

# Objektinis Programavimas

## 2-oji praktinė užduotis



# Duomenų apdorojimas

## Turinys

- 👉 Užduoties formuluotė
- 👉 Reikalavimai versijoms ir jų atlikimo terminai
- 👉 Versijos (v0.1) kūrimas
- 👉 Versijų (v0.2) ir (v0.3) kūrimas

# Praktinės užduoties formuluotė

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

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

# Reikalavimai (v0.1) versijai (1)

- ☞ Pagal užduoties reikalavimus realizuokite programą ir atspausdinkite ekrane aktualią informaciją:
  - ☞ studento vardą ir pavardę,
  - ☞ namų darbų ir egzamino rezultatus
  - ☞ bei galutinį balą **dviejų skaičių po kablelio tikslumu**
- ☞ Papildykite programą, kad vietoj **vidurkio** būtų galima naudoti **medianą**.

# Reikalavimai (v0.1) versijai (2)

- ☞ 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 užduotį realizuokite dviem būdais:
  - ☞ naudojant C masyvus.
  - ☞ naudojant `vector` ar kito tipo konteinerių.
- ☞ Papildykite, kad būtų galimybė, jog mokinio gautieji balai už namų darbus bei egzaminą būtų generuojami atsitiktinai.

# Reikalavimai (v0.2) versijai (1)

**Terminas: 2018-02-27**

- ☞ Papildykite (v0.1) taip, kad būtų galima duomenis ne tik įvesti bet ir nuskaityti iš failų, t.y., sukurkite ir užpildykite failą **kursiokai.txt**, kurio (pleriminari) struktūra:

Pavardė	Vardas	ND1	ND2	ND3	ND4	ND5	Egzaminas
Pocius	Paulius	8	9	10	6	10	9
Makevičius	Augustinas	7	10	8	5	4	6
...							

# Reikalavimai (v0.2) versijai (2)

**Terminas: 2018-02-27**

- 👉 Papildykite programą taip, nuskaičiuos duomenis iš failo, galutinis rezultatas (output'as) pleriminariai atrodytu taip:

Pavardė	Vardas	Gal.-vidurkis	Gal.-mediana
Makevičius	Augustinas	x . xx	y . yy
Pocius	Paulius	z . zz	q . qq
...			

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

# Reikalavimai (v0.3) versijai (1)

**Terminas: 2018-03-06**

- 👉 Atlikite versijos (v0.2) kodo reorganizavimą (refactoring'ą):
  - 👉 Kur tikslina, programoje naudokite (jeigu dar nenaudojote) struct'ūras
  - 👉 Funkcijas, naujus duomenų tipus (struct'ūras) perkelkite į antraštinius (**header** (\*.h)) failus, t.y. tokiu būdu turėtumete projekte turęti kelis \*.cpp failus, kaip ir kelis \*.h failus.

# Reikalavimai (v0.3) versijai (2)

**Terminas: 2018-03-06**

👉 Panaudokite išimčių valdymą (**Exception Handling**):

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

**Kam viso to reikia?** Kas atsitiks, jei failas, kuri bandote atidaryti neegzistuoja; arba bandote gauti masyvo elementą, kurio nėra?

# Reikalavimai (v0.4) versijai (1)

**Terminas: 2018-03-13**

- 👉 Programos veikimo greičio (spartos) analizė:
- 👉 Sūrušiuokite (padalinkite) studentus į dvi kategorijas:
- 👉 Studentai, kurie **surinko < 60%** už namų darbų užduotis ("vargšiukai", "nuskriaustukai" ir pan.)
- 👉 Studentai, kurie **surinko >= 60%** ("kietiakai", "galvočiai", "vizunčikai" ir pan.) ir buvo prileisti prie egzamino.

# Reikalavimai versijai (v0.5)

**Terminas: 2018-03-20**

- 👉 Išmatuokite programos veikimo spartą (be failų generavimo, nes visais atvejais **privatote** naudoti tuos pačius sugeneruotus studentų duomenų failus) priklausomai nuo naudojamo vieno iš trijų konteinerių:
  - 👉 vector
  - 👉 list
  - 👉 deque

T.y., jeigu Jūs susikurėte struktūrą **Studentai** (ar kaip jūs ją pavadinote) ir iki šiol naudojote `std::vector<Studentai>`, tai turite ištirti ar pasikeistų ir kaip pasikeistų

# Realiujų skaičiu tikslumas

- ☞ <ioomanip> antraštės (header) faile deklaruotas manipulatorius `setprecision`, kuris leidžia kontroliuoti kiek reikšmingų skaitmenų išvedima informacija naudotu.
- ☞ Kai naudojame `endl`, kuris taip pat yra manipulatorius, mums nereikia ištraukti <ioomanip> antraštės. **Kodėl?**

# std::setprecision() pVZ.<sup>1</sup>

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <limits>
int main() {
    const long double pi = std::acos(-1.L);
    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>

# (v0.1) kūrimas (1)

## **Rezultatų spausdinimas 2 skaičių po kablelio tikslumu**

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

# 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).

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

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

**VS.**

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

---

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

# while (cin >> x) reiškinio analizė

Norint nuskaityti reikšmę į **x** ir patikrinti ar pavyko:

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

Tai ekvivalentu:

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

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

# Kada skaitymas iš `std::cin` nesėkmingas?

- 👉 Mes galėjome pasiekti įvesties failo pabaigą.
- 👉 Galbūt susidūrēme su įvestimi, nesuderinama su kintamojo tipu, kurį bandome perskaityti, pvz., taip gali atsitikti, jei mes bandome perskaityti `int` bet reikšmė yra nėra skaičius.
- 👉 Sistemoje gali būti aptikta įvesties įrangos gedimas.

Visais šiaisiai atvejais: naudojant salygoje `cin` reikšmę bus `false`.

# Srauto (stream) būsenos (1)

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

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

# 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.

# (v0.1) kūrimas (2)

## **Nežinomo skaičiaus namų darbų (nd) nuskaitymas**

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

# (v0.1) kūrimas (3)

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

```
// Aukščiau: using std::vector
vector<double> nd; // vektorius su namų darbų rezultatais
double x;           // kintamasis i kuri nuskaityti
// invariantas: `nd' kaupia visus iki šiol nuskaitytus ND
while (cin >> x) {
    nd.push_back(x); // Pridedame nuskaitytą rezultatą
}
```

# (v0.1) kūrimas (4)

## 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;
}
```

👉 Kodėl `return 1;` ?

# Tipų apibrėžimas naudojant **typedef** ir **using** keywords'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>

# typedef ir using pavyzdžiai

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

# Medianos radimo procedūra

- ☞ Norėdami apskaičiuoti medianą, privalome:
  - ☞ Išsaugoti nuskaitytas reikšmes, nežinant iš anksto, kiek reikšmių bus.
  - ☞ Kai visos reikšmės nuskaitytos - išrūšiuoti jas.
  - ☞ Gauti vidutinę(-es) reišmę(-es) efektyviai.

# (v0.1) kūrimas (5)

## Medianos radimas

```
// surūšiuojame ND, naudojant `sort` iš <algorithm>
sort(nd.begin(), nd.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.

# Didelių programų struktūra

## **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.

# (v0.2) ir (v0.3) kūrimas (1)

## 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.

# Išimtys (exceptions)

- 👉 Kai vektorius yra tuščias, paleidžiame (**throw**) išimti (**exception**).
- 👉 Kai programa paleidžia išimti, 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**.
- 👉 Kai mes sukuriame `std::domain_error` išimti, o `string/const char*` informuojame, kas blogo įvyko. Ši informacija vėliau gali būti panaudota, pvz. diagnostiniame pranešime.

# (v0.2) ir (v0.3) kūrimas (2)

## Funkcija vidurkio apskaičiavimui

```
// Apskaičiuojame vektoriaus elementų vidurki
// Saugiau būtų: `double vidurkis(cons 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 nekopijuoja 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 praeitį 😱

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

```
int x = 10;
const int y = 20;
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 i "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.
- ☞ Iprastai 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`.

# Ivairūs int \* const variantai

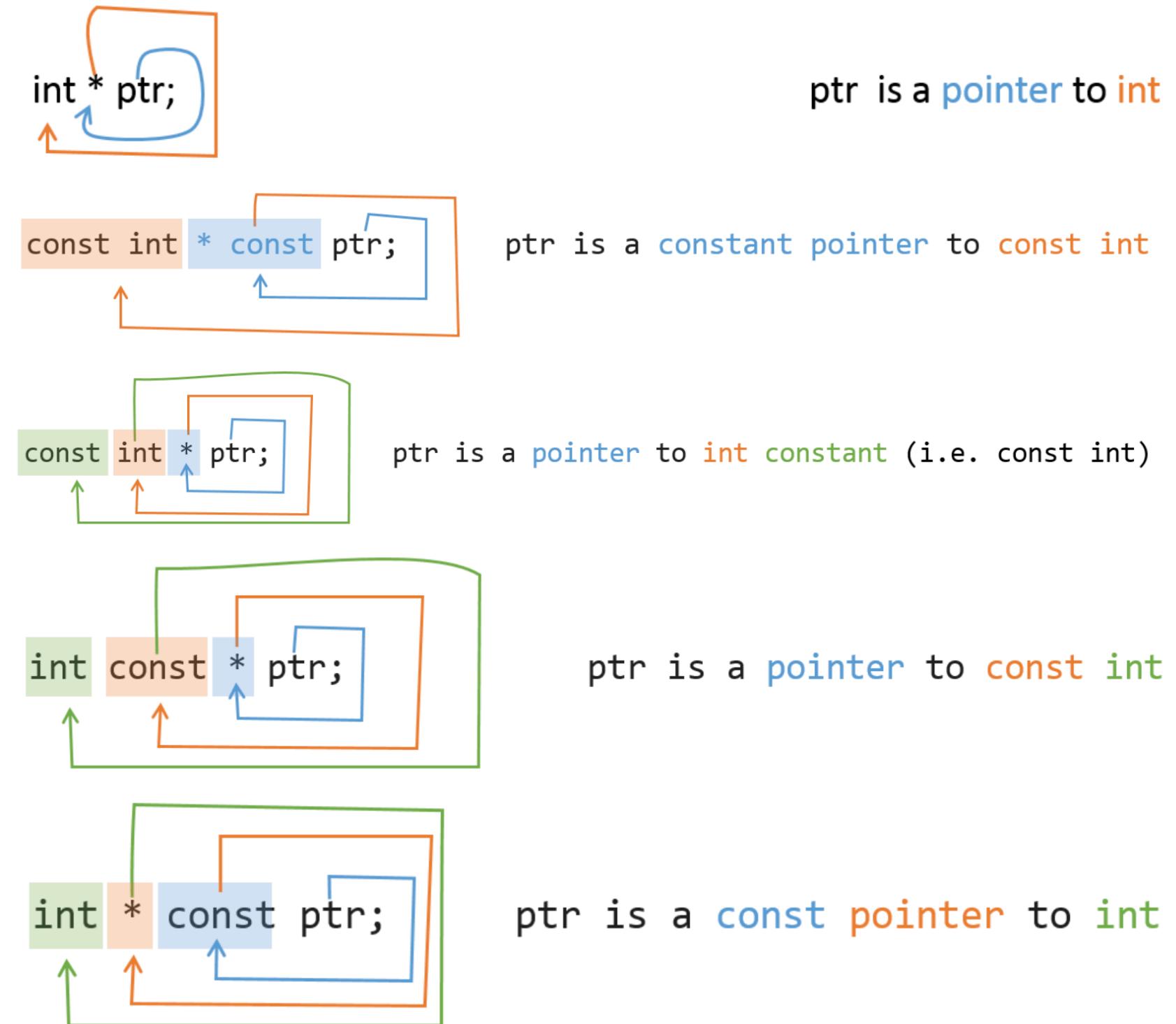
- ☞ Skaitome "žydiškai" (parempta **Clockwise/Spiral Rule**)<sup>4</sup>

int * ptr	- ptr is a pointer to int
const int * ptr	- ptr is a pointer to const int
int const * ptr	- ptr is a pointer to const int
int * const ptr	- ptr is a const pointer to int
int const * const ptr	- ptr is a const pointer to const int

- ☞ 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>



# Rodykles 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>”

---

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

# Rodyklės į funkcijas (2)

- ☞ **Tikslas:** patobulinti galBalas() funkcija, 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 vidurki 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)
}
```

☞ Iš šia funkciją kreiptis galime vienu iš dviejų būdu:

```
galBalas(8.0, nd);           // naudoja medianą (default) ND įverčiui
galBalas(8.0, nd, vidurkis); // naudoja vidurki 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 veiktu 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 reiktu:  
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:**

```
/* PRIDÉTI: naudojamus `using`; reikiamus <header> failus; bei funkcijų:
 * readNd(), mediana (vector<double>), galBalas(double, double),
 * galBalas(double, const vector<double>&) realizacijas */
int main() {
    /* PRIDÉTI: realizaciją nuskaitytį vardą, pavardę, egzamino rezultatai */
    vector<double> nd;
    readNd(cin, nd); // nuskaityti ND
    try { // apskaičiuoti ir atspausdinti galutinių balų (jei īmanoma)
        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 throw'inti?

☞ Reikia apgalvoti, kas gali paleisti (throw'inti) išimti. Panagrinėkite tokį pvz.:

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

- ☞ Jei galBalas() paleidžia išimti, 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
Pocius	Paulius	8	9	10	6	10	9
Makevičius	Augustinas	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 būtinas!
```

☞ 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**

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

// `Studentas` objektui apskaičiuoti galutinių balų 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, **predikata**:

```
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 tuščios vietos
        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 galutini balus pagal du kriterijus (ND iverčiams)
            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);
        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
    }
}
```

# (v0.2) ir (v0.3) kūrimas (14)

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

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

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

## **Source ir header failai: mediana.{h, hh,.hpp}**

👉 Header failai suteikia galimybę visiems naudotis sukurtomis funkcijomis:

```
#include "mediana.h"; // ištrau  
int main()/*...*/
```

👉 Header failuose turi būti naudojami tik **būtini** elementai:

```
// header failas (mediana.h)  
#ifndef GUARD_MEDIANA_H  
#define GUARD_MEDIANA_H  
  
#include <vector>  
double mediana(std::vector<double>);  
  
#endif
```

# (v0.2) ir (v0.3) kūrimas (16)

## **Source ir 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 būtinės

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
```

# (v0.4) 10. užduoties formuluotė

**10.** Surūšiuokite (padalinkite) studentus į dvi kategorijas:

- 👉 Studentai, kurie surinko  $< 60\%$  už namų darbų užduotis ("vargšiukai", "nuskriaustukai" ir pan.)
- 👉 Studentai, kurie surinko  $\geq 60\%$  už namų darbų užduotis ("kietiakiai", "galvočiai" ir pan.) ir buvo prileisti prie egzamino.

**P.s. Logiškesnė užduoties formuluotė būtų studentus dalinti į dvi kategorijas (išlaikiusius ir neišlaikiusius) pagal galutinį balą, o ne namų darbų ivertį.** Tuomet, kieno galutinis balas  $< 5.0$  ("vargšiukai"), o kieno  $\geq 5.0$  ("kietiakai"). Jeigu jau realizavote pagal namų darbus - palikite kaip yra, tačiau jeigu dar nedarėte - tuomet rūšiuokite pagal galutinį balą.

# (v0.4)-(v1.0) 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 būtų ištirti kiekvieno studento galutinių balų ir pagal jų studentą priskirti (nukopijuoti) į vieną iš dviejų naujų vektorių:
  - 👉 "kietųjų", t.y., egzaminą išlaikiusių;
  - 👉 "minkštųjų", t.y., gavusių skolą.

# (V0.4)-(V.1.0) 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!

# (v0.4)-(v1.0) 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; // gražina studentus gavusius skolą
}
```

# (v0.4)-(v1.0) kūrimas (4)

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

```
// minkštus studentus nukopijuojama į naujį 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, kurį(-iuos) elementą(-us) pašalinti:  
`studentai.erase(studentai.begin() + i).` **Ar visada šitas veikia?**
- ☞ 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.

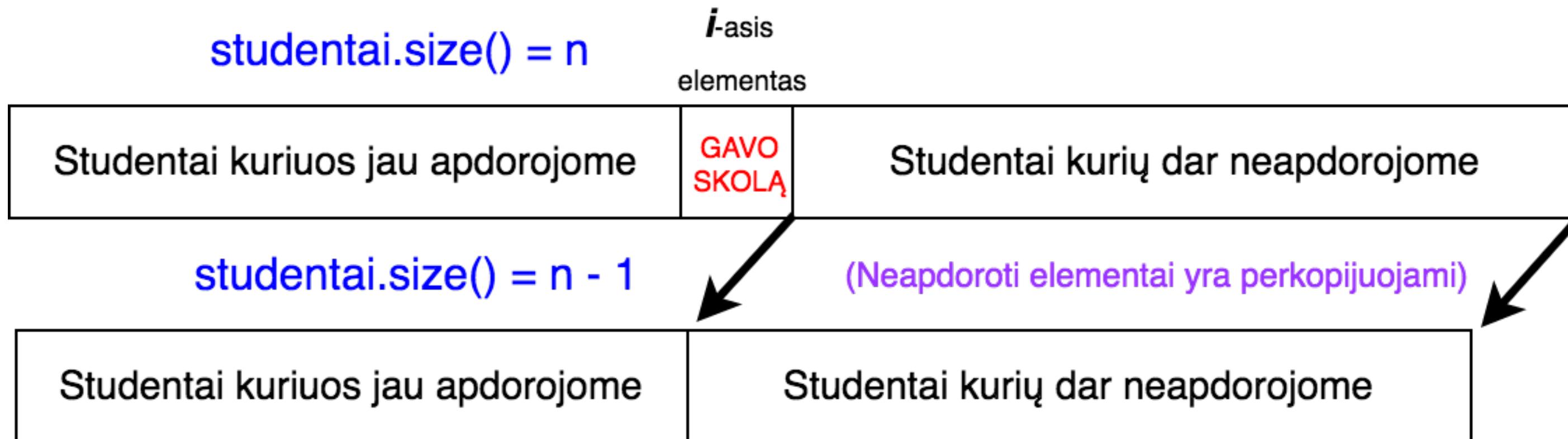
1) Removes the element at pos; 2) Removes the elements in the range [first; last).  
Invalidates iterators and references at or after the point of the erase, including the

---

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

# Elementų šalinimas (2)

## `studentai.erase()` veikimo illustracija



👉 Po ištrynimo, *i*-tasis bus sekantis neapdorotas elementas.

# Klaidinga 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, tame yra vienu elementu mažiau, todėl dalis nuorodų `studentai [ i ]` būtų į neegzistuojančius elementus!

# Nuoseklus vs. atsitiktinis prieinamumas

- ☞ Abi studentus skirtančios 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 tikslø naudoti indeksus (`operator[]`), kuriuos palaiko ne visi konteineriai.
- ☞ Todėl būtū 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 iš kurias 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:

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

kontakteinero-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 nutiktų jeigu neapskliaustumė?**
- ☞ Todėl įvestas operatorius → su kuriuo `(*it).vardas` ekvivalentu `it->vardas`.

---

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

# (V0.4)-(V1.0) 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>::cons_iterator it = studentai.begin();`?
- 👉 Kodėl `it = studentai.erase(it);` o ne tiesiog `studentai.erase(it);`?

# Klaidinga 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; // gražina 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 kuriaj konteinerio vietą yra įprastas, todėl C++ turi tam optimizuotus konteinerius `list` (sarašas) apibrėžtus `<list>`.
- ☞ Sarašų struktūrą sudėtingesnę, todėl jie yra lėtesni nei vektoriai, jei elementai pasiekiamas tik nuosekliai. Taip pat yra neparankūs cacheavimui.
- ☞ Sarašai ir vektoriai turi daug bendro, todėl galime programas naudojančias `vectorius` paprastai transformuoti iš programas naudojančias `listus`.

# (v0.4)-(v1.0) kūrimas (6)

## **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 identiškas, tačiau vietoje `vector` konteinerių naudojame `list`.

# 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 `list`ai 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 ant Studentas obj.
```

# typedef/using generalizavimas

- 👉 Ar naudodamiesi `typedef` (`using`), galime parašyti vieną programos realizaciją, kuri lengvai leistų pasirinkti tarp skirtinių 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	1 ms	~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

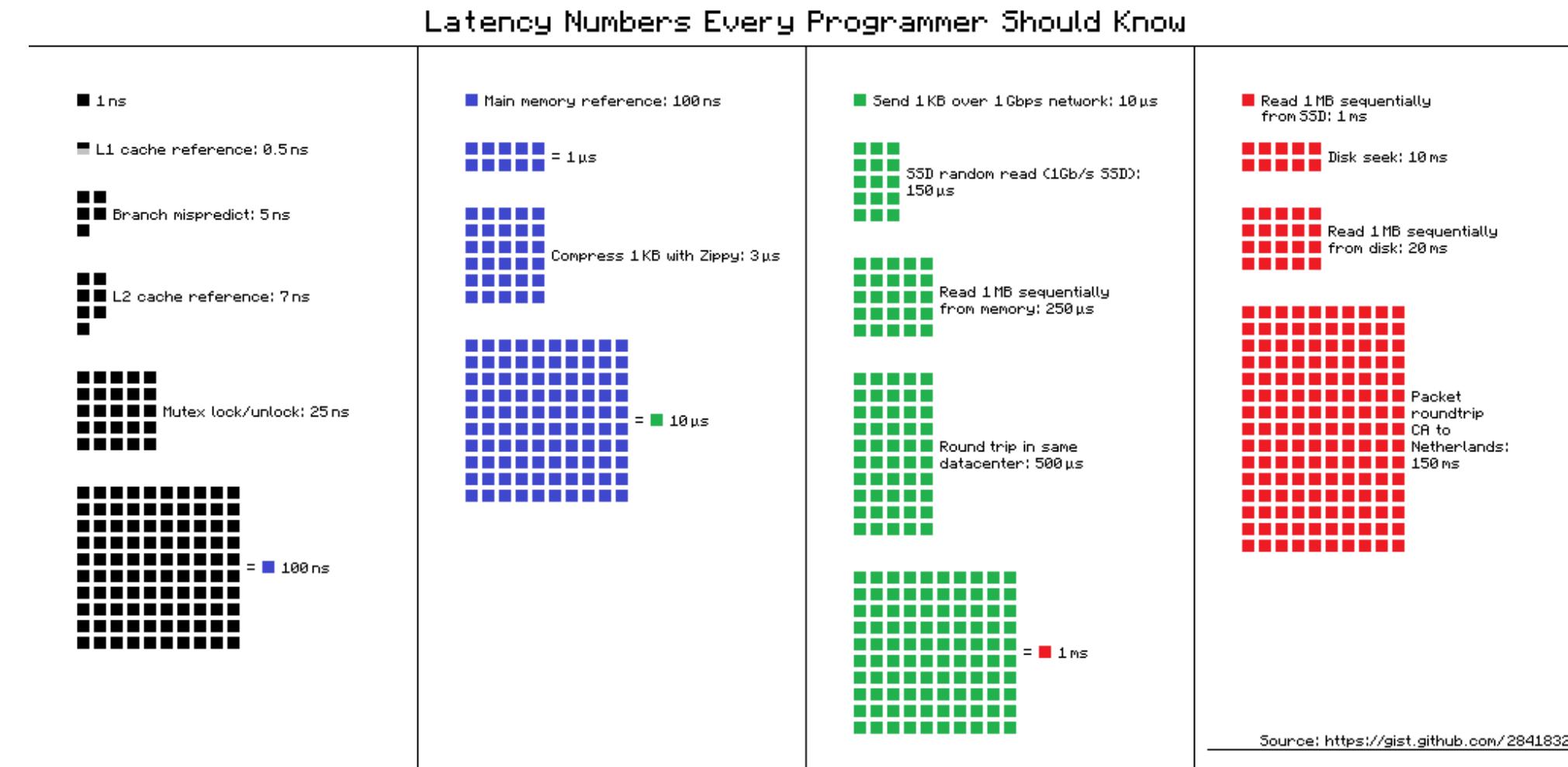
1 ms =  $10^{-3}$  seconds = 1,000 us = 1,000,000 ns

---

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

# Atminties hierarchinė schema (1)

## Latency Numbers Every Programmer Should Know



# (V0.4)-(V1.0) kūrimas (7)

## **Algoritmiškas 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);
}

// Skirstime į dvi kategorijas naudojant 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

```
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)

# (v0.4)-(v1.0) kūrimas (8)

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

- ☞ Naudojant `std::remove_copy_if` algoritma, predikatas nusako kokius elementus "ištrinti".
- ☞ 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);
```

# (v0.4)-(v1.0) kūrimas (9)

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

- ☞ Tuomet naudojant `std::remove_if` algoritma "ištriname" elementus (studentus), kurie **tenkina** predikatą `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());
```

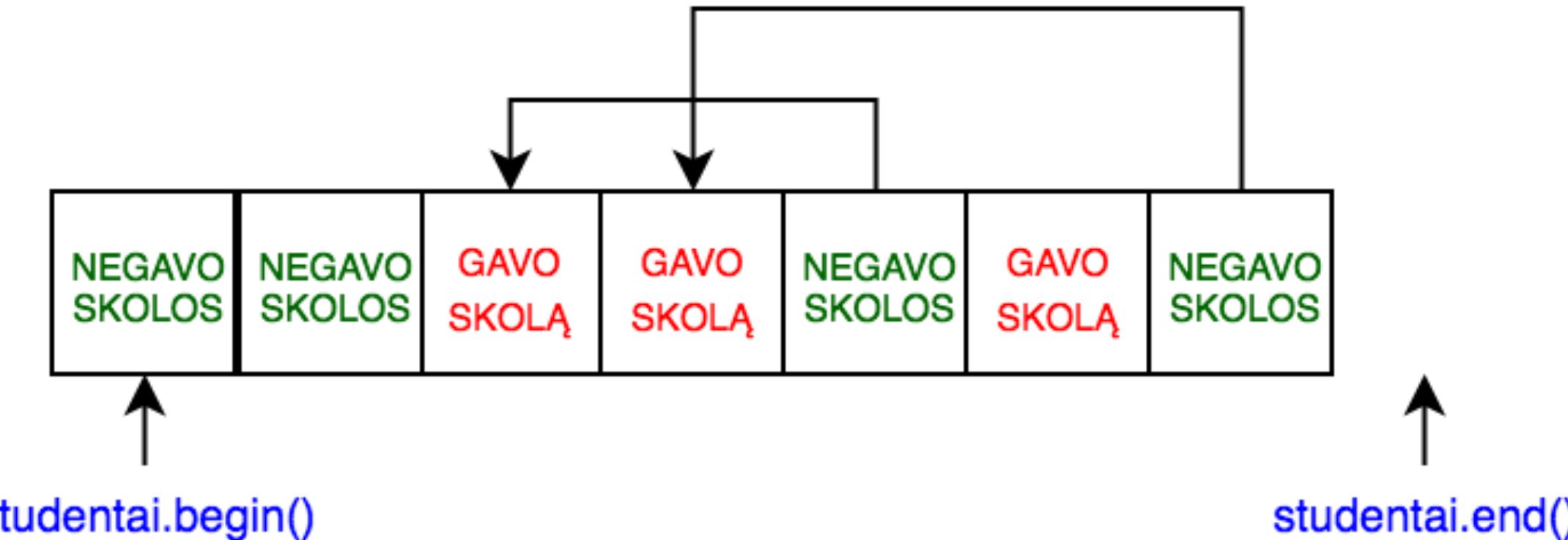
# std::remove\_if iliustracija (1)

- ☞ 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:



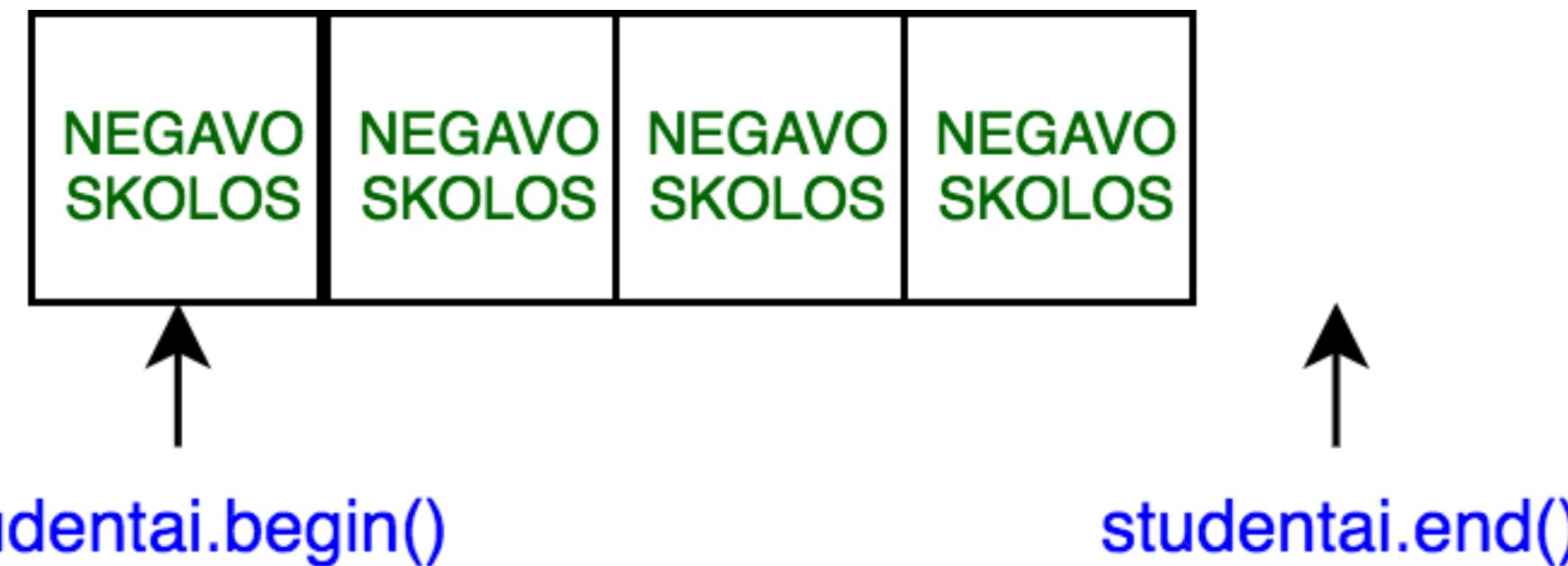
# `std::remove_if` iliustracija (3)

- ☞ `std::remove_if` gražina iteratorių į elementą esantį viena pozicija toliau už paskutinį elementą, kurio "neištrynė":



# std::erase ir std::remove\_if

- ☞ Galiausiai mums reikia **ištrinti** nereikalingus studentus iš studentai vektoriaus esančius tarp remove\_if() ir studentai.end() iteratorių, panaudojant std::erase:



# (v0.4)-(v1.0) kūrimas (10)

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

- ☞ Pirmosios versijos akivaizdus neefektyvumas yra tame, kad ši realizacija kiekvieną studentą **du kartus** patikrina ar jis gavo skolą!
- ☞ Kaip **@BlackDude22** pastebėjo, šis variantas reikalauja papildomai atminties minksti vektoriui saugoti, tol kol .erase() neįvykdyta:

```
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, kuri galime lengvai adaptuoti: jis reorganizuja elementų seką taip, kad visi elementai kurie tenkina predikatą eitų prieš tuos, kurie netenkina.
- 👉 Yra dvi šio algoritmo versijos: `std::partition` ir `std::stable_partition`.
- 👉 Skirtumas tas, kad `std::partition` gali pertvarkyti kiekvienos kategorijos elementus, 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`.

# (V0.4)-(V1.0) kūrimas (11)

## **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; // gražina studentus gavusius skolą
}
```

# stable\_partition iliustr. (1)

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



`studentai.begin()`

`studentai.end()`

# stable\_partition iliustr. (2)

☞ Įvykdžius stable\_partition algoritma, gauname:



# (v0.4)-(v1.0) kūrimas (12)

## **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; // gražina studentus gavusius skolą
}
```

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

# Klausimai!

# !?

