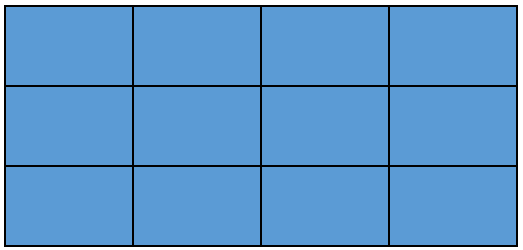
Класа матрице, конструктор премештања, паметни показивачи, нека својства основних типова

Матрице

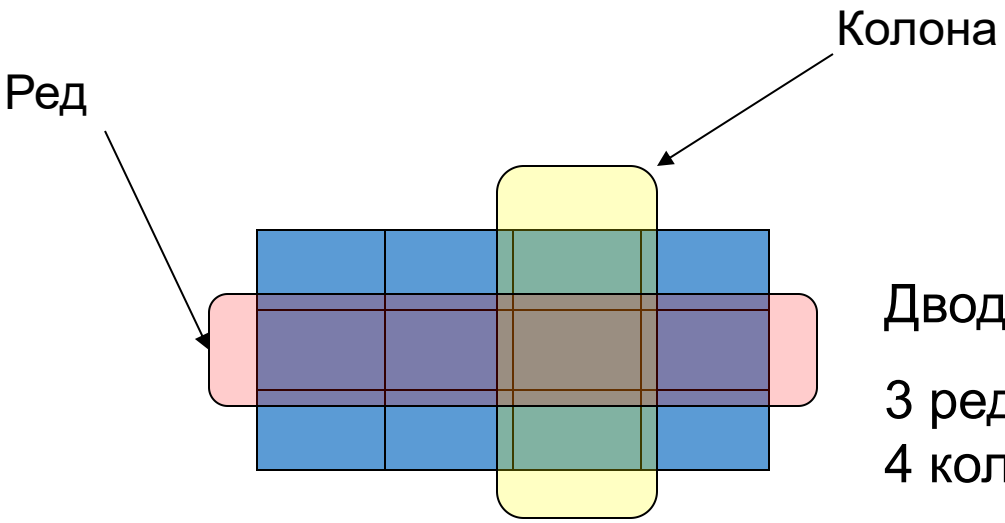
- Стандардни вектор (**vector**) и уграђени низ (тзв. „цеовски“) су једнодимензионални
- Шта ако нам требају две, или више димензија?
- Било би лепо да имамо нешто овако:



Matrix<int> m(3,4)
дводимензионална матрица, 3 пута 4

Матрице

- `Matrix<int> m(3, 4);`



Дводимензионална матрица 3 пута 4:

3 реда
4 колоне

Цеовски вишедимензионални низови

- Уграђена подршка

```
int ai[4];
double ad[3][4];
char ac[3][4][5];
ai[1] = 7;
ad[2][3] = 7.2;
ac[2][3][4] = 'c';
```

  - У суштини, низ низова

Цеовски вишедимензионални низови

- Проблеми
  - Фиксне величине (тј. морају бити познате током превођења)
    - Одређивање величине током извршавања захтева динамичко заузимање меморије
  - Компликовано преношење параметара и повратне вредности
    - Цеовски низ се своди на показивач на први елемент
  - Нема могућности провере опсега приликом приступа
    - Цеовски низ не зна своју величину
  - Не постоје операције које раде над низовима
    - Нема чак ни доделе (копирања)
- Један од **главних** извора багова

Цеовски вишедимензионални низови

- Проблем преношења цеовског низа као параметра постаје још израженији код вишедимензионалних низова

```
void f1(int a[3][5]);

void f1(int a[][5]);      // исто као горње: 3 се игнорише

void f2(int[ ][5], int dim1);  // прва димензија може варирати

void f3(int[ ][ ], int dim1, int dim2);   // грешка!

void f4(int* m, int dim1, int dim2)  // ово ради... али је мало чудно
{
  for (int i = 0; i < dim1; ++i)
    for (int j = 0; j < dim2; ++j)
      m[i*dim2+j] = 0;
}
```

Matrix библиотека

- Погледати у књизи поглавља 24.5 и 24.6

- На наредним слајдовима је представљена наша варијанта матрице, које је мало другачија (поједностављенија) од оне која је у књизи дата.

Matrix библиотека у књизи:

- Обавља провере током превођења и током извршавања
- Нуди матрице произвољних димензија
- Матрице су регуларне променљиве
  - Могу се нормално прослеђивати функцијама
- Уобичајене матричне операције
  - Индексирање: ( )
  - Обраћање делу матрице: [ ] и .slice()
  - Додела: =
  - Скалирање: +=, -=, *=, %=, итд.
  - Сабирање матрица
  - Векторске операције (нпр., **res[i] = a[i]*c+b[i]**)
  - Скаларни производ (res = sum of **a[i]*b[i]**)

- За општи тип матрице, ефикасност је подједнака коду писаном на нижем нивоу апстракције
- Библиотеку можете проширивати по потреби (нема никакве магије)

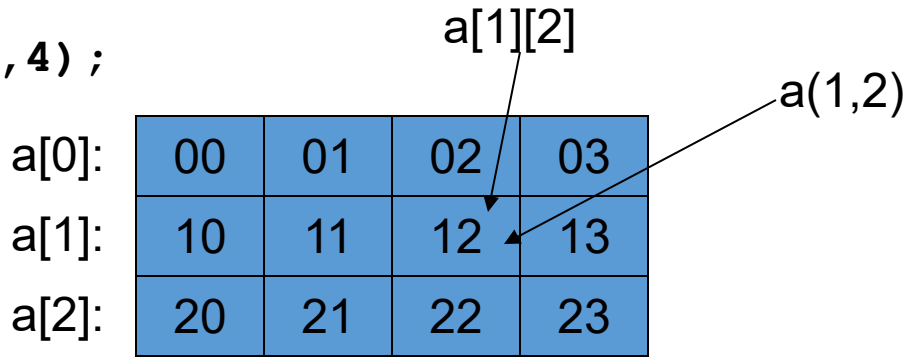
Наша Matrix библиотека:

- Обавља провере током превођења и током извршавања
- Нуди дводимензионалне матрице
- Матрице су регуларне променљиве
  - Могу се нормално прослеђивати функцијама
- Уобичајене матричне операције
  - Индексирање: ( )
  - Обраћање делу матрице: [ ] и .slice()
  - Додела: =
  - Скалирање: +=, -=, *=, %=, итд.
  - Сабирање матрица
  - Векторске операције (нпр., **res[i] = a[i]*c+b[i]**)
  - Скаларни производ (res = sum of **a[i]*b[i]**)
  - Испис на стандардни излаз и учитавање матрице из датотеке
- За општи тип матрице, ефикасност је подједнака коду писаном на нижем нивоу апстракције
- Библиотеку можете проширивати по потреби (нема никакве магије)

Matrix библиотека

- 2д матрицу индексирамо паром (ред, колона):

```
Matrix<int> a(3,4);
```



- Елементи су поређани у меморији један иза другог, **ред по ред** (могло би бити колона по колона, али није):

00	01	02	03	10	11	12	13	20	21	22	23

Matrix библиотека

```
void init(Matrix<int>& a)
{
  for (int i = 0; i < a.d1(); ++i)
    for (int j = 0; j < a.d2(); ++j)
      a(i, j) = 10*i+j;
}
```

```
void print(const Matrix<int>& a) // да ли се штампа по редовима или по колонама?
{
  for (int i = 0; i < a.d1(); ++i) {
    for (int j = 0; j < a.d2(); ++j) {
      cout << a(i, j) << '\t';
    }
    cout << '\n';
  }
}
```

Матрица

```
Matrix<int> a(10, 20);
a.size();          // број елемената
a.d1();            // број елемената у реду
a.d2();            // број елемената у колони
int* p = a.data();
a(i, j);           // (i, j)-ти елемент

Matrix<int> a2{a}; // a2 = a
a = a2;
a = a + a2;
```

Имплементација класе Matrix

- Желимо да имплементирамо матрицу чији елементи могу да буду било шта.

- Ево неколико покушаја... (погледати кодове уз ово предавање)

Матрица генерички у Јави

```
public class Matrix<T> {
      protected Object m_elem[];
      protected int m_sz;
      protected int m_d1;
      protected int m_d2;

      public Matrix(int d1, int d2) {
            m_elem = new Object[d1 * d2];
            m_sz = d1 * d2;
            m_d1 = d1;
            m_d2 = d2;
      }

      public T get(int i, int j) { return (T)m_elem[i * m_d1 + j]; }
      public void set(int i, int j, T e) { m_elem[i * m_d1 + j] = e; }

      public int d1() { return m_d1; }
      public int d2() { return m_d2; }
...
```

Матрица генерички у Јави

```
...
  public static <T> Matrix<T> add(Matrix<T> x, Matrix<T> y){
    Matrix<T> z = new Matrix<T>(x.d1(), x.d2());
    for (int i = 0; i < x.m_sz; ++i) {
      z.m_elem[i] = ??????????;
    }
    return z;
    }
}
```

Матрица генерички у Јави

```
...
  public static <T> Matrix<T> add(Matrix<T> x, Matrix<T> y){
    Matrix<T> z = new Matrix<T>(x.d1(), x.d2());
    for (int i = 0; i < x.m_sz; ++i) {
      z.m_elem[i] = (Integer)x.m_elem[i] + (Integer)y.m_elem[i];
    }
    return z;
    }
}
```

Резултати

- Це++, void*: 0,048
- Јава: 0,028
- Це++, шаблони: ???

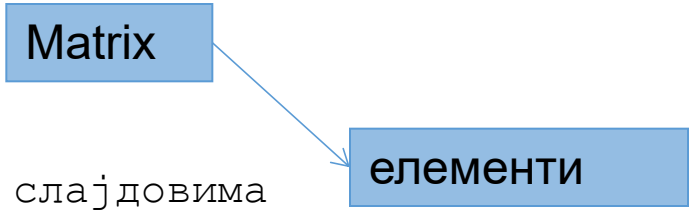
Резултати

- Це++, void*: 0,048
- Јава: 0,028
- Це++, шаблони: 0,0015

- >20 пута брже од void* и >10х брже од Јаве.

Имплементација класе Matrix

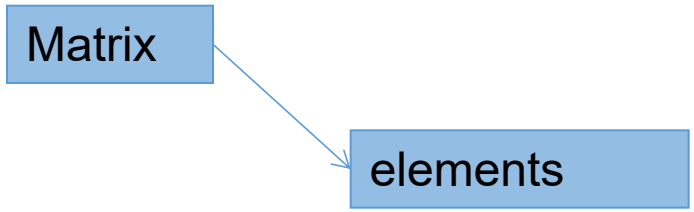
- Кроз Matrix променљиву се управља приступом елементима




```
template<class T> class Matrix {
public:
  // спрега, као што је описана на ранијим слајдовима
protected:
  T* m_elem; // елементи су поређани један за другим, као Цеовски низ
  const int m_sz;
  const int m_d1;
  const int m_d2; // број елемената по свакој димензији
};
```

Имплементација класе Matrix

- Руковање ресурсима:
  - Променљива типа Matrix мора руковати својим елементима
  - Конструктор заузима меморију за елементе и иницијализује их
  - Деструктор уништава елементе и ослобађа меморију која је за њих заузета
  - Додела копира елементе



Имплементација класе Matrix

- Требају нам конструктори:
  - Конструктор за подразумеване елементе
    - **`Matrix<T>::Matrix(Index, Index);`**
    - **`Matrix<int> m0(2, 3); //`** `сви елементи постављени на 0`
  - Конструктор са иницијализационом листом
    - **`Matrix<T>::Matrix(std::initializer_list<T>);`**
    - **`Matrix<int> m1 = {{1, 2, 3}, {4, 5, 6}};`**
  - Конструктор копије
    - **`Matrix<T>::Matrix(const Matrix<T>&);`**
    - **`Matrix<int> m2 = m1;`**

Преношење матрице

- Посматрајмо ову функцију:

```
Matrix operator+(Matrix a, Matrix b)
{
     Matrix res(a.d1(), a.d2());
     // Сабери матрице и резултат смести у res
     return res;
}

Matrix x, y, z;

z = x + y; // Колико пута се копирају матрице?
```

Преношење матрице

- Улазне матрице се беспотребно копирају
- Компајлер може оптимизовати код тако што ће уклонити беспотребна копирања, али се на то не можемо ослањати у општем случају.
- Решење за улазне параметре:

```
Matrix operator+(const Matrix& a, const Matrix& b)
```

- Шта са повратном вредношћу?

Преношење матрице

- Једна идеја:
  - Враћамо показивач на објекат заузет помоћу **new**

```
Matrix* operator+(const Matrix& a, const Matrix& b);
Matrix& z = *(x + y);
```

    - Проблеми:
    - Ружно на месту позива.
    - Ко зове **delete**?

Преношење матрице

- Друга идеја:
  - Враћамо референцу на објекат заузет помоћу **new**

```
Matrix& operator+(const Matrix& a, const Matrix& b);
Matrix& z = x + y;
```

    - Проблеми:
    - ~~Ружно на месту позива.~~
    - Ко зове **delete**?
      - Који **delete**? Где је овде показивач?

Преношење матрице

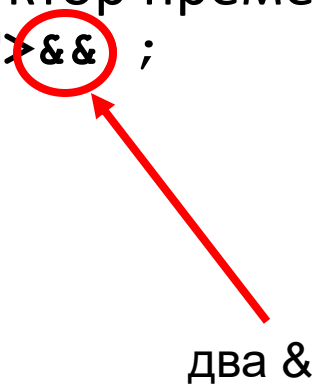
- Трећа идеја:
  - Прослеђујемо референцу на већ заузет објекат у који треба да се смести резултат

```
void operator+(const Matrix& a, const Matrix& b, Matrix& res);
Matrix res = x + y;
void plus(const Matrix& a, const Matrix& b, Matrix& res);
plus(x, y, res);
```

    - Проблеми:
    - Ружно на месту позива.
      - А и оператор сабирања прима само два параметра
    - ~~Ко зове **delete**?~~

Преношење матрице - потпуно решење

- Требају нам конструктори:
  - Конструктор за подразумеване елементе
    - `Matrix<T>::Matrix(Index,Index);`
    - `Matrix<int> m0(2,3);  // сви елементи постављени на 0`
  - Конструктор са иницијализационом листом
    - `Matrix<T>::Matrix(std::initializer_list<T>);`
    - `Matrix<int> m1 = {{1,2,3}, {4,5,6}};`
  - Конструктор копије
    - `Matrix<T>::Matrix(const Matrix<T>&);`
    - `Matrix<int> m2 = m1;`
  - Мув (move) конструктор (Конструктор премештања)
    - **`Matrix<T>::Matrix(Matrix<T>&& );`**
    - **`return m1;`**

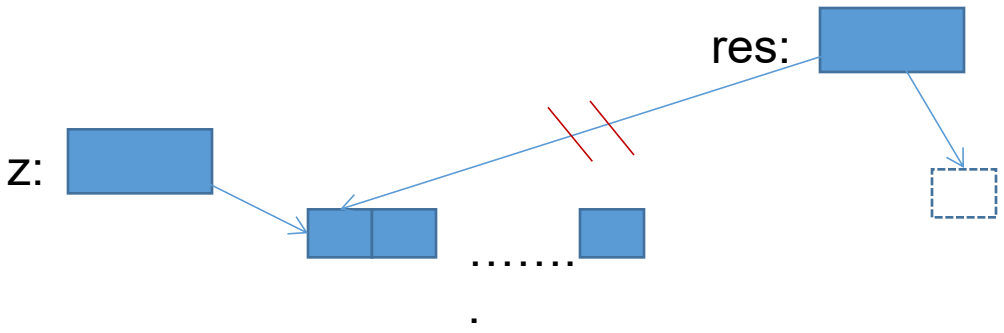


Мув семантика

- Функција враћа променљиву типа **Matrix**

```
Matrix operator+(const Matrix& a, const Matrix& b)
{
    Matrix res;
    // Сабери матрице и резултат смести у res
    return res;
}
Matrix z = x + y;
```

- Али се позива мув конструктор (конструктор премештања)
  - Нема копирања: само се „преузима репрезентација“

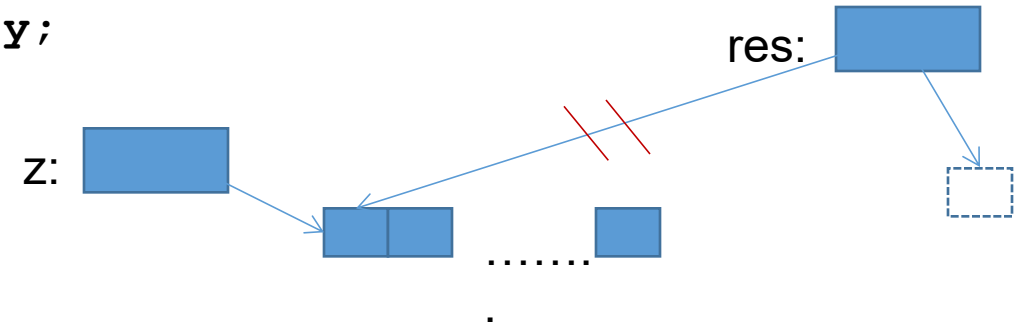


Мув семантика

- Мув конструктор

```
class Matrix {
        // …
      Matrix(Matrix&& a)
        : m_elem(a.m_elem),
          m_sz(a.m_sz),
          m_d1(a.m_d1),
          m_d2(a.m_d2)
      {
            a.m_elem = nullptr;
      }
};

Matrix z = x + y;
```



Мув семантика

- А може и мув оператор доделе

```
class Matrix {
        // …
      Matrix& operator=(Matrix&& a)
      {
            delete[] m_elem;
            m_elem = a.m_elem;
            a.m_elem = nullptr;
            m_sz = a.m_sz;
            m_d1 = a.m_d1;
            m_d2 = a.m_d2;

            return *this;
      }
};

Matrix z;
z = x + y;
```

Мув семантика

- Мув конструктор или мув додела се имплицитно позивају у следећа два случаја (суштински, када компајлер неоспорно зна да се животни век десног операнда завршава):

```
Matrix operator+(const Matrix& a, const Matrix& b) {
     Matrix res;
     // ... //
     return res; //<- Овде.
}

Matrix z = x + y; //<- Овде. Десни операнд је привремени
Matrix z{x + y};  // објекат, резултат сабирања, који живи
Matrix z;         // само до краја наредбе
z = x + y;
```

- Али можемо из позвати и експлицитно:

```
Matrix x, y;
x = std::move(y);
Matrix z{std::move(y)};
```

- Захтев је да десни објекат (y) остане у стању које дозвољава да се уништи или да му се додели нова вредност.

Мув семантика

- Када је std::move корисно:

```
void swap(T& a, T& b) {
  T tmp(a);
  a = b;
  b = tmp;
}
```

```
void swap(T& a, T& b) {
  T tmp(std::move(a));
  a = std::move(b); // нова вредност у a
  b = std::move(tmp); // нова вредност у b
} // деструктор tmp-a
```

Класе - подсећање

- Кључне операције:
  - Подразумевани конструктор (своди се на празан код)
    - Поништава се ако се декларише било који други конструктор
  - Конструктор копије (подразумевано се своди на копирање података)
  - Додела копије (подразумевано се своди на копирање података)
  - Деструктор (подразумевано се своди на празан код)

- Правило **тројке**: „Ако вам не одговара подразумевана верзија бар једне од ове три функције, онда вам највероватније не одговара подразумевана верзија ни једне од њих.“
- То јест: „Најчешће ћете дефинисати или све три функције, или ниједну“

Класе

- Кључне операције:
  - Подразумевани конструктор (своди се на празан код)
    - Поништава се ако се декларише било који други конструктор
  - Конструктор копије (подразумевано се своди на копирање података)
  - Додела копије (подразумевано се своди на копирање података)
  - Деструктор (подразумевано се своди на празан код)
  - Конструктор премештања (мув конструктор)
  - Додела премештањем (мув додела)

Ако мув верзија нема, онда ће бити позване верзије са копирањем.

- Правило **петице**: „Ако вам не одговара подразумевана верзија бар једне од ових пет функција, онда вам највероватније не одговара подразумевана верзија ни једне од њих.“
- То јест: „Најчешће ћете дефинисати или свих пет функција, или ниједну“

Показивачи и референце - подсећање

- На пример, референце је боље користити у следећим случајевима:
  - Пренос параметара

```
int foo(const int* x) {
  return *x + 1;
}
```

```
int foo(const int& x) {
  return x + 1;
}
```

Показивачи и референце - подсећање

- На пример, референце је боље користити у следећим случајевима:
  - Пренос параметара
  - Униформан приступ атрибуту класе

```
struct vector {
  int* at(int x) {
    return &elem[x];
  }
}

*v.at(i) = 5;
cout << *v.at(i);
```

```
struct vector {
  int& at(int x) {
    return elem[x];
  }
}

v.at(i) = 5;
cout << v.at(i);
```

Показивачи и референце - подсећање

- На пример, референце је боље користити у следећим случајевима:
  - Пренос параметара
  - Униформан приступ атрибуту класе
  - За краћи назив променљиве

```
if (x->clan1->clan2->clan3.get() == 5)
{
  x->clan1->clan2->clan3.set(8);
}

clan3Type* y = &(x->clan1->clan2->clan3);
if (y->get() == 5)
{
  y->set(8);
}

clan3Type& y = x->clan1->clan2->clan3;
if (y.get() == 5)
{
  y.set(8);
}
```

Показивачи и референце - подсећање

- На пример, референце је боље користити у следећим случајевима:
  - Пренос параметара
  - Униформан приступ атрибуту класе
  - За краћи назив променљиве
  - За везе између објеката које увек морају постојати

```
struct zavisnaKlasa{
  refKlasa* ptr; // мора увек показивати
                 // не неку инстанцу refKlasa типа
  zavisnaKlasa() : ptr(&globRefKlasa) {}
  zavisnaKlasa(int count) {} // грешка неће бити откривена
};


struct zavisnaKlasa{
  refKlasa& ptr; // мора увек показивати
                 // не неку инстанцу refKlasa типа
  zavisnaKlasa() : ptr(globRefKlasa) {}
  zavisnaKlasa(int count) {} // грешка у превођењу
};
```

Показивачи

- Још неки случајеви:
  - Када динамички заузимамо објекат зато што желимо да му животни век траје дуже од досега у којем је направљен. Референца овде не помаже.

```
mojTip* makeMojTip() {
  return new mojTip(1, 2, 3);
}

void foo() {
  //...
  mojTip* p = makeMojTip();
    // p је сада "власник" објекта
  //...
  // је ли било delete?
}
```

Паметни показивачи

- Још неки случајеви:
  - Када динамички заузимамо објекат зато што желимо да му животни век траје дуже од досега у којем је направљен. Референца овде не помаже.

```
std::unique_ptr<mojTip> makeMojTip() {
  return std::unique_ptr<mojTip>(new mojTip(1, 2, 3));
}

void foo() {
  //...
  std::unique_ptr<mojTip> p = makeMojTip();
    // p је сада "власник" објекта
  //...
  // не треба delete... јер ће бити позвано у деструктору p-а
}
```

Паметни показивачи

- Још неки случајеви:
  - Када динамички заузимамо објекат зато што желимо да му животни век траје дуже од досега у којем је направљен. Референца овде не помаже.

```
std::unique_ptr<mojTip> makeMojTip() {
  return std::make_unique<mojTip>(1, 2, 3);
}

void foo() {
  //...
  std::unique_ptr<mojTip> p = makeMojTip();
    // p је сада "власник" објекта
  //...
  // не треба delete... јер ће бити позвано у деструктору p-а
}
```

Паметни показивачи

- У <memory>
  - unique_ptr

```
struct mojTip {
  mojTip(int a, int b, int c) : x(a), y(b), z(c) {}
  int x, y, z;
};

void bar(mojTip& x);

void foo() {
  //...
  std::unique_ptr<mojTip> p = std::make_unique<mojTip>(1, 2, 3);
  // auto p = std::make_unique<mojTip>(1, 2, 3);
    // p је сада "власник" објекта
  //...
  cout << p->x;
  bar(*p);
  p++; // грешка!
  p[5]; // грешка!
}
```

Паметни показивачи

- У <memory>
  - unique_ptr

```
struct mojTip {
  mojTip(int a, int b, int c) : x(a), y(b), z(c) {}
  int x, y, z;
};

void zol(mojTip* x);

void foo() {
  //...
  std::unique_ptr<mojTip> p = std::make_unique<mojTip>(1, 2, 3);
    // p је сада "власник" објекта
  //...
  zol(p.get()); // али zol не сме звати delete!
  std::unique_ptr<mojTip> q{p}; // грешка! може бити само један
  q = p; // грешка!
  q = std::move(p); // p више није власник објекта
  mojTip* r = q.release(); // q више није власник објекта
}
```

Паметни показивачи

- Основни принцип RAII: „власништво“ над ресурсом (објектом који нема досег) доделити некој променљивој која има досег.

Паметни показивачи

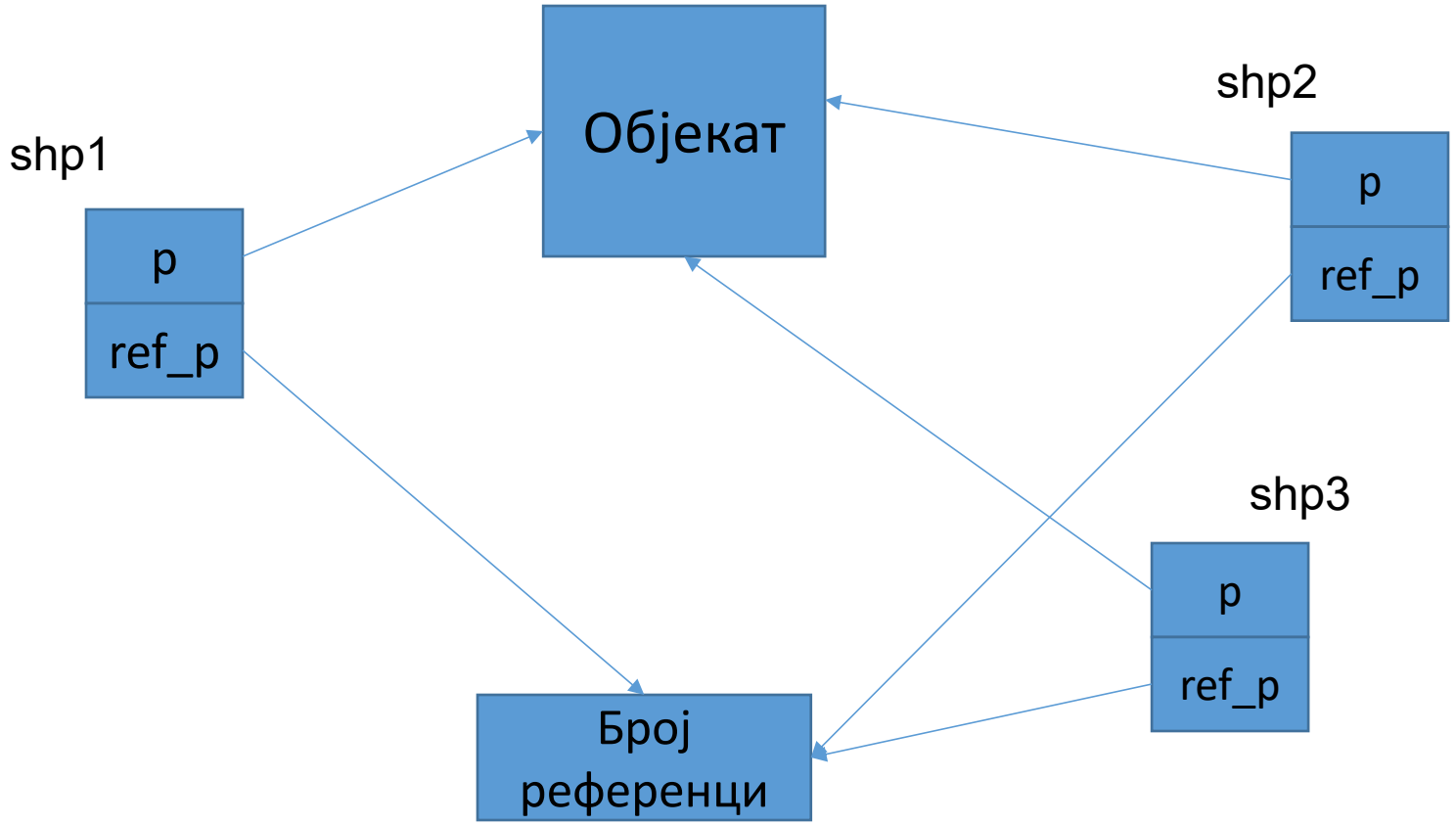
- У <memory>
  - unique_ptr
  - shared_ptr

- Шта ако имамо више показивача који показују на исти објекат?
- Ко је задужен за његово брисање? (Ко је власник?)
- Не постоји једноставан одговор на питање: „Ко показује на овај објекат?“

- У деструктору „дељеног показивача“ проверава се колико још других „дељених показивача“ показује на тај објекат. **delete** се позива тек ако је тај показивач последњи који показује на објекат.
- Та техника се зове бројање референци.
- Колико референци – толико „власника“.

Паметни показивачи

- У <memory>
  - unique_ptr
  - shared_ptr



Паметни показивачи

- У <memory>
  - unique_ptr
  - shared_ptr

```
void foo() {
  //...
  std::shared_ptr<mojTip> p = std::shared_pointer<mojTip>(
    new mojTip(1, 2, 3)); // 2 new, за објекат и за референце
  // број референци: 1 – p.use_count()
  std::shared_ptr<mojTip> q{p}; // број реф.: 2
  {
    std::shared_ptr<mojTip> r = {p}; // број реф.: 3
  } // број реф.: 2
  p = nullptr; // број реф.: 1
} // број реф.: 0 -> зове се delete
```

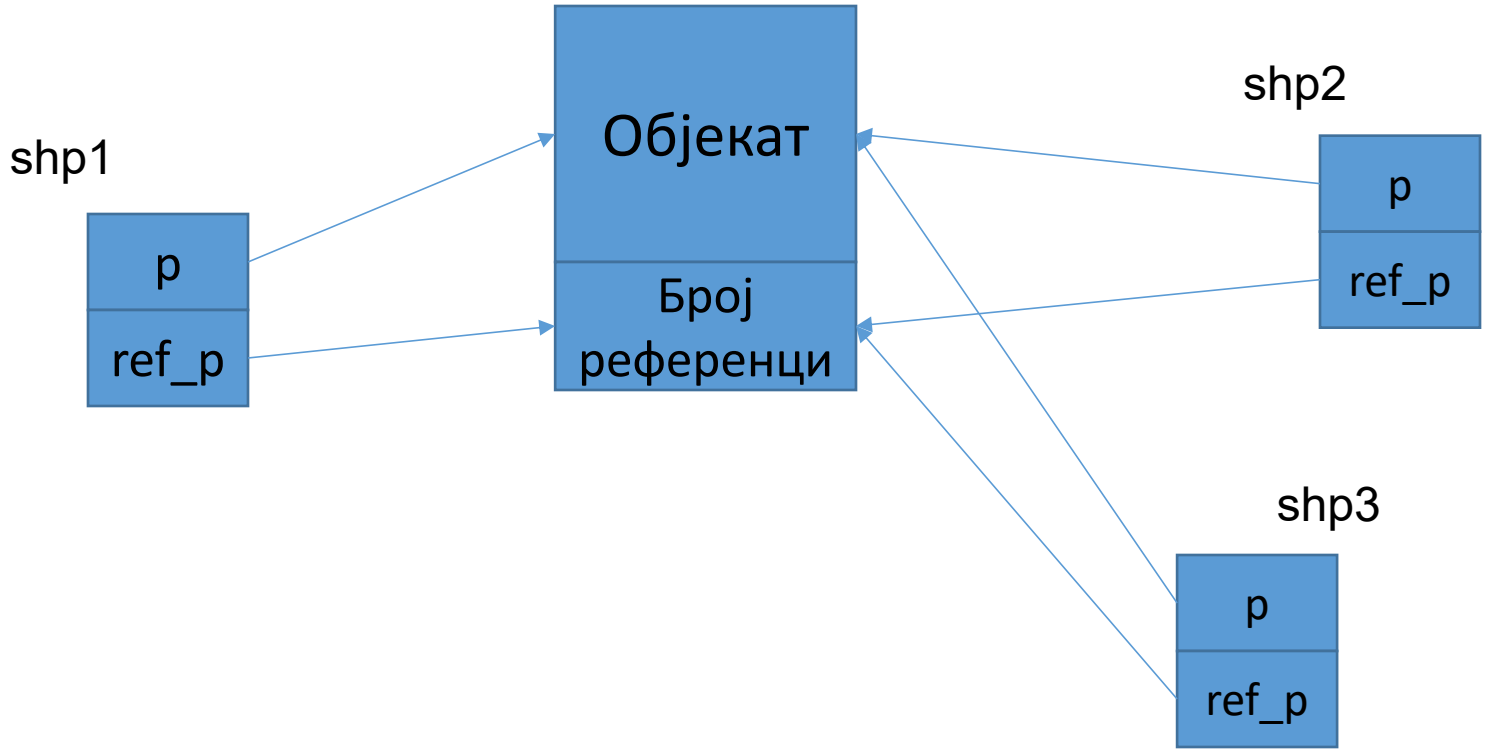
Паметни показивачи

- У <memory>
  - unique_ptr
  - shared_ptr

```
void foo() {
  //...
  std::shared_ptr<mojTip> p = std::make_shared<mojTip>(1, 2, 3);
  // број референци: 1 – p.use_count()
  std::shared_ptr<mojTip> q{p}; // број реф.: 2
  {
    std::shared_ptr<mojTip> r = {p}; // број реф.: 3
  } // број реф.: 2
  p = nullptr; // број реф.: 1
} // број реф.: 0 -> зове се delete
```

Паметни показивачи

- У <memory>
  - unique_ptr
  - shared_ptr



Како се везују референце

- & или && одређује са каквим објектима референца може да се инцијализује.
- То својство објекта је познато током превођења.

```
int x, y;

int& r = x; // <- иницијализација
void foo(int& ar) ...
foo(x); // int& ar = x; <- иницијализација
foo(y); // int& ar = y; <- иницијализација

int&& rr = x; // !!! грешка у превођењу
void bar(int&& arr) ...
bar(x); // int&& arr = x; !!! грешка у превођењу

foo(Nesto()); // int& ar = Nesto(); !!! грешка у превођењу
bar(Nesto()); // int&& arr = Nesto(); OK!
```