Objektinis Programavimas

Copy ir move semantika

Iš čia kylama
į žvaigždes

Turinys

1. Objektų kopijavimas (*copy* semantika)
2. Rules of three and five
3. Objektų perkėlimas (*move* semantika)
4. `std::move()`

Objektų kopijavimas (1)

— Objektai, kaip ir fundamentalieji kintamieji, gali būti nukopijuoti.

— Default kopijavimas vyksta perkopijuojant kiekvieną objekto narį.

— **Turime apsvarstyti**, ar objektas gali (ir kaip gali) būti kopijuojamas!

```
#include <iostream>
#include "Complex.h"
int main() {
  Complex c1 {1.0, 1.0};  // sukonstruojame kompleksinį sk.: (1.0, 1.0)
  Complex c2 {c1};        // copy-initialization
  Complex c3;             // default konstruktorius
  c3 = c1;                // copy-assignment
  std::cout << "c1: " << c1 << "c2: " << c2 << "c3: " << c3 << std::endl; // Ką gausime?
}
```

Objektų kopijavimas (2)

— Paprastiems konkretiems tipams panariui (*memberwise*) kopijavimas dažniausiai yra teisinga kopijavimo semantika (strategija).

— Tačiau sudėtingesniems (konkretiems) tipams panariui kopijavimas dažnai yra klaidinga strategija.

— Kai klasė yra atsakinga už resursų valdymą (t.y. kai klasė atsakinga už objektą pasiekiamą per rodyklę), panariui kopijavimas dažniausiai yra didelės nelaimės pradžia. **Kodėl?**

Vector klasė (1)

```
#include <iostream>
class Vector {
 private:
  int sz;
  double* elem;
 public:
  Vector(int s, double val) : sz(s), elem(new double[sz]) { std::fill_n(elem, s, val); }
  int size() const { return sz; }
  double& operator[](int i) { return elem[i]; } // Ar čia viskas gerai?
};

int main() {
  Vector v1 {2, 1.0}; // sukonstruojame vektorių: (1.0, 1.0)
  std::cout << v1[1] << ", " << v1[2] << std::endl; // Ką gausime?
}
```

Vector klasė (2)

```
#include <iostream>
class Vector {
 private:
  int sz;
  double* elem;
 public:
  Vector(int s, double val) : sz(s), elem(new double[sz]) { std::fill_n(elem, s, val); }
  int size() const { return sz; }
  double& operator[](int i) {
    if (i < 0 || size() <= i) throw std::out_of_range {"Vector::operator[]"};
    return elem[i];
  }
};

int main() {
  Vector v1 {2, 1.0}; // sukonstruojame vektorių: (1.0, 1.0)
  std::cout << v1[1] << ", " << v1[2] << std::endl; // Ką gausime?
}
```

Vector klasė (3)

```
#include <iostream>
class Vector {
 private:
  int sz;
  double* elem;
 public:
  Vector(int s, double val) : sz(s), elem(new double[sz]) { std::fill_n(elem, s, val); }
  int size() const { return sz; }
  double& operator[](int i) {
    if (i < 0 || size() <= i) throw std::out_of_range {"Vector::operator[]"};
    return elem[i];
  }
};

int main() {
  Vector v1 {2, 1.0}; // sukonstruojame vektorių: (1.0, 1.0)
  std::cout << v1[0] << ", " << v1[1] << std::endl; //
  const Vector v2 {2, 2.0}; // sukonstruojame const vektorių: (2.0, 2.0)
  std::cout << v2[0] << ", " << v2[1] << std::endl; // Ką gausime?
}
```

Vector klasė (4)

```
#include <iostream>
class Vector {
 private:
  int sz;
  double* elem;
 public:
  Vector(int s, double val) : sz(s), elem(new double[sz]) { std::fill_n(elem, s, val); }
  int size() const { return sz; }
  double& operator[](int i) {
    if (i < 0 || size() <= i) throw std::out_of_range {"Vector::operator[]"};
    return elem[i];
  }
  const double& operator[](int i) const { return elem[i]; } // Turi buti throw
};

int main() {
  Vector v1 {2, 1.0}; // sukonstruojame vektorių: (1.0, 1.0)
  std::cout << v1[0] << ", " << v1[1] << std::endl; //
  const Vector v2 {2, 2.0}; // sukonstruojame const vektorių: (2.0, 2.0)
  std::cout << v2[0] << ", " << v2[1] << std::endl; // Ką gausime?
}
```

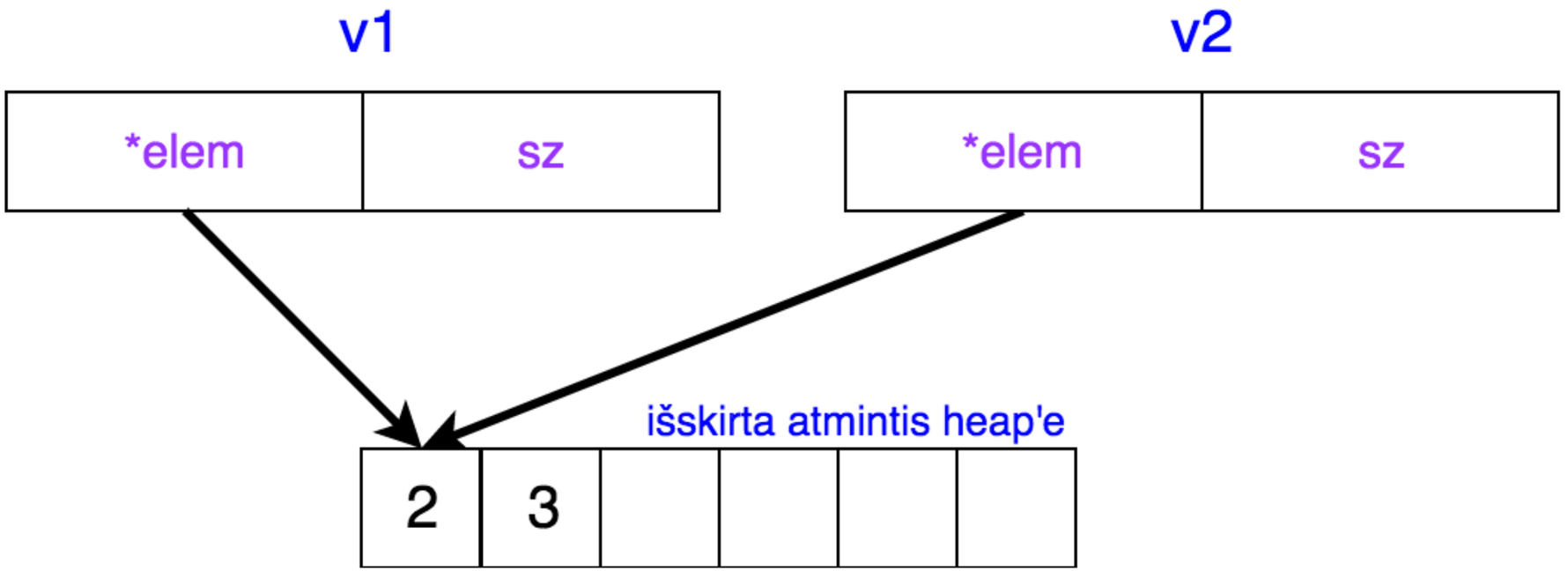
Objektų kopijavimas (3)

Vector klasės objektų kopijavimas

```
#include <iostream>
class Vector {
 private:
  int sz;
  double* elem;
 public:
  Vector(int s, double val) : sz(s), elem(new double[sz]) { std::fill_n(elem, s, val); }
  double& operator[](int i) { return elem[i]; } // Ar čia viskas gerai?
  const double& operator[](int i) const { return elem[i]; } // KAM ŠITO REIKIA?
};
int main() {
  Vector v1 {2, 1.0};  // sukonstruojame vektorių: (1.0, 1.0)
  Vector v2 = v1;      // nukopijuojame v1 vektorių
  v1[0] = 2.0;
  v2[1] = 3.0;
  std::cout << v1[0] << ", " << v1[1] << std::endl; // Ką gausime?
  std::cout << v2[0] << ", " << v2[1] << std::endl; // Ką gausime?
}
```

Objektų kopijavimas (4)

Vector'ių kopijavimo (*memberwise*) iliustracija



— Bet jeigu taip daryti blogai, kodėl (po 😈) kompiliatorius neįspėjo?

Objektų kopijavimas (5)

— Ar prisimenate C++ *Garbage collector* ?

The technique of acquiring resources in a constructor and releasing them in a destructor, known as Resource Acquisition Is Initialization or RAII.

— Bjarne Stroustrup

— Jei `Vector` turėtų **destruktorių**, kompiliatorius turėtų bent įspėti, kad *memberwise* semantika yra neteisinga!

Objektų kopijavimas (6)

```
#include <iostream>
class Vector {
 private:
  int sz;
  double* elem;
 public:
  Vector(int s, double val) : sz(s), elem(new double[sz]) { std::fill_n(elem, s, val); }
  ~Vector() { delete[] elem; }
  double& operator[](int i) { return elem[i]; }
};
int main() {
  Vector v1 {2, 1.0};  // sukonstruojame vektorių: (1.0, 1.0)
  Vector v2 = v1;      // nukopijuojame v1 vektorių
  v1[0] = 2.0;
  v2[1] = 3.0;
  std::cout << v1[0] << ", " << v1[1] << std::endl; // Ką dabar gausime?
  std::cout << v2[0] << ", " << v2[1] << std::endl; // Ką dabar gausime?
}
```

Objektų kopijavimas (7)

— Klasės objekto kopijavimą apibrėžia **kopijavimo konstruktorius** (*copy constructor*) ir kopijavimo **priskirties operatorius** (*copy assignment*) 🙀

```
class Vector {
private:
  double* elem;
  int sz;
public:
  Vector() : sz(0), elem(new double[sz]) {}  // konstruktorius išskiria resursus
  Vector(int s) : sz{ s }, elem{ new double[sz] } { std::fill_n(elem, s, 0.0); }
  Vector(int s, double val)
  ~Vector() { delete[] elem; }               // destruktorius: atlaisvina resursus

  Vector(const Vector& v);                   // copy konstruktorius
  Vector& operator=(const Vector& v);        // priskyrimo operatorius

  double& operator[](int i);
  int size() const;
};
```

Copy konstruktorius (1)

```
// Vector.cpp realizacija
Vector::Vector(const Vector& v) // copy konstruktorius
    : sz{v.sz},                 // inicializuojame sz
      elem{new double[v.sz]}    // išskiriame atmintį elem
{
  for (int i=0; i!=sz; ++i)   // nukopijuojame elementus
    elem[i] = v.elem[i];
}
```

— Ar pastebite kažką neįprasto?

Copy konstruktorius (2)

```
// Vector.cpp realizacija
Vector::Vector(const Vector& v) // copy konstruktorius
    : sz{v.sz},                 // inicializuojame sz
      elem{new double[v.sz]}    // išskiriame atmintį elem
{
  for (int i=0; i!=sz; ++i)   // nukopijuojame elementus
    elem[i] = v.elem[i];
}
```

You can access private members of a class from within the class, even those of another instance. [access]

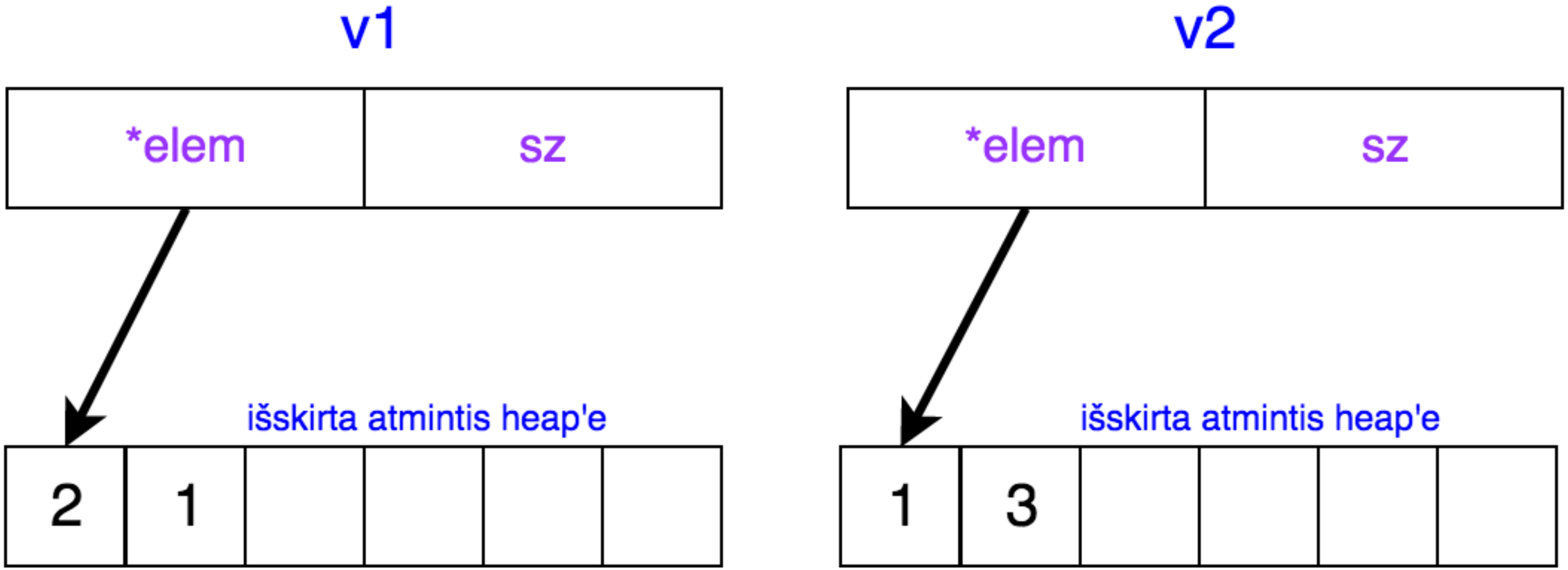
[access] https://stackoverflow.com/a/4117020/3737891

Objektų kopijavimas (8)

```
#include <iostream>
class Vector {
 private:
  int sz;
  double* elem;
 public:
  Vector() : sz(0), elem(new double[sz]) {}  // default konstruktorius
  Vector(int s, double val) : sz(s), elem(new double[sz]) { std::fill_n(elem, s, val); }
  ~Vector() { delete[] elem; }
  Vector(const Vector& v) :elem{new double[v.sz]}, sz{v.sz} { // copy konstruktorius
    for (int i=0; i!=sz; ++i) elem[i] = v.elem[i];
  }
  double& operator[](int i) { return elem[i]; }
};
int main() {
  Vector v1 {2, 1.0};  // sukonstruojame vektorių: (1.0, 1.0)
  Vector v2 = v1;      // copy konstruktorius kopijuoja
  v1[0] = 2.0;
  v2[1] = 3.0;
  std::cout << v1[0] << ", " << v1[1] << std::endl; // Ką dabar gausime?
  std::cout << v2[0] << ", " << v2[1] << std::endl; // Ką dabar gausime?
}
```

Objektų kopijavimas (9)

Vector'ių kopijavimas: copy konstruktorius



Objektų kopijavimas (10)

```
int main() {
  Vector v1 {2, 1.0};  // sukonstruojame vektorių: (1.0, 1.0)
  Vector v2 {v1};      // (ar tik) kita sintaksė?
  v1[0] = 2.0;
  v2[1] = 3.0;
  std::cout << v1[0] << ", " << v1[1] << std::endl; // Ką dabar gausime?
  std::cout << v2[0] << ", " << v2[1] << std::endl; // Ką dabar gausime?
}
```

— O ką dabar gausime?

Objektų kopijavimas (11)

```
int main() {
  Vector v1 {2, 1.0};  // sukonstruojame vektorių: (1.0, 1.0)
  Vector v2;           // default konstruktorius
  v2 = v1;             // v2 priskiriame vektorių v1
  v1[0] = 2.0;
  v2[1] = 3.0;
  std::cout << v1[0] << ", " << v1[1] << std::endl; // Ką dabar gausime?
  std::cout << v2[0] << ", " << v2[1] << std::endl; // Ką dabar gausime?
}
```

— O ką dabar gausime?

Priskyrimo operatorius (operator=)

```
// Vector.cpp realizacija
Vector& Vector::operator=(const Vector& v) { // priskyrimo operatorius
    // Savęs priskyrimo aptikimas
    if (&v == this) return *this;

    double* p = new double[v.sz];
    for (int i=0; i!=v.sz; ++i) // nukopijuojame v elementus
        p[i] = v.elem[i];
    delete[] elem; // atlaisviname seną atmintį!
    elem = p;      // elem point'ina į naują atmintį
    sz = v.sz;     // atnaujiname size
    return *this;  // grąžiname objektą
}
```

Objektų kopijavimas (12)

Naudojant priskyrimo operatorių

```
// Includinti Vector.h su realizuotu operator=
int main() {
  Vector v1 {2, 1.0};  // sukonstruojame vektorių: (1.0, 1.0)
  Vector v2;           // default konstruktorius
  v2 = v1;             // nukopijuojame v1 per operator=
  v1[0] = 2.0;
  v2[1] = 3.0;
  std::cout << v1[0] << ", " << v1[1] << std::endl; // Ką dabar gausime?
  std::cout << v2[0] << ", " << v2[1] << std::endl; // Ką dabar gausime?
}
```

Rule of three[r3]

The rule of three (the Law of The Big Three) is a rule of thumb in C++ (prior to C++11) that claims that if a class defines one (or more) of the following it should probably explicitly define all three:

— destructor

— copy constructor

— copy assignment operator

[r3] https://en.wikipedia.org/wiki/Rule*of*three(*C%2B%2B*programming)

Vectorių sudėtis (1)

```
#include <iostream>
class Vector {
 private:
  int sz;
  double* elem;
 public:
  Vector() : sz(0), elem(new double[sz]) {}  // default konstruktorius
  Vector(int s) : sz{s}, elem{new double [sz]} {}
  Vector(int s, double val) : sz(s), elem(new double[sz]) { std::fill_n(elem, s, val); }
  ~Vector() { delete[] elem; }
  int size() const { return sz; }
  double& operator[](int i) { return elem[i]; }
};

int main() {
  Vector a {10, 1}; // kviečia Vector(int, double) konstruktorių
  Vector b {10, 2};
  Vector c  = a + b;
  for (auto i = 0; i != c.size(); ++i) std::cout << c[i] << " ";
}
```

Vector::operator+() realizacija (1)

```
// Vector.cpp faile
#include <exception>

Vector operator+(const Vector& a, const Vector& b) {
  if (a.size() != b.size())
    throw std::runtime_error("Vektorių dydžio neatitikimas!");
  auto size = a.size();
  Vector c(size);
  for (auto i = 0; i != a.size(); ++i)
    c[i] = a[i] + b[i];
  return c;
}
```

Vectorių sudėtis (2)

```
#include <iostream>
class Vector {
 private:
  int sz;
  double* elem;
 public:
  Vector() : sz(0), elem(new double[sz]) {}  // default konstruktorius
  Vector(int s) : sz{s}, elem{new double[sz]} {}
  Vector(int s, double val) : sz(s), elem(new double[sz]) { std::fill_n(elem, s, val); }
  ~Vector() { delete[] elem; }
  int size() const { return sz; }
  double& operator[](int i) { return elem[i]; }
  const double& operator[](int i) const { return elem[i]; } // Kam šio reikia?
};

/* Pridedam Vector operator+ realizaciją */

int main() {
  Vector a {10, 1};
  Vector b {10, 2};
  Vector c  = a + b;
  for (auto i = 0; i != c.size(); ++i) std::cout << c[i] << " ";
}
```

Vectorių sudėtis (3)

```
#include <iostream>
class Vector {
 private:
  int sz;
  double* elem;
 public:
  Vector() : sz(0), elem(new double[sz]) {}  // default konstruktorius
  Vector(int s) : sz{s}, elem{new double[sz]} {}
  Vector(int s, double val) : sz(s), elem(new double[sz]) { std::fill_n(elem, s, val); }
  ~Vector() { delete[] elem; }
  int size() const { return sz; }
  double& operator[](int i) { return elem[i]; }
  const double& operator[](int i) const { return elem[i]; }
};

/* Pridedam Vector operator+ realizaciją */

int main() {
  Vector a {1, 3, 5, 7}; // Kuris konstruktorius?
  Vector b {2, 4, 6, 8};
  Vector c  = a + b;
  for (auto i = 0; i != c.size(); ++i) std::cout << c[i] << " ";
}
```

Initializer-list konstruktorius

```
// #include <algorithm>
Vector(std::initializer_list<double> il)
  : sz{static_cast<int>(il.size())}, // kam reikalingas static_cast?
    elem{new double[il.size()]}
{
  std::copy(il.begin(),il.end(),elem); // nukopijuoti iš il į elem
}

int main() {
  Vector a {1, 3, 5, 7};
  Vector b {2, 4, 6, 8};
  Vector c  = a + b;
  for (auto i = 0; i != c.size(); ++i)
    std::cout << c[i] << " ";
}
```

Benchmark'iname `Vector`ių kopijavimą (`operator+`) (1)

Copy semantikos (ne)efektyvumo tyrimas

Panagrinėkime, kas čia vyksta detaliai (*LIVE demonstracija auditorijoje*):

```
// Include Vector.h & Timer.h
int main() {
  int size = 1e7;
  Vector a(size);
  Vector b(size);
  Timer t;  // Paleisti timer'į
  Vector c;
  c = a + b; // a + b yra r-value!
  std::cout << size << " elementų sudėti užtruko: "
            << t.elapsed() << " s\n";
  return 0;
}

// 10000000 elementų sudėti užtruko: 0.259861 s
```

Benchmark'iname `Vector`ių kopijavimą (`operator+`) (2)

Copy semantikos (ne)efektyvumo tyrimas

Potencialiai atliekame tokius veiksmus:

1. iškviečiamas konstruktorius (su 1 parametru) sukonstruoti **a**
2. iškviečiamas konstruktorius (su 1 parametru) sukonstruoti **b**
3. iškviečiamas (default) konstruktorius sukonstruoti **c**
4. iškviečiamas `copy-konstruktorius` nukopijuot **a+b** į *temporary*.
5. iškviečiamas `copy-operator=` nukopijuot *temporary* į **c**.

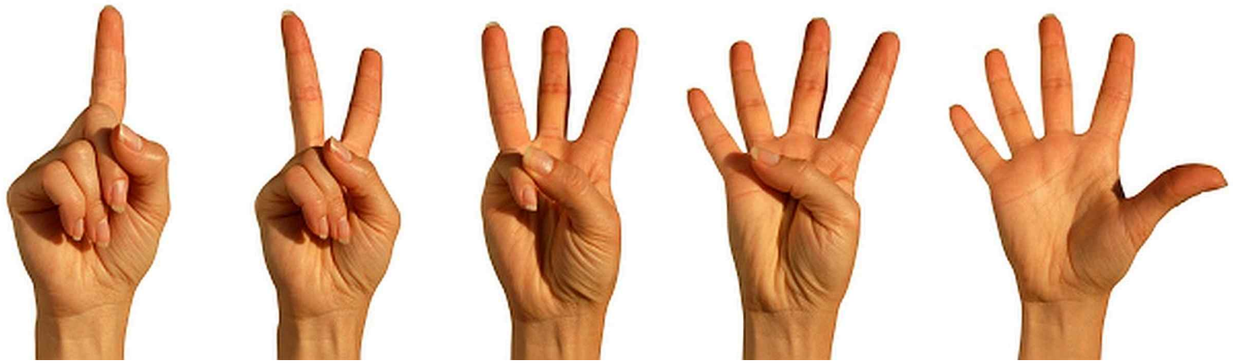
— "*Gudrūs*" kompiliatoriai išvengia 4 žingsnio, bet be šansų išvengti 5!

— Dar daugiau, sumos rezultato *temporary* mums daugiau nereikia.

Rule of five[r5] (nuo C++11)

In C++11 the rule of three can be broadened to the rule of five as C++11 implements move semantics, allowing destination objects to grab (or steal) data from temporary objects:

- destructor
- copy constructor
- **move constructor**
- copy assignment operator
- **move assignment operator**



[r5] https://en.wikipedia.org/wiki/Rule*of*three(*C%2B%2B*programming)

Copy vs move: konstruktorius ir operator=

```
class Vector {
  // ...
  Vector(const Vector& v);               // copy konstruktorius
  Vector& operator=(const Vector& v); // copy priskyrimas =

  Vector(Vector&& v);                    // move konstruktorius
  Vector& operator=(Vector&& v);      // move priskyrimas =
};
```

— Koks esminis skirtumas tarp copy ir move semantikos?

— Nėra const ir *move* semantinka naudoja *r-value* nuorodas!

Copy vs. move konstruktoriai

```
/* copy konstruktorius
   1. išskiria naują vietą
   2. perkopijuoja reikšmes iš vektoriaus v */
Vector(const Vector& v)
    : sz{v.sz},                // inicializuojame sz
      elem{new double[v.sz]}  // išskiriame atmintį elem
{
  for (int i = 0; i != sz; ++i)  // nukopijuojame elementus
    elem[i] = v.elem[i];
}

/* move konstruktorius
   1. "pavagiame" vektoriaus v duomenis
   2. priskiriame `nullptr` - saugiam sunaikinimui */
Vector(Vector&& v)
    : sz{v.sz},
      elem{v.elem}  // "pavok elementus" iš v
{
  v.elem = nullptr;  // v neturi jokių elementų
  v.sz = 0;
}
```

Move operator=

```
Vector& operator=(Vector&& v) {
  // Savęs priskyrimo aptikimas
  if (&v == this)
    return *this;
  delete[] elem;      // atlaisviname seną atmintį!
  elem = v.elem;      // elem point'ina į v.elem atmintį
  sz = v.sz;          // atnaujiname size
  v.elem = nullptr;   // v neturi jokių elementų
  v.sz = 0;
  return *this;       // grąžiname objektą
}
```

Benchmark'iname `Vector`ių perkėlimą (*move*)

Atlikus tą patį eksperimentą su pilnai realizuota "*Rule of five*":

```
// Include Vector.h su move konstrutoriumi ir move operator=
int main() {
  int size = 1e7;
  Vector a(size);
  Vector b(size);
  Timer t;  // Paleisti timer'į
  Vector c;
  c = a + b; // a + b yra r-value!
  std::cout << size << " elementų sudėti užtruko: "
            << t.elapsed() << " s\n";
  return 0;
}

// 10000000 elementų sudėti užtruko: 0.15689 s (su copy semantika: 0.259861 s)
```

Benchmark'iname Vectorių kopijavimą (1)

Realizuokime swap() funkciją (aka std::swap()), sukeičiančią 2-jų kintamųjų reikšmes:

```
template<class T>
void swap(T& x, T& y) {
  T temp { x }; // iškviečia copy-ctor
  x = y;        // iškviečia copy-assignment
  y = temp;     // iškviečia copy-assignment
}

int main() {
  int size = 1e7;
  Vector a(size);
  Vector b(size);
  Timer t;  // Paleisti
  swap(a,b);
  std::cout << "swap(a,b) užtruko: " << t.elapsed() << " s\n";
  // Swap(a,b) užtruko: 0.361689 s
}
```

Priverstinė *move* semantiką su `std::move`

— Ši versija atlieka 3 kopijas! Įsivaizduokime jei **T** objektai yra "*brangūs*".

— Tačiau čia pakaktų 3 kartus atlikti **move** operaciją, o ne **copy**!

— Problema ta, kad parametrai x ir y yra **l-value** nuorodos, todėl negalima iškviesti **move** konstruktoriaus ir **move** priskyrimo operatoriaus, vietoje **copy** konstruktoriaus ir priskyrimo operatoriaus.

— Čia mums gali pagelbėti C++11 funkcija: `std::move` (apibrėžta `<utility>` header'yje) - ji konvertuoja **l-value** į **r-value**.

— Beje, tokia swap funkcija jau yra realizuota: **std::swap**.

Benchmark'iname Vectorių kopijavimą (2)

```
template<class T>
void swap(T& x, T& y) {
  T temp { std::move(x) }; // iškviečia move-ctor
  x = std::move(y);        // iškviečia move-assignment
  y = std::move(temp);     // iškviečia move-assignment
}

int main() {
  int size = 1e7;
  Vector a(size);
  Vector b(size);
  Timer t;
  swap(a,b); // O kaip dėl std::swap funkcijos naudojimo čia?
  std::cout << "swap(a,b) užtruko: " << t.elapsed() << " s\n";
  // swap(a,b) užtruko: 1.3098e-05 s
}
```

Move semantikos realizacija Studentas klasei

```
class Studentas {
private:
  std::string vardas_;
  std::string pavarde_;
  double egzaminas_;
  std::vector<double> nd_; // gali būti didelis!
public:
  // ...
  Studentas(Studentas&& s) : // move c-tor
    vardas_{s.vardas_},
    pavarde_{s.pavarde_},
    egzaminas_{s.egzaminas_},
    // s yra vardinė r-value nuoroda, todėl l-value
    nd_{std::move(s.nd_)} // be std::move kviestų copy c-tor
  {
  }
};
```

Nenaudingo kopijavimo vengimas

Copy elision

Standartas leidžia kompiliatoriui elito kopijuoti arba perkelti c-tor-nepriklausomai nuo šalutinių poveikių! - šiais atvejais:

Klausimai !?