

# Objektinis Programavimas

**std::vector, std::array ir std::string konteineriai**



# Turinys

- std::vector vs C-array
- Dinaminė atmintis C priemonėmis: new ir delete
- Automatinis STL konteinerių augimas
- Keturios svarbios STL konteinerių funkcijos
  - - size()
  - capacity()
  - resize()
  - reserve()
- C++ konteinerių naudojimas C API
- std::array()



## Paskaitos tikslas

- Išmokti efektyviau naudoti STL konteinerius: `std::vector`, `std::array` ir `std::string` vietoj tradicinių C-array tipo masyvų.
- Skaidrėse pateikiamos nuorodos į atitinkamus **Effective STL** knygos skyrius, kuriuose galima rasti daugiau informacijos konkrečiai temai.

# Effective STL

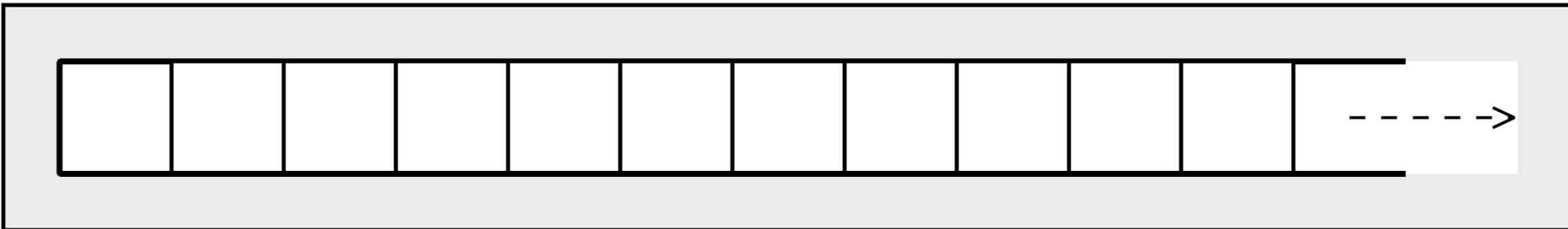
50 Specific Ways to Improve Your Use of the Standard Template Library

Scott Meyers

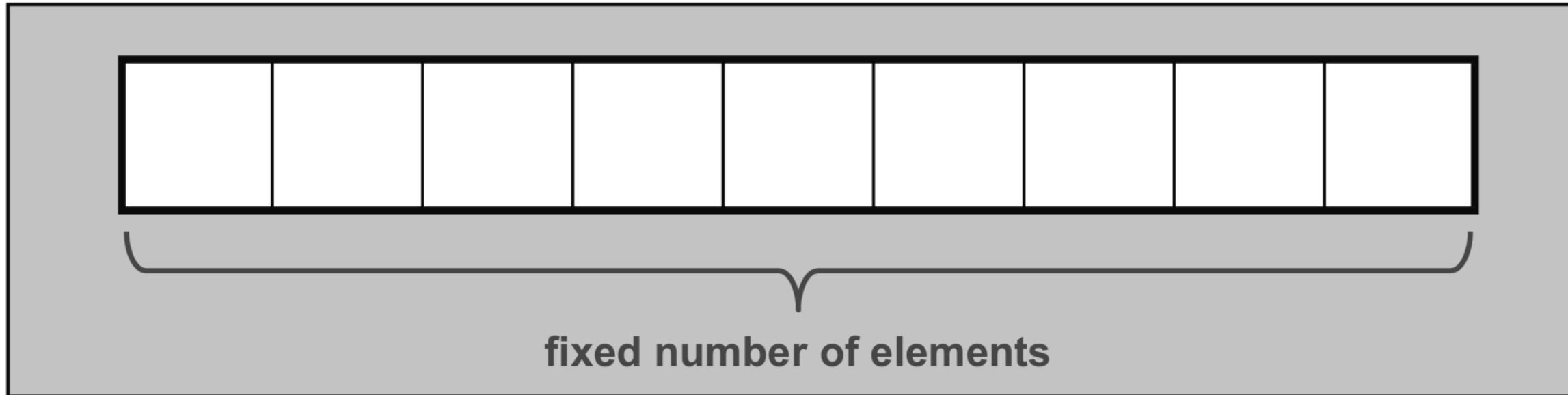


# **std::vector vs C-array**

- vector struktūra:



- C-array (o taip pat ir nuo C++11: `std::array`) struktūra:



# Dinaminė atmintis C priemonėmis: new ir delete (1)<sup>1</sup>

- Naudodami new, prisiimate šias atsakomybes:
  - Vėliau atlaisvinsite išskirtą atmintį (su delete) arba įvyks išteklių švaistymas.
  - Teisinga delete forma yra naudojama. Jeigu buvo išskirta vietas vienam objektui - delete, o jeigu grupei (masyvui) - delete [].

<sup>1</sup> Item 13: Prefer vector and string to dynamically allocated arrays

# Dinaminė atmintis C priemonėmis: new ir delete (2)

- Neteisingos delete formos naudojimas lems neapibrėžtą (**undefined**) rezultatą.
- Būti tikriems, kad delete naudojamas **tik vieną kartą**. Jei panaudosite daugiau nei vieną kartą, rezultatas bus neapibrėžtas.
- Todėl, kai tik ruošiatės dinamiškai išskirti atmintį (new T [...]), (beveik) visuomet reiktų rinktis vektorių (vector) ar eilutę (string).

# **Automatinis STL konteinerių augimas (1)**

Vienas iš nuostabiausių dalykų apie STL konteinerius yra tai, kad jie automatiškai auga iki jų maksimalaus dydžio `max_size()`:

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> v;
    std::cout << "Maksimalus 'vector' dydis: " << v.max_size() << "\n";
}
```

Maksimalus 'vector' dydis: 4611686018427387903

# **Automatinis STL konteinerių augimas (2)**

Konteinerių automatinis augimas apima:

1. Naujo atminties bloko, kuris yra nuo 1.5 iki 2 kartų didesnis už ankstesnį, išskyrimas.
2. Visų elementų iš senosios atminties talpyklos kopijavimas į naują atmintį (talpyklą).
3. Senoje atmintyje esančių elementų sunaikinimas.
4. Senos atminties atlaisvinimas.

# Automatinis STL konteinerių augimas (3)

- Šie 4 žingsniai (**allocation**, **copying**, **deallocation**, ir **destruction**) yra "**brangūs**".
- Dar daugiau, kiekvieną kartą, kai šie veiksmai įvyksta, visi iteratoriai (iterator), rodyklės (pointer) ir nuorodos (reference) į vektorių (ar eilutę) tampa neteisingi!
- Būtina žinoti strategijas, kaip to išvengti.

# Keturios STL konteinerių funkcijos (1)<sup>2</sup>

## size() funkcija

1. size() nurodo, kiek elementų yra konteineryje, tačiau nenurodo, kiek atminties išskirta jiems.

```
#include <vector>
#include <iostream>
int main() {
    std::vector<int> vi {1, 3, 5, 7, 9};
    std::cout << "vi turi " << vi.size() << " elementus.\n";
}
```

Rezultatas:

vi turi 5 elementus.

<sup>2</sup> Item 14: Use reserve to avoid unnecessary reallocations

# **Keturios STL konteinerių funkcijos (2)**

## **capacity() funkcija**

**2.** capacity() nurodo, konteinerio talpą, t.y., kiek elementų telpa išskirtame atminties bloke.

- Norit gauti, kiek laisvos atminties turi vektorius/eilutę, reikia: capacity() - size().
- Jei capacity() == size(), konteineryje nėra daugiau tuščios vietas ir kitas įterpimas (push\_back(), insert()) reikalaus atminties perskirstymo.

# Keturios STL konteinerių funkcijos (3)

## capacity() pavyzdys

```
#include <vector>
#include <iostream>
int main() {
    std::vector<int> vi {1, 3, 5, 7, 9};
    std::cout << "vi talpa yra " << vi.capacity() << " elementai.\n";
    vi.push_back(11);
    std::cout << "vi talpa po push_back() yra " << vi.capacity() << " elementai.\n";
}
```

Rezultatas:

vi talpa yra 5 elementai.

vi talpa po push\_back() yra 10 elementai

# Keturios STL konteinerių funkcijos (4)

## resize() funkcija

**3.** `resize(Container::size_type n)` konteinerio dydis tampa **n** elementų, t.y., `size() = n`.

- Jei **n** mažesnis negu dabartinis konteinerio dydis, **pabaigos elementai bus sunaikinti!**
- Jei **n** didesnis nei dabartinis konteinerio dydis, nauji (**default**) elementai bus įtraukti į konteinerio pabaigą.
- Jei **n** didesnis už konteinerio talpa, **atminties perskirstymas įvyks** prieš įtraukiant elementus.

# Keturios STL konteinerių funkcijos (5)

**Pagalbinis header failas:** mano\_funkcijos.h

```
#ifndef MANO_FUNKCIJOS_H
#define MANO_FUNKCIJOS_H

#include <iostream>
#include <vector>
using std::cout; using std::endl; using std::vector;
// Atspausdinti konteinerio (coll) elementus
void printElements(const vector<int>& coll) {
    cout << "coll = { ";
    for (const auto& elem : coll) {
        cout << elem << " ";
    }
    cout << "}\n";
}
// Atspausdinti konteinerio statistika
void printStats(const vector<int>& coll) {
    cout << "size() = " << coll.size() << "\n";
    cout << "capacity() = " << coll.capacity() << "\n";
}
#endif
```

# Keturios STL konteinerių funkcijos (6)

## resize() pavyzdys

```
#include "mano_funkcijos.h"
int main() {
    std::vector<int> c = {1, 2, 3};
    printElements(c);
    printStats(c);
    c.resize(5);
    std::cout << "Po resize() iki 5: \n";
    printElements(c);
    printStats(c);
    c.resize(2);
    std::cout << "Po resize() iki 2: \n";
    printElements(c);
    printStats(c);
}
```

Pradžioje: coll = { 1 2 3 }, size() = 3, capacity() = 3  
Po resize(5): coll = { 1 2 3 0 0 }, size() = 5, capacity() = 6  
Po resize(2): coll = { 1 2 }, size() = 2, capacity() = 6

# Keturios STL konteinerių funkcijos (7)

## reserve() funkcija

4. `reserve(Container::size_type n)` padidina konteinerio **talpą** iki **n**, kai  $n > \text{capacity}()$ .

- Jei  $n < \text{capacity}()$ , std::vector tipo konteineriai ignoruoja, bet std::string gali sumažinti talpą iki  $\max\{\text{size}(), n\}$ , tačiau paties string dydis nesikeičia.
- `reserve()` niekada nekeičia elementų konteineryje skaičiaus (`size()`).

# Keturios STL konteinerių funkcijos (8)

## Svarbus skirtumas tarp resize() ir reserve()

- resize() - inicializuojas vektoriaus elementai:

```
std::vector<int> vi;
vi.resize(4);
std::cout << "vi[0] = " << vi[0]; // vi[0] = 0
```

- reserve() - išskiria vietą, bet neinicializuojas reikšmė:

```
std::vector<int> vi;
vi.reserve(4);
std::cout << "vi[0] = " << vi[0]; // Undefined rezultatas
```

# Keturios STL konteinerių funkcijos (9)

## reserve() pavyzdys

```
#include "mano_funkcijos.h"
int main() {
    std::vector<int> c = {1, 2, 3};
    printElements(c);
    printStats(c);
    c.reserve(5);
    std::cout << "Po reserve() iki 5: \n";
    printElements(c);
    printStats(c);
    c.reserve(2);
    std::cout << "Po reserve() iki 2: \n";
    printElements(c);
    printStats(c);
}
```

Pradžioje: coll = { 1 2 3 }, size() = 3, capacity() = 3  
Po reserve(5): coll = { 1 2 3 }, size() = 3, capacity() = 5  
Po reserve(2): coll = { 1 2 3 }, size() = 3, capacity() = 5

# Keturios STL konteinerių funkcijos (10)

## reserve() - sparta! (1)

Reikia visada naudoti reserve() kai elementų skaičius žinomas iš anksto. Todėl vietoj šio kodo:

```
vector<int> v;
for (int i = 1; i <= 1000; ++i) {
    v.push_back(i);
}
```

Jei capacity() auga dvigubai (x2), tai: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 – atmintis perskirstoma 11 kartų!

# Keturios STL konteinerių funkcijos (11)

## reserve() - sparta! (2)

Reikytų naudoti reserve():

```
vector<int> v;
v.reserve(1000); // rezervuojame 1000 elem.
for (int i = 1; i <= 1000; ++i) {
    v.push_back(i);
}
```

Atminties perskirstymas neatliekamas ne karto!

# Keturios STL konteinerių funkcijos (12)

## reserve() - sparta! (3)

Jei nežinome elem. skaičiaus, rezervuojame "pakankamai daug" talpos:

```
vector<int> v;
v.reserve(2000);
for (int i = 1; i <= 1000; ++i) {
    v.push_back(i);
}
printStats(v);
```

Rezultatas:

```
size() = 1000
capacity() = 2000
```

# Keturios STL konteinerių funkcijos (13)

## reserve() ir shrink\_to\_fit()

- Kai visi pageidauti elementai įtraukti, galime atlaisvinti perteklių:

```
vector<int> v;
v.reserve(2000);
for (int i = 1; i <= 1000; ++i) {
    v.push_back(i);
}
v.shrink_to_fit(); // Nuo C++11
printStats(v);
```

Rezultatas:

```
size() = 1000
capacity() = 1000
```

# Keturios STL konteinerių funkcijos (14)

**reserve() ir swap() triukas (C++03/C++98)<sup>3</sup>**

```
vector<int> v;
v.reserve(2000);
for (int i = 1; i <= 1000; ++i) {
    v.push_back(i);
}
vector<int>(v.begin(), v.end()).swap(v); // C++03 `versija` shrink_to_fit
printStats(v);
```

Rezultatas:

size() = 1000

capacity() = 1000

<sup>3</sup> Item 17: Use “the swap trick” to trim excess capacity

# C++ konteinerių naudojimas C API (1)<sup>4</sup>

- Kai turime std vektorių:

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

tai `v[0]` yra nulinio `v` elemento nuoroda (**reference**), o `&v[0]` - rodyklė (**pointer**) į tą elementą.

- Vektoriaus elementai saugomi vientisame atminties bloke (kaip ir C-array), todėl norint perduoti std vektorių į C API funkciją, pvz.:

```
void oldFunc(const int* pInt, size_t nEl);
```

<sup>4</sup> Item 16: Know how to pass vector and string data to legacy APIs

## C++ konteinerių naudojimas C API (2)

- Iš ją kreipiamės tokiu būdu:

```
oldFunc(&v[0], v.size());
```

- Vienintelė problema jei **v** tuščias (**v.size() = 0**). Tuomet &**v[0]** yra rodyklę į tai kas neegzistuoja! – ko pasekoje gaunamas neapibrėžtas (**undefined**) rezultatas. Tokiu atveju reikytų:

```
if (!v.empty()) { oldFunc(&v[0], v.size()); }
```

## C++ konteinerių naudojimas C API (3)

- Tačiau šis būdas netinka vietoj std::vector naudojant std::string, nes:
  - nėra garantijos, kad string elementai yra saugomi vientisame atminties bloke.
  - nėra garantijos, kad string užsibaigia null ('\0') simboliu.
- Tam tikslui reikia naudoti funkciją .c\_str(), kuri gražina const char\* tipo rodyklę, rodančią į C-stiliaus eilutę.

## C++ konteinerių naudojimas C API (4)

Todėl jei C stiliaus programos kontekste:

```
std::string s = "Tekstas";
void oldFunc(const char* pString);
```

Tuomet į oldFunc() string kintamojo turinį perduodame:

```
oldFunc(s.c_str());
```

Tai veikia, nei jei string yra nulinio dydžio!

# C++ konteinerių naudojimas C API (5)

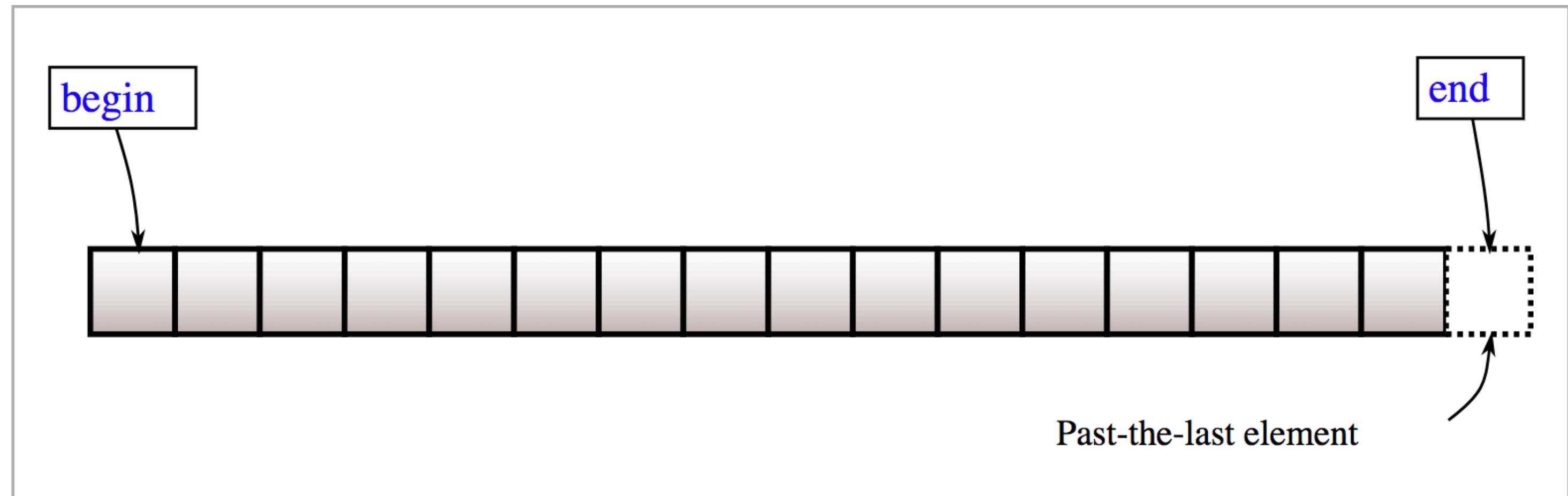
Dar kartą pažiūrėkime į šias dvi funkcijų deklaracijas:

```
void oldFunc(const int *pInt, size_t nEl);  
void oldFunc(const char *pString);
```

- Abiem atvejais perduodamos rodyklės į const rodykles. Tokiu atveju vektoriaus ar eilutės duomenys galės būti tik skaitomi - tai yra saugu ir ko mums reikia!.
- **Kas nutiktų jei leistume modifikuoti?**

# v.begin() ir v.end() (1)

v.begin() ir v.end() grąžina iteratorius ("rodykles") į pirmajį ir vieną po paskutinio **v** elementus.



## v.begin() ir v.end() (2)

```
// vector::begin/end C++98/C++03 versija
#include <iostream>
#include <vector>
int main () {
    std::vector<int> v;
    for (int i=1; i<=5; ++i)
        v.push_back(i);
    std::cout << "v susideda iš:";
    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it)
        std::cout << ' ' << *it;
    return 0;
}
```

v susideda iš: 1 2 3 4 5

## v.begin() ir v.end() (3)

```
// vector::begin/end C++11 versija
#include <iostream>
#include <vector>
int main () {
    std::vector<int> v {1, 2, 3, 4, 5};
    std::cout << "v susideda iš:";
    for (auto it = v.begin(); it != v.end(); ++it)
        std::cout << ' ' << *it;
    return 0;
}
```

myVector susideda iš: 1 2 3 4 5

## v.begin() ir v.end() (4)

```
// vector::begin/end C++11 versija naudojant `foreach
#include <iostream>
#include <vector>
int main () {
    std::vector<int> v {1, 2, 3, 4, 5};
    std::cout << "v susideda iš: ";
    for (const auto& el : v )
        std::cout << el << ' ';
    return 0;
}
```

myVector susideda iš: 1 2 3 4 5

# Ar galime naudoti v.begin() vietoj &v[0]? (1)

```
#include <iostream>
#include <vector>
int main () {
    std::vector<int> v {1, 2, 3, 4, 5};
    std::cout << "v.begin() = " << v.begin() << std::endl;
}
```

```
error: no match for 'operator<<' (operand types are 'std::basic_ostream<char>' and 'std::vector<int>::iterator')
    std::cout << "v.begin() = " << v.begin() << std::endl;
```

begin() - iteratorius, todėl v.begin() != &v[0].

## Ar galime naudoti v.begin() vietoj &v[0]? (2)

```
#include <iostream>
#include <vector>
int main () {
    std::vector<int> v {1, 2, 3, 4, 5};
    std::cout << "v[0] = " << v[0] << std::endl;
    std::cout << "*v.begin() = " << *v.begin() << std::endl;
    std::cout << "&v[0] = " << &v[0] << std::endl;
    std::cout << "&*v.begin() = " << &*v.begin() << std::endl;
}
```

```
v[0] = 1
*v.begin() = 1
&v[0] = 0xd80c20
&*v.begin() = 0xd80c20
```

## **Ar galime naudoti v.begin() vietoj &v[0]? (3)**

Todėl &v[0] analogas naudojant begin() yra &\*v.begin().

Frankly, if you're hanging out with people who tell you to use v.begin() instead of &v[0], you need to rethink your social circle.<sup>4</sup>

<sup>4</sup> Item 16: Know how to pass vector and string data to legacy APIs

## **std::array**

Nuo TR1 C++ standartinė biblioteka turi C-array apvalkalą (**wrapper**) statiniams masyvams<sup>5</sup>.

It is safer and has no worse performance than an ordinary array.

Norint juos naudoti, reikia `<array>` header'io:

```
#include <array>
```

<sup>5</sup> Bjarne Stroustrup. The C++ Programming Language, Third Edition. MA: Addison- Wesley, 1997

## **std::array īcializavimas**

Naudoja neįprastą (du parametrai) īcializavimo sintakę:

```
std::array<int,5> x = {1, 3, 5, 7, 9};
```

Tokia īcializacija negalima:

```
std::array<int,5> x; // OOPS: x reikšmės undefined
```

Tačiau:

```
std::array<int,5> x = {};// Vikas OK: visi elementai = 0 (int())
```

# Klausimai !?

```
(function repeat() {  
    eat();  
    sleep();  
    code();  
    repeat();  
})();
```

I HATE PROGRAMMING  
I HATE PROGRAMMING  
I HATE PROGRAMMING  
I HATE PROGRAMMING  
IT WORKS  
I LOVE PROGRAMMING