

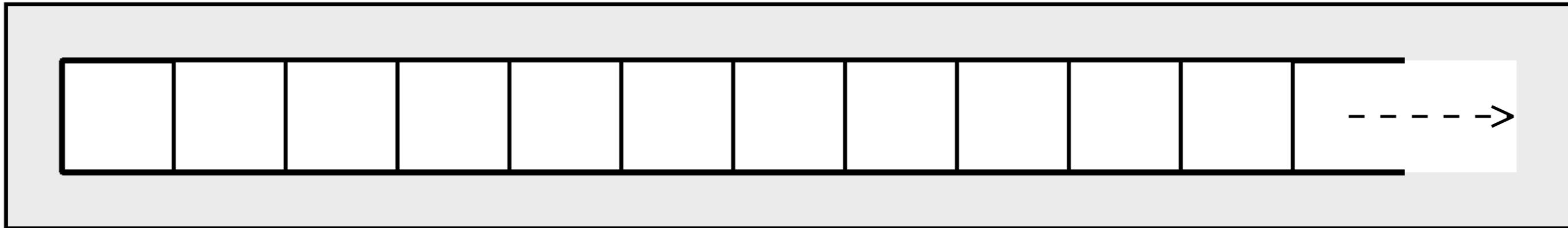


Objektinis Programavimas

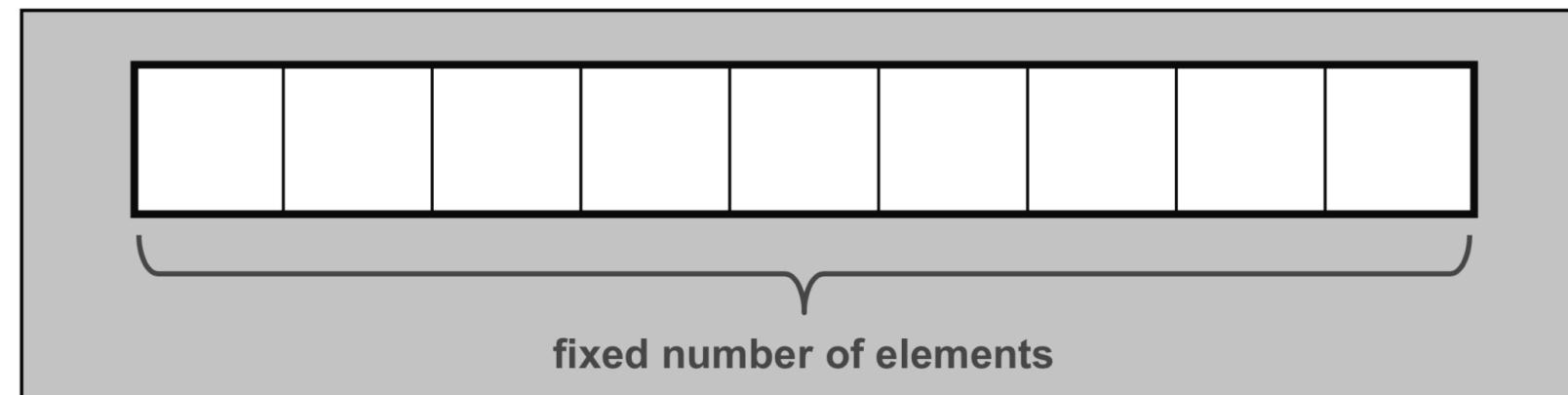
vector vs **C-array** vs **std::array**



vector struktūra:



C-array (std::array) struktūra:





Chapter 2: vector and string

Tikslias: padėti jums geriau suprasti kaip reiktū efektyviau naudoti du labiausiai naudingus STL konteinerius: **vector** ir **string**.

Effective STL

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

Scott Meyers



Dinaminė atmintis su `new` ir `delete` (1)¹

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

¹ Item 13: Prefer vector and string to dynamically allocated arrays

Dinaminė atmintis su new ir delete (2)

- Neteisingos `delete` formos naudojimas lems neapibrėžtą (**`undefined`**) rezultatą.
- Būti tikriems, kad `delete` naudojamas **tik viena karta**. Jei panaudosite daugiau nei vieną kartą, rezultatas bus neapibrėžtas.

Todėl, kai tik ruošiate dinamiškai išskirti atmintį (`new T [...]`), (beveik) visuomet reiktu rinktis vektorių (**vector**) ar eilutę (**string**).

Automatinis STL konteineriu 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 konteineriu augimas (2)

Konteineriu automatinis augimas apima:

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

Automatinis STL konteinerių augimas (3)

- Šie 4 žingsniai (allocation, copying, deallocation, ir destruction) yra "**brangūs**".
- Dar daugiau, kiekviena karta, 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)²

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.

² Item 14: Use reserve to avoid unnecessary reallocations

Keturios STL konteinerių funkcijos (2)

capacity() funkcija

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

- Norit gauti, kiek laisvos atminties turi vektorius/eilutę, reikia: **capacity() - size()**.
- Jei **capacity() == size()**, konteineryje nėra tuščios vietas ir kitas įterpimas (**push_back()**, **insert()**) reikalauja 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 elementu, t.y., `size() = n`.

- Jei n mažesnis negu dabartinis konteinerio dydis, **pabaigos elementai bus sunaikinti**.
- Jei n didesnis nei dabartinis dydis, nauji elementai bus ištraukti iš konteinerio pabaiga.
- Jei n didesnis už talpa, **atminties perskirstymas įvyks** prieš ištraukiant 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 talpa iki n , kai $n > \text{capacity}()$.
 - Jei $n < \text{capacity}()$, `vector` tai ignoruoja, bet `string` gali sumažinti talpa 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 elementus:

```
std::vector vi;  
vi.resize(4);  
vi[0] = 10; // Viskas OK
```

- `reserve()` - išskiria vietos, be inicializavimo:

```
std::vector vi;  
vi.reserve(4);  
vi[0] = 10; // 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. Nenaudojant:

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

Jei **capacity()** x2: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 - atmintis perskirstoma 11 kartu!

Keturios STL konteinerių funkcijos (10)

reserve() – sparta! (2)

Tačiau naudojant **reserve()**:

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

atminties perskirstymas neatliekamas ne karto!

Keturios STL konteinerių funkcijos (11)

reserve() – sparta! (3)

Jei iš anksto nežinome elementų 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 (12)

`reserve()` ir `shrink_to_fit()`

Kai elementai ištraukti, atlaisviname 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 (13)

`reserve()` ir `swap()` triukas (C++03/C++98)³

```
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`

³ Item 17: Use “the swap trick” to trim excess capacity

C++ konteinerių naudojimas C API (1)⁴

- Kai `vector<int> v;` tai `v[0]` – nulinio elemento nuoroda (**reference**), o `&v[0]` – rodyklė (**pointer**) į tai elementą.
- Vektoriaus elementai saugomi vientisame atminties bloke (kaip ir masyvo), todėl norint perduoti vektorių į C API funkcija, pvz.:

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

⁴ 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 rodyklė į tai kas neegzistuoja! – neapibrežtas (**undefined**) rezultatas. Tokiu atveju reiktu:

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

C++ konteinerių naudojimas C API (3)

- Tačiau šis būdas netinka vietoj **vector** naudojant **string**, nes:
 - nėra garantijos, kad **string** elementai yra saugomi vientisame atminties bloke.
 - nėra garantijos, kad **string** užsibaigia null ('\\0') simboliu.
- Todėl **.c_str()** gražina **const char*** tipo rodykľę, kuri rodo į C-stiliaus eilutę.

C++ konteinerių naudojimas C API (4)

Todėl jei C stiliaus kontekste:

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

Tuomet į `oldFunc()` `string` turinį perduodame:

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

Tai veikia, nei jei `string` yra nulinio dydžio!

C++ konteinerių naudojimas C API (5)

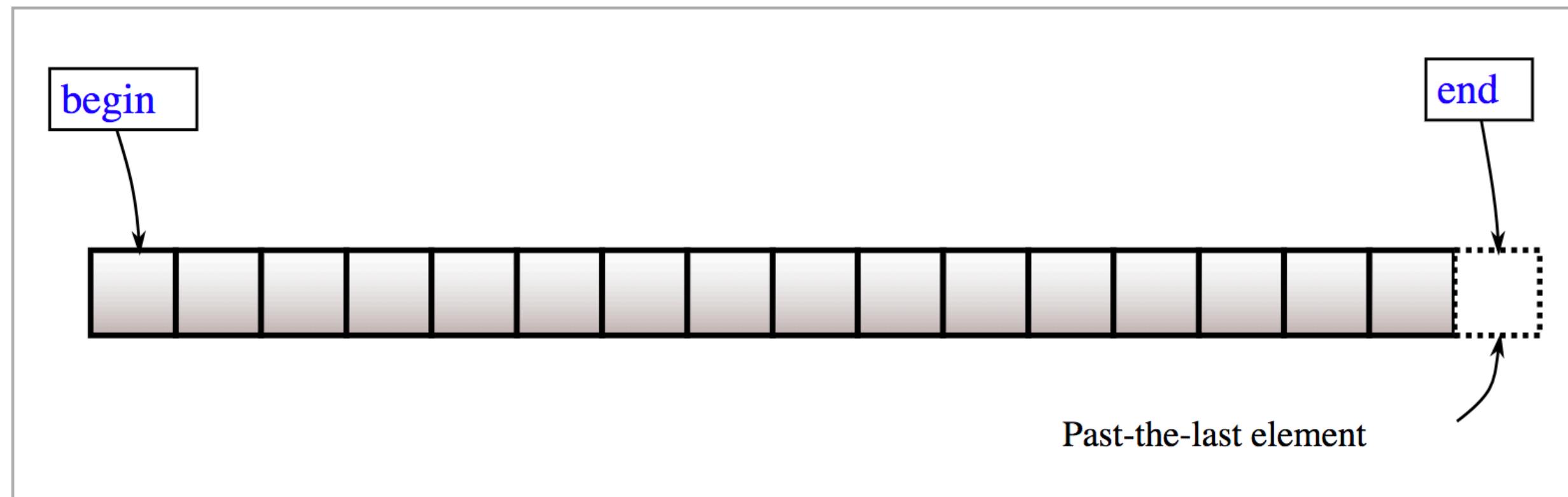
Dar kartą pažiūrėkime į deklaracijas:

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

- Abiem atvejais perduodamos rodyklęs į **const** rodykles. Tokiu atveju vektoriaus ar eilutės duomenys bus tik skaitomi - tai yra saugu! .
- **Kas nutiktų jei leistume modifikuoti?**

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

`begin()` ir `end()` gražina iteratorius ("rodykles") į pirmąjį ir vieną po paskutinio elementus.



Ar galime naudoti v.begin() vietoj &v[0]? (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;
}
```

myVector susideda iš: 1 2 3 4 5

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

Ar galime naudoti v.begin() vietoj &v[0]? (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 (auto& el : v )
        std::cout << el << ' ';
    return 0;
}
```

myVector susideda iš: 1 2 3 4 5

Ar galime naudoti v.begin() vietoj &v[0]? (5)

```
#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]? (6)

```
#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]`? (7)

Todėl `&v[0]` analogas naudojant `begin()` –
`&*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.⁴

⁴ Item 16: Know how to pass vector and string data to legacy APIs

std::array

Nuo TR1 C++ standartinė biblioteka turi C-array apvalkala (**wrapper**) statiniams masyvams⁵.

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

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

```
#include <array>
```

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

std::array inicializavimas - du parametrai

Naudoja neįprastą inicializavimo sintaksę:

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

Tačiau:

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

Bet:

```
std::array<int,5> x = {};
```

// Vikas OK: visi elementai = 0 (int())

