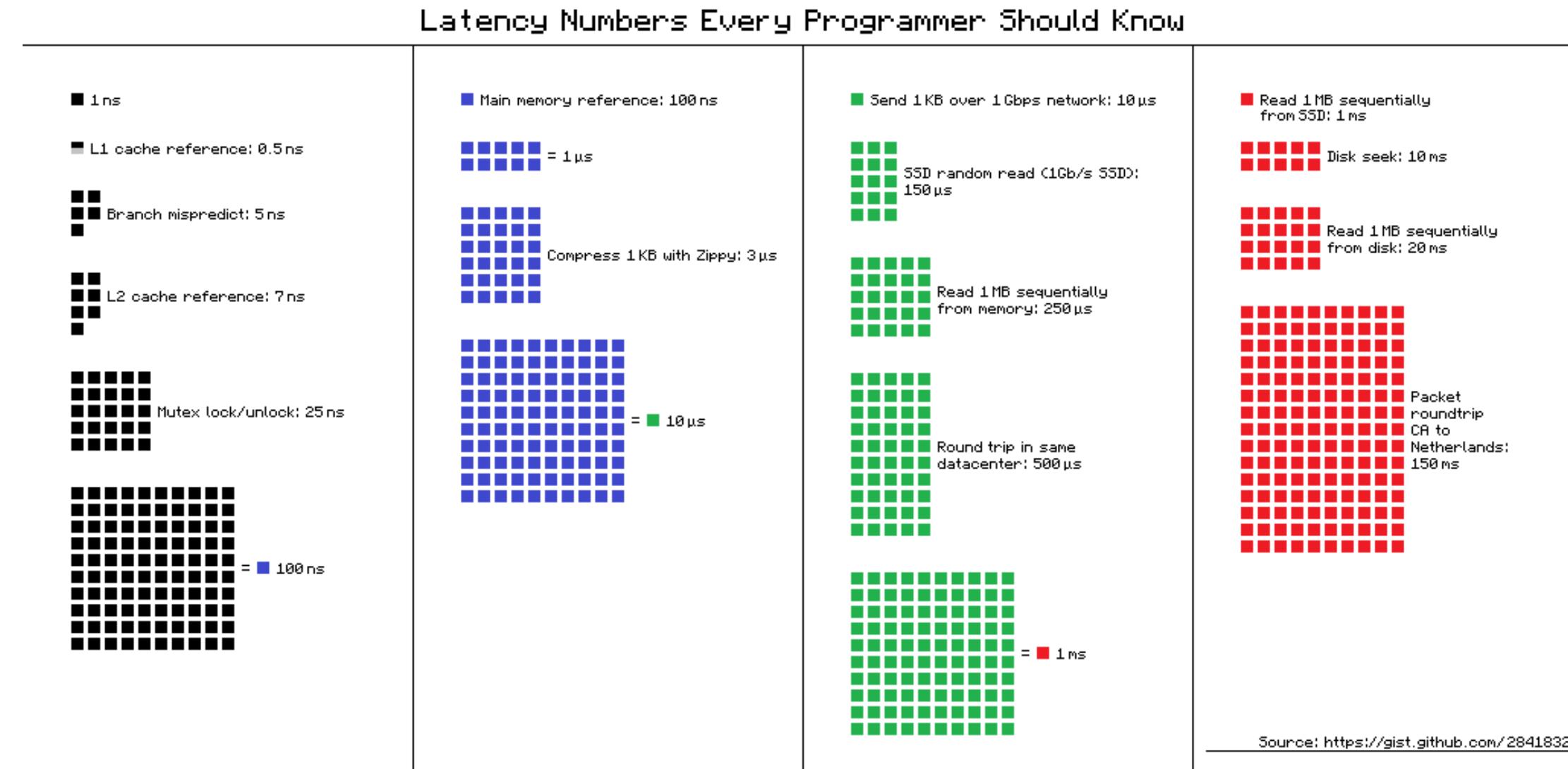
1 us =  $10^{-6}$  seconds = 1,000 ns

1 ms =  $10^{-3}$  seconds = 1,000 us = 1,000,000 ns

latency <https://gist.github.com/jboner/2841832>

# Atminties hierarchinė schema (1)

## Latency Numbers Every Programmer Should Know



# Versijos (v1.0) kūrimas (1)

## Algoritmiskas studentų skirstymas į dvi kategorijas (1 variantas)

```
// predikatas patikrinantis ar studentas negavo skolos (išlaikė)
bool negavoSkolos(const Studentas& s,
                    double (*kriterijus)(vector<double>) = mediana) {
    return !gavoSkola(s, kriterijus);
}
```

```
// Skirstome į dvi kategorijas panaudojant STL algoritmus
vector<Studentas> raskMinkstus(vector<Studentas>& studentai) {
    vector<Studentas> minksti;
    remove_copy_if(studentai.begin(), studentai.end(),
                   back_inserter(minksti), negavoSkolos);
    studentai.erase(remove_if(studentai.begin(), studentai.end(),
                             gavoSkola), studentai.end());
    return minksti; // grąžina studentus gavusius skolą
}
```

# **std::remove\_copy\_if** funkcija

```
template< class InputIt, class OutputIt, class UnaryPredicate >
OutputIt remove_copy_if( InputIt first, InputIt last, OutputIt d_first,
                        UnaryPredicate p );
```

Copies elements from the range **[first, last)**, to another range beginning at **d\_first**, **omitting the elements which satisfy specific criteria**. Source and destination ranges cannot overlap. Ignores all elements for which predicate **p** returns **true**.

<sup>rcif</sup> [https://en.cppreference.com/w/cpp/algorithm/remove\\_copy](https://en.cppreference.com/w/cpp/algorithm/remove_copy)

# Versijos (v1.0) kūrimas (2)

## Algoritmiškas studentų skirstymas į dvi kategorijas (1 variantas)

- Naudojant std::remove\_copy\_if algoritma, predikatas nusako kokius elementus "ištinti".
- Šiame kontekste studento "ištrynimas" yra suprantamas jo **nenukopijavimu** į naują vektorių - minksti.
- Todėl naudojant negavoSkolos() predikatą į minksti vektorių nukopijuojami visi, kurie netenkina predikato, t.y., tik **skolą gavę studentai**:

```
remove_copy_if(studentai.begin(), studentai.end(),
               back_inserter(minksti), negavoSkolos);
```

# Versijos (v1.0) kūrimas (3)

## Algoritmiškas studentų skirstymas į dvi kategorijas (1 variantas)

- Tuomet naudojant std::remove\_if algoritmą "ištriname" elementus (studentus), kurie **tenkina** predikatai gavoSkola().
- Šiame kontekste studento "ištrynimas" ir vėl yra sąlyginis.
- Iš tiesų std::remove\_if nieko neištrina, o nukopijuojat visus studentus, kurie netenkina predikato, t.y. **negavusius skolos**.
- Tačiau kaip, nes juk naudojamas tik vienas konteineris?

```
studentai.erase(remove_if(studentai.begin(), studentai.end(),
                         gavoSkola), studentai.end());
```

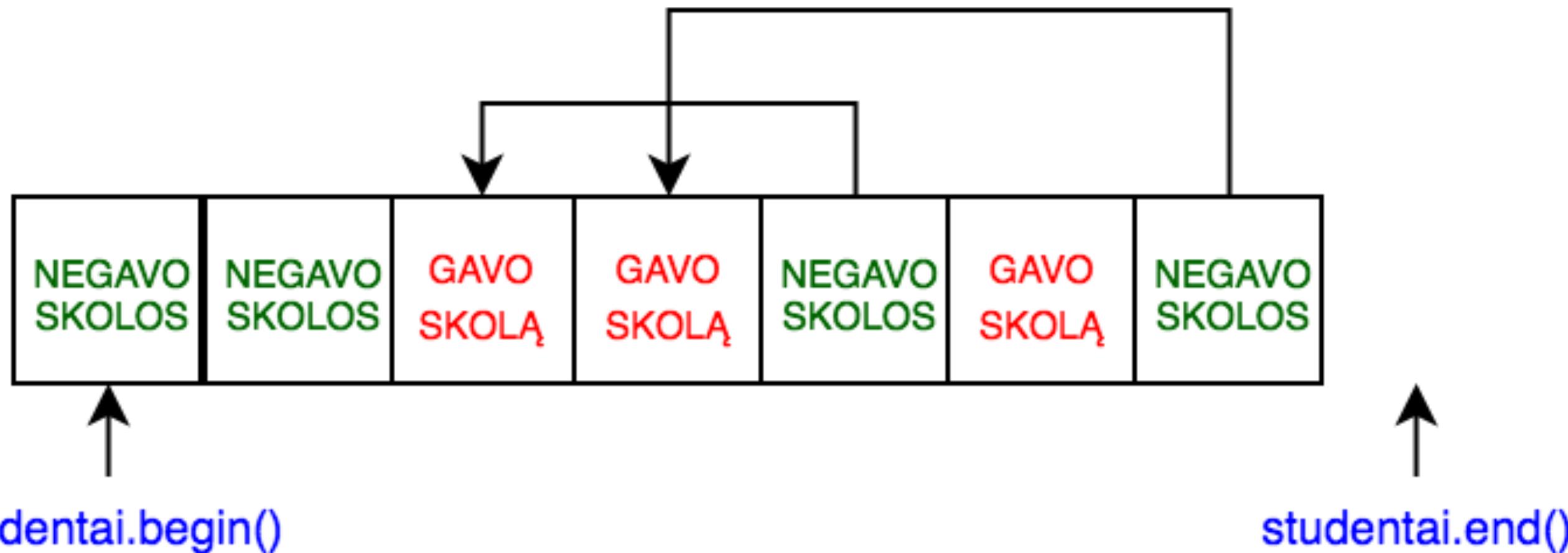
# **std::remove\_if iliustracija (1)**

- Tarkime, kad pradžioje studentai vektoriuje yra 7 studentai, tokie, kad iš jų galutinių balų gauname:



## std::remove\_if iliustracija (2)

- Algoritmas std::remove\_if palieka pirmus du studentus nepaliestus, tačiau kitus du "ištrina", t.y. jų vietą išsaugo kaip galimą naudoti kitiems skolos negavusiems studentams, t.y. 5-ajam ir 7-ajam atitinkamai:



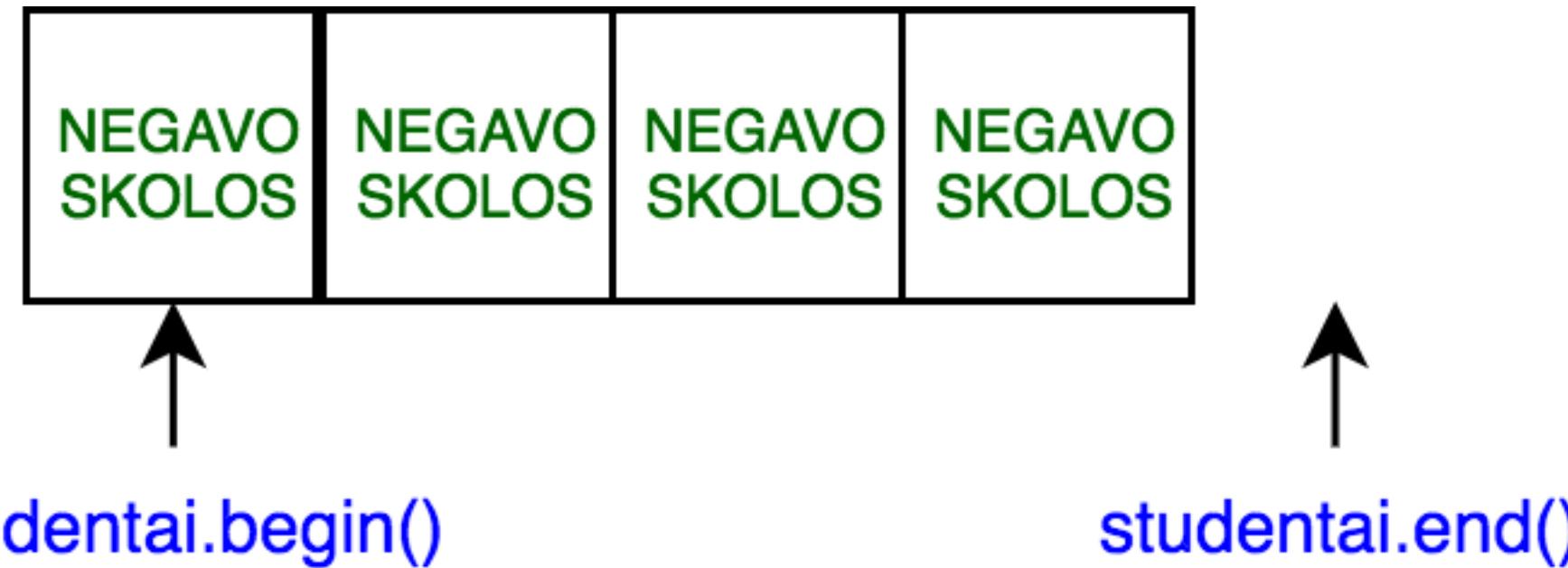
# `std::remove_if` iliustracija (3)

- `std::remove_if` grąžina iteratorių į elementą esantį viena pozicija toliau už paskutinij elementą, kurio "neištrynė":



## `std::erase` ir `std::remove_if`

- Galiausiai mums reikia iš tiesų **ištrinti** nereikalingus studentus iš studentai vektoriaus esančius tarp `remove_if()` ir `studentai.end()` iteratorių, panaudojant `std::erase`:



# Versijos (v1.0) kūrimas (4)

## Algoritmiškas studentų skirstymas į dvi kategorijas (1 variantas)

- Šio 1-ojo varianto akivaizdus neefektyvumas yra tame, kad ši realizacija kiekvieną studentą **du kartus** patikrina ar jis gavo skolą!
- Dar daugiau, šis variantas reikalauja papildomai kompiuterio atminties minksti vektoriui saugoti, tol kol .erase() neįvykdytą:

```
vector<Studentas> raskMinkstus(vector<Studentas>& studentai) {  
    vector<Studentas> minksti;  
    remove_copy_if(studentai.begin(), studentai.end(),  
                  back_inserter(minksti), negavoSkolos);  
    studentai.erase(remove_if(studentai.begin()), studentai.end(),  
                  gavoSkola), studentai.end());  
    return minksti; // grąžina studentus gavusius skolą  
}
```

## **std::partition ir std::stable\_partition**

- Nors std nėra (?) algoritmo, kuris tiesiogiai darytų tai, ko mums reikia, tačiau yra (bent) vienas, kurį galime lengvai adaptuoti: jis reorganizuojas elementų seką taip, kad visi elementai kurie tenkina predikatą eitų prieš tuos, kurie netenkina.
- Yra dvi versijos: std::partition ir std::stable\_partition.
- Skirtumas tas, kad std::partition gali sukeisti elementų tvarką, o std::stable\_partition išsaugo juos ta pačia tvarka.
- Jeigu studentai yra surūšiuoti pagal pavardę ir mes norime juos išlaikyti surūšiuotus tarp kategorijų, tuomet naudojame: std::stable\_partition.

# Versijos (v1.0) kūrimas (5)

## Algoritmiskas studentų skirstymas į dvi kategorijas (2 variantas)

```
vector<Studentas> raskMinkstus(vector<Studentas>& studentai) {  
    vector<Studentas>::iterator it =  
        stable_partition(studentai.begin(), studentai.end(), negavoSkolos);  
    vector<Studentas> minksti(it, studentai.end());  
    studentai.erase(it, studentai.end());  
    return minksti; // gražina studentus gavusius skolą  
}
```

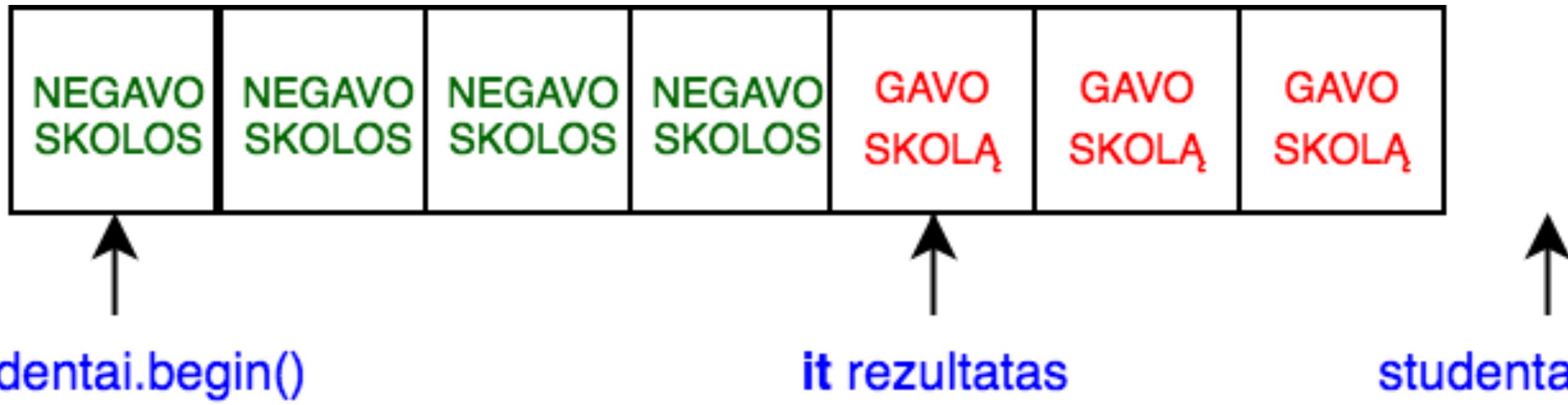
# stable\_partition iliustr. (1)

- Tarkime, kad pradžioje studentai vektoriuje yra 7 studentai, tokie, kad iš jų galutinių balų gauname:



## stable\_partition iliustr. (2)

— Įvykdžius stable\_partition algoritmą, gauname:



# **Versijos (v1.0) kūrimas (6)**

## **Algoritmiškas studentų skirstymas į dvi kategorijas (2 variantas)**

```
vector<Studentas> raskMinkstus(vector<Studentas>& studentai) {  
    vector<Studentas>::iterator it =  
        stable_partition(studentai.begin(), studentai.end(), negavoSkolos);  
    vector<Studentas> minksti(it, studentai.end());  
    studentai.erase(it, studentai.end());  
    return minksti; // grąžina studentus gavusius skolą  
}
```

- Tuomet vektorius minksti sukonstruojamas perkopijavus neišlaikiusius studentus iš intervalo - [it, studentai.end())
- Galiausiai ištriname neišlaikiusius iš studentai vektoriaus.

# Klausimai!?

