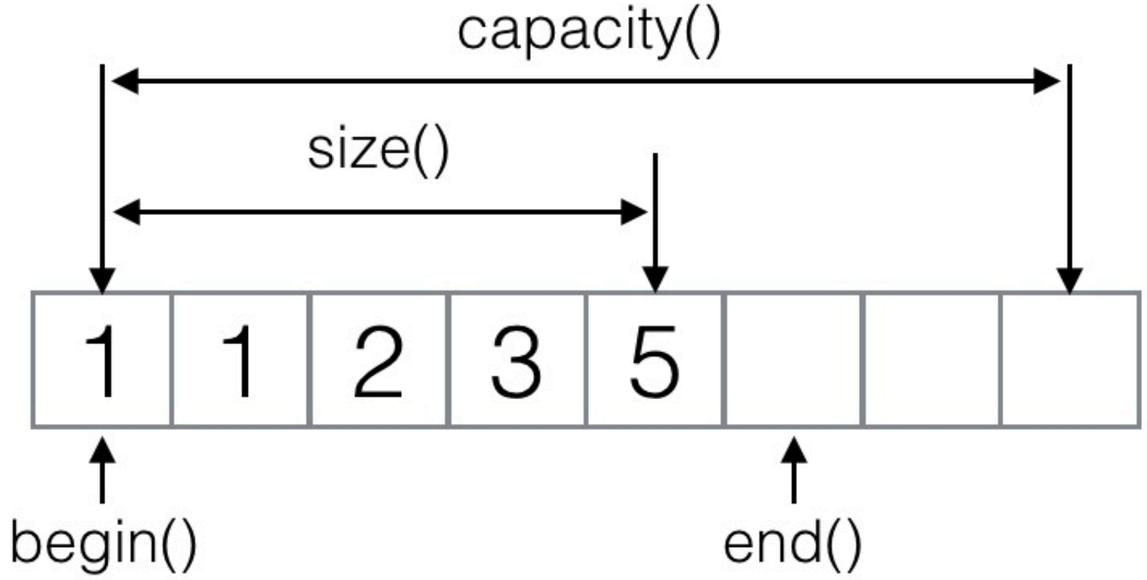
# Objektinis Programavimas

## Vector klasės realizacija


**Iš čia kylama į žvaigždes**



# Turinys

# Vector **klasės realizacija (1)**

```
template <class T>
class Vector {
 public:
    // interfeisas
 private:
    // realizacija
};
```

— Šis kodas sako, kad kuriama Vector yra šabloninė klasė, su vienu parametro-tipu T.

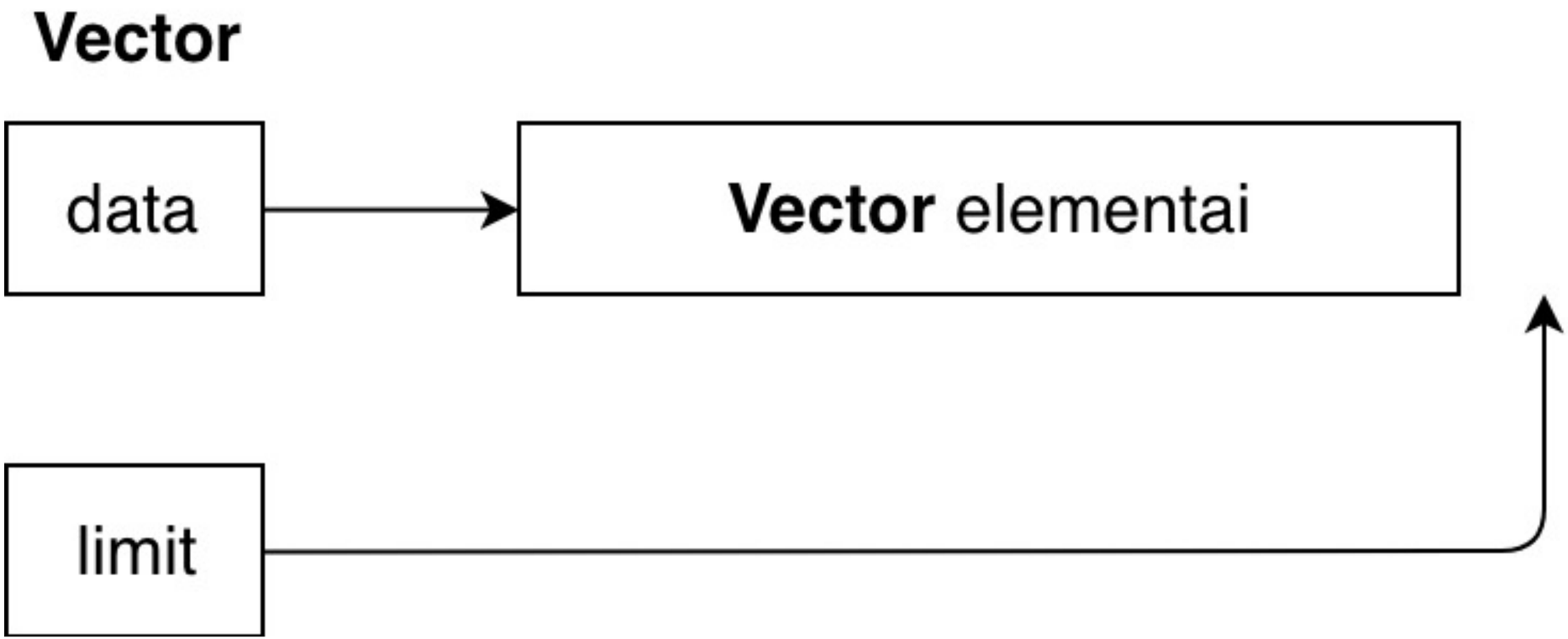
— Kaip ir visų klasių atveju turėsime public and private dalis, kurios apibrėžia **interfeisą** ir **realizaciją**, atitinkamai.

# Vector **klasės realizacija (2)**

- **Mums reikės**:
  - dinamiškai išskiriamos vietos, kurioje saugosime Vector elementus, o taip pat elementų skaičiaus skaitiklio (counter'io).
- Kokią informaciją apie dinaminį masyvą tikslinga saugoti?
  - Kadangi turėsime realizuoti begin, end, ir size funkcijas, todėl logiška būtų saugoti adresus pirmojo ir vieno už paskutinio elementų, o taip pat ir elementų skaičių.
  - Tačiau visų trijų saugoti nėra tikslo, kadangi trečiasis gali būti apskaičiuotas iš kitų dviejų.

# Vector **klasės realizacija (3)**

— Mūsų Vector konteinerio iliustracija:



# Vector **klasės realizacija (4)**

— Todėl galime atnaujinti mūsų Vector klasės realizaciją:

```
template <class T> class Vector {
public:
    // interfeisas
private:
    T* data;          // pirmasis Vector elementas
    T* limit;         // pirmasis po paskutiniojo Vector elementas
};
```

# **Dinaminis atminties valdymas su** new **ir** delete **(1)**

— Pirmasis natūralus pasirinkimas būtų išskirti dinaminę atmintį mūsų Vector naudojant new T[n], kur n yra elementų, kuriems norime išskirti atmintį, skaičius.

— Tačiau kas "nelabai efektyvaus" įvyksta naudojant new:

```
#include <iostream>
class Test {
public:
    Test() { std::cout << "Default c-tor\n"; }
};

int main() {
    Test *x = new Test[10]; // Ką gausime?
}
```

# **Dinaminis atminties valdymas su** new **ir** delete **(2)**

— Kaip matyti new T[n] ne tik išskiria vietą atmintyje, tačiau ir inicializuoja elementus panaudodama **default konstruktorių** T tipo elementams: T::T()

— Dar daugiau, su new T[n] veiks tik tuomet kai T tipas turės **default konstruktorių**. Įsitikinkite!

— Pasirodo C++ standartinėje bibliotekoje yra realizuota speciali klasė, suteikianti detalesnę atminties kontrolę, t.y. leidžia iš pradžių tik išskirti atmintį, o tuomet atskirame žingsnyje sukurti joje objektus.

— Tačiau mes prie jos panaudojimo sugrįšime truputį vėliau, kai realizuosime mūsų pagalbines funkcijas, pvz. create().

# Vector **klasės realizacija (5)**

## Konstruktoriai

```
template <class T> class Vector {
public:
    // konstruktoriai
    Vector() { create(); } // TODO: create atsakinga už atminties išskyrimą
    explicit Vector(size_type n, const T& val = T{}) { create(n, val); }
    // likusi interfeiso realizacija
private:
    T* data;
    T* limit;
};
```

— **default konstruktorius** (be input parametrų) turės sukurti tuščią Vector, todėl data ir limit bus "nuliai".

# Vector **klasės realizacija (6)**

## Konstruktoriai

— Mūsų antrasis konstruktorius turi du parametrus, iš kurių antrasis turi ir **default** reikšmę, todėl jis iš tiesų apibrėžia du konstruktorius: vieną su vienu size_type tipo parametru ir kitą su size_type ir const T& tipo parametrais

— Raktažodis explicit užtikrina, kad kompiliatorius naudotų konstruktorių tik šiame kontekste, kuriame mes norime, ir ne kitaip:

```
Vector<int> vi(100);      // ok, explicitly sukonstruojame Vector
Vector<int> vi = 100;     // error, implicitly neleidžiame konstruot Vector
```

# Vector **klasės realizacija (7)**

## Naujų tipų apibrėžimai

— Pagal standartinių šablonų klasių taikomas praktikas ir mes norime įvesti tam tikrus tipų pavadinimus, kuriuos galės naudoti mūsų Vector vartotojai.

— Tuo tikslu, panaudosime typedef'us (arba using nuo **C++11**) const ir neconst **iterator**'ių tipams, o taip pat ir tipui, kurį naudosime Vector'iaus **size** funkcijai.

— Pasirodo C++ apibrėžia dar vieną tipą, pavadintą value_type, kuris yra **sinonimas** konteinerio saugomų objektų tipui.

# Vector **klasės realizacija (8)**

## Naujų tipų apibrėžimai

— Kadangi naudosime masyvą (**array**) Vector'iaus elementams saugoti, todėl galime naudoti yprastas rodykles (**plain pointers**) kaip mūsų Vector **iterator**'ių tipą.

— Yprastos rodyklės palaiko visas **random-access-iterator** operacijas.

— Akivaizdu, kad value_type turi būti T tipo.

— Kaip dėl size_type (Vector::size_type) tipo? Šioje vietoje puikiausiai galime panaudoti size_t!.

# Vector **klasės realizacija (9)**

```
template <class T> class Vector {
public:
    typedef T* iterator;                    // pridėta
    typedef const T* const_iterator;        // pridėta
    typedef size_t size_type;               // pridėta
    typedef T value_type;                   // pridėta

    Vector() { create(); }
    explicit Vector(size_type n, const T& val = T{}) { create(n, val); }
    // likusi interfeiso realizacija
private:
    iterator data;                          // pakeista iš T* į iterator
    iterator limit;                         // pakeista iš T* į iterator
};
```

# Vector **klasės realizacija (10)**

## **Indeksavimas ir** size

— Funkcija size tradiciškai yra naudojama norint sužinoti Vector elementų skaičių, o operator[] naudojamas Vector elementams pasiekti, konteksuote kaip šis:

```
for (i = 0; i != v.size(); ++i)
    std::cout << v[i].vardas();
```

# Vector **klasės realizacija (11)**

## **Indeksavimas ir** size

```
template <class T> class Vector {
public:
    /* viskas kaip buvo prieš tai */

    // size funkcija
    size_type size() const { return limit - data; }
    // indeksavimas
    T& operator[](size_type i) { return data[i]; }
    const T& operator[](size_type i) const { return data[i]; }
    // likusi interfeiso realizacija
private:
    iterator data;
    iterator limit;
};
```

# Vector **klasės realizacija (12)**

## **Indeksavimas ir** size

— Kaip matyti, size funkcija apskaičiuoja Vector elementų skaičių iš dviejų rodyklių (point'erių) skirtumo, kuris ir parodo kiek yra elementų tarp tų rodyklių (ptrdiff_t tipo reikšmė).

— Grąžinant šią reikšmę iš size funkcijos, ji yra konvertuojama į size_type, kuris yra sinonimas size_t.

— Kadangi ši funkcija nekeičia vektoriaus Vector dydžio todėl ją padarome const, o tuo pačiu galėsime dabar sužinoti dydį ir const Vector objektų.

# Vector **klasės realizacija (13)**

## Funkcijos grąžinančios iteratorius

```
template <class T> class Vector {
public:
    /* viskas kaip anksčiau */

    // naujos funkcijos, grąžinančios iteratorius
    iterator begin() { return data; }                // pridėta
    const_iterator begin() const { return data; }    // pridėta
    iterator end() { return limit; }                 // pridėta
    const_iterator end() const { return limit; }     // pridėta
private:
    iterator data;
    iterator limit;
};
```

— Realizavome dvi begin ir end persidengiančias funkcijas, priklausomai nuo to, ar Vector yra const.

— Tuomet const versija grąžina const_iterators.

# Vector **klasės realizacija (14)**

Rule of three**: Copy konstruktorius, priskyrimo (=) operatorius ir destruktorius**

```
template <class T> class Vector {
public:
    // copy konstruktorius
    Vector(const Vector& v) { create(v.begin(), v.end()); } // TODO: kita create() versija
    // priskyrimo operatorius
    Vec& operator=(const Vec&);
    // destruktorius
    ~Vec() { uncreate(); } // TODO: turėsime realizuoti uncreate()
    /* kaip anksčiau */
};
```

# Vector **klasės realizacija (15)**

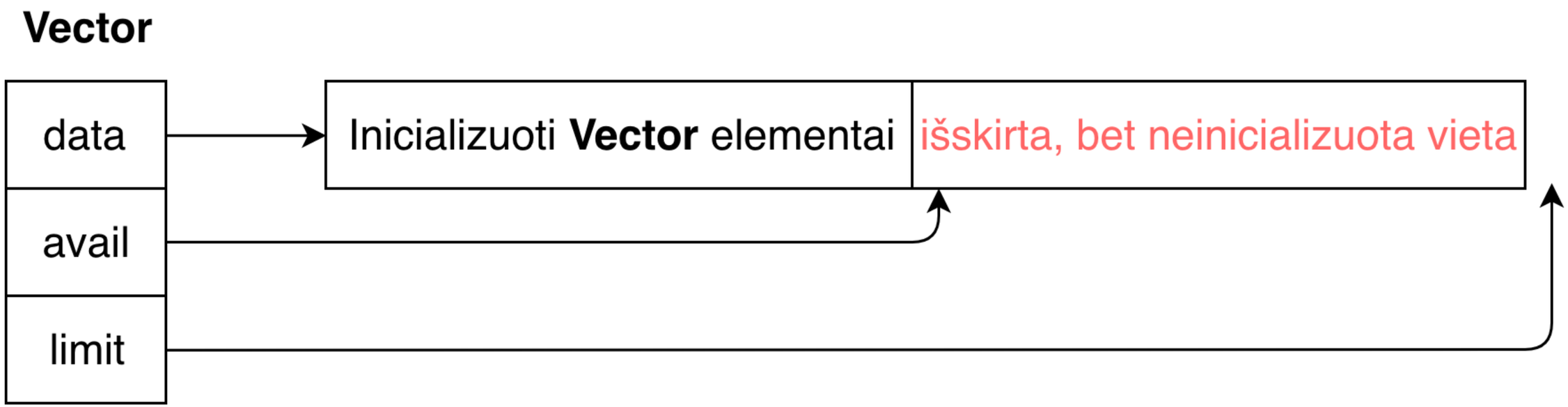
## Priskyrimo (=) operatoriaus realizacija

```
// priskyrimo operatoriaus realizacija
template <class T>
Vector<T>& Vector<T>::operator=(const Vector& rhs) {
    // patikriname ar ne lygu
    if (&rhs != this) {
        // atlaisviname seną this objekto atmintį
        uncreate();
        // perkopijuojame elementus iš rhs į lhs (this)
        create(rhs.begin(), rhs.end());
    }
    return *this;
}
```

# Vector **klasės realizacija (16)**

## push_back() **realizacijai reikia pergalvoti mūsų Vector**

— Atnaujinta mūsų kuriamo Vector konteinerio iliustracija:



# Vector **klasės realizacija (17)**

## Vector **klasės invariantas**

Nusakomas 4-omis taisyklėmis:

1. data rodyklė nukreipta į pirmą data elementą (jei egzistuoja) ir nullptr - priešingu atveju.
2. `data <= avail <= limit`
3. Sukonstruoti elementai yra intervale: **[data, avail)**.
4. Nesukonstruoti elementai yra intervale: **[avail, limit)**.

# Vector **klasės realizacija (18)**

## push_back() **realizacija**

```
template <class T> class Vec {
public:
    // push_back() realizacija
    void push_back(const T& val) {
        if (avail == limit)          // išskirti vietos, jei reikia
            grow();                  // TODO: reikia realizuoti
        unchecked_append(val) ;      // įdėti naują elementą
    }
private:
    iterator data;      // kaip buvo anksčiau
    iterator avail;     // pirmasis po paskutiniojo sukonstruoto Vector elementas
    iterator limit;     // pirmasis po paskutiniojo Vector elementas
    /* visa kita, kaip buvo anksčiau */
};
```

# Vector **klasės realizacija (19)**

allocator**'iai**

— Dinaminiam atminties valdymui <memory> header'yje yra tam skirta klasė allocator<T>

— Ją panaudojant galime išskirti (neinicializuotą) atminties bloką, skirtą T tipo objektų saugojimui.

— C++ standartinė biblioteka suteikia ir galimybę konstruoti objektus šioje išskirtoje atmintyje, o taip pat ir sunaikinti objektus, neatliekant išskirtos atminties **deallocation**.

— Tačiau programų kūrėjų darbas yra pasirūpinti sekti, kuri atminties vieta yra tik išskirta, o kur jau ir objektai sukonstruoti.

# Vector **klasės realizacija (20)**

## allocator**'iai**

The std::allocator class template is the default Allocator used by all standard library containers if no user-specified allocator is provided.[allocator]

— Mūsų poreikiams reikės pasinaudoti 4-omis allocator klasės nario funkcijomis:

```
template<class T> class allocator {
public:
    T* allocate(size_t);           // išskirti `raw` atmintį
    void deallocate(T*, size_t);   // atlaisvinti atmintį
    void construct(T*, const T&); // sukonstruoti 1 objektą
    void destroy(T*);              // sunaikinti 1 objektą
    // ...
};
```

[allocator] https://en.cppreference.com/w/cpp/memory/allocator

# Vector **klasės realizacija (21)**

uninitialized_fill **ir** uninitialized_copy **funkcijos**

— Reikės pasinaudoti ir dvejomis susijusiomis funkcijomis: uninitialized_fill[unin_fill] ir uninitialized_copy[unin_copy]

```
// Copies the given value to an uninitialized memory area, defined by the range [first, last)
template<class ForwardIt, class T>
void uninitialized_fill(ForwardIt first, ForwardIt last, const T& value);

// Copies elements from the range [first, last) to an uninitialized memory area beginning at d_first
template<class InputIt, class ForwardIt>
ForwardIt uninitialized_copy(InputIt first, InputIt last, ForwardIt d_first);
```

unin_fill https://en.cppreference.com/w/cpp/memory/uninitialized_fill

unin_copy https://en.cppreference.com/w/cpp/memory/uninitialized_copy

# Vector **klasės realizacija (22)**

```
template <class T> class Vec {
public: // interfeisas
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef size_t size_type;
    typedef T value_type;

    // rule of three
    Vec() { create(); }
    explicit Vec(size_type n, const T& t = T{}) { create(n, t); }
    Vec(const Vec& v) { create(v.begin(), v.end()); }
    Vec& operator=(const Vec&);
    ~Vec() { uncreate(); }

    T& operator[](size_type i) { return data[i]; }
    const T& operator[](size_type i) const { return data[i]; }

    void push_back(const T& t) {
        if (avail == limit)
            grow();
        unchecked_append(t);
    }

    size_type size() const { return avail - data; }
    iterator begin() { return data; }
    const_iterator begin() const { return data; }
    iterator end() { return avail; }
    const_iterator end() const { return avail; }
```

# Vector **klasės realizacija (23)**

```
// tęsinys ankstesnės skaidrės
private:
    iterator data;     // kaip buvo anksčiau
    iterator avail;    // pirmasis po paskutiniojo sukonstruoto Vector elementas
    iterator limit;    // pirmasis po paskutiniojo Vector elementas

    // atminties išskyrimo valdymui
    allocator<T> alloc;   // objektas atminties valdymui

    // išskirti atmintį (array) ir ją inicializuoti
    void create() ;
    void create(size_type, const T&);
    void create(const_iterator, const_iterator);

    // sunaikinti elementus array ir atlaisvinti atmintį
    void uncreate();

    // pagalbinės funkcijos push_back realizacijai
    void grow();
    void unchecked_append(const T&);
};
```

# **Persidengiančios** create() **funkcijos realizacijos**

```
template <class T> void Vector<T>::create() {
    data = avail = limit = nullptr;
}

template <class T> void Vector<T>::create(size_type n, const T& val) {
    data = alloc.allocate(n); // grąžina ptr į array pirmą elementą
    limit = avail = data + n; // sustato rodykles į vietas
    uninitialized_fill(data, limit, val); // inicializuoja elementus val reikšme
}

template <class T>
void Vector<T>::create(const_iterator i, const_iterator j) {
    data = alloc.allocate(j - i); // išskirti vietos j-i elementams
    limit = avail = uninitialized_copy(i, j, data); // nukopijuoja elementus iš intervalo
}
```

# uncreate() **funkcijos realizacija**

```
template <class T> void Vector<T>::uncreate() {
    if (data) {
        // sunaikinti (atbuline tvarka) sukonstruotus elementus
        iterator it = avail;
        while (it != data)
            alloc.destroy(--it);

        // atlaisvinti išskirtą atmintį. Turi būti data != nullptr
        alloc.deallocate(data, limit - data);
    }
    // reset'inam pointer'iuss - Vector'ius tuščias
    data =  limit  =  avail  =  nullptr;
}
```

# **Pagalbinės** push_back() **funkcijos**

## grow **ir** unchecked_append **funkcijų realizacijos**

```
template <class T> void Vector<T>::grow() {
    // dvigubai daugiau vietos, nei buvo
    size_type new_size = max(2 * (limit - data), ptrdiff_t(1));

    // išskirti naują vietą ir perkopijuoti egzistuojančius elementus
    iterator new_data = alloc.allocate(new_size);
    iterator new_avail = uninitialized_copy(data, avail, new_data);

    // atlaisvinti seną vietą
    uncreate();

    // reset'int rodykles į naujai išskirtą vietą
    data = new_data;
    avail = new_avail;
    limit = data + new_size;
}

// tariame, kad `avail` point'ina į išskirtą, bet neinicializuotą vietą
template <class T> void Vector<T>::unchecked_append(const T& val) {
    alloc.construct(avail++, val);
}
```

# Klausimai?!