

Вектор

Вектор

- Може садржати произвољан број елемената
 - Чиме је величина ограничена?
- Број елемената се може мењати током времена
 - Нпр. помоћу **push_back()**
- Пример

```
vector<double> age(4);  
age[0] = .33;  
age[1] = 22.0;  
age[2] = 27.2;  
age[3] = 54.2;
```



Вектор

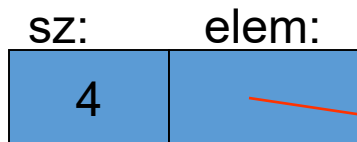
```
// вектор елемената типа double (као vector<double>):  
class vector  
{  
    int sz;          // броје елемената (size)  
    double* elem; // показивач на први елемент  
public:  
    vector(int s);  
    int size() const { return sz; }  
};
```

Животни век (трајност) променљиве

- За почетак животног века, тј. настајање променљиве, везано је заузимање ресурса.
- За завршетак је везано ослобађање ресурса (и видели смо на које све начине се то обавља).
- Међутим, при настајању променљиве можемо урадити још неке ствари.
- На пример, иницијализовати променљиву или још неке активности.
- То радимо кроз специјалну функцију коју називамо **конструктор**.

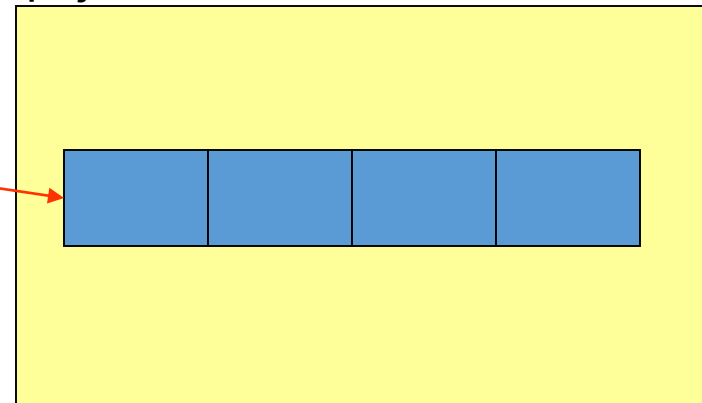
Вектор (конструктор)

```
vector::vector(int s)
: sz(s),
  elem(new double[s])
{
}
// new овде не иницијализује елементе
```



Показивач

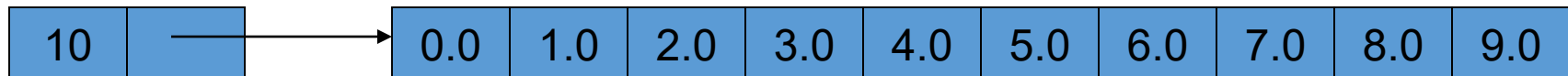
Слободна
меморија:



Вектор (конструктор и једноставан приступ)

```
class vector {  
    int sz;  
    double* elem;  
public:  
    vector(int s) : sz(s), elem(new double[s]) { }  
    double get(int n) const { return elem[n]; }  
    void set(int n, double v) { elem[n] = v; }  
    int size() const { return sz; }  
};
```

```
vector v(10);  
for (int i = 0; i < v.size(); ++i) {  
    v.set(i, i);  
    cout << v.get(i) << ' ' ;  
}
```



Вектор (деструктор)

- И нестајању променљиве можемо придружити још неке активности. То радимо кроз специјалну функцију коју називамо деструктор.

```
class vector
{
    int sz;
    double* elem;
public:
    vector(int s)
        : sz(s), elem(new double[s]) { }
    ~vector()
        { delete[] elem; }
    // ...
};
```

- **Ово је пример врло важне технике:**

- заузми ресурсе у конструктору
- ослободи их у деструктору
- Примери ресурса: меморија, датотеке, нити, сокети...

Бектор

```
void foo()  
{  
    double* elem = new double[50];  
    ...  
    throw some_exception();  
    ...  
    delete[] elem;  
}
```

```
void foo()  
{  
    std::vector<double> vec(50);  
    ...  
    throw some_exception();  
    ...  
}
```


Вектор

```
class vector
{
    int sz;
    double* elem;
public:
    vector(int s) : sz(s), elem(new double[s]) { }

    ~vector() { delete[] elem; }

    double get(int n) { return elem[n]; }
    void set(int n, double v) { elem[n] = v; }
    int size() const { return sz; }
};
```

Проблем

- Копирање не ради како бисмо очекивали

```
void f(int n)
{
    vector v(n);
    vector v2 = v; // ?
    // vector v2{v};

    vector v3;
    v3 = v;        // ?
    // v3 == v ?
    // ...
}
```

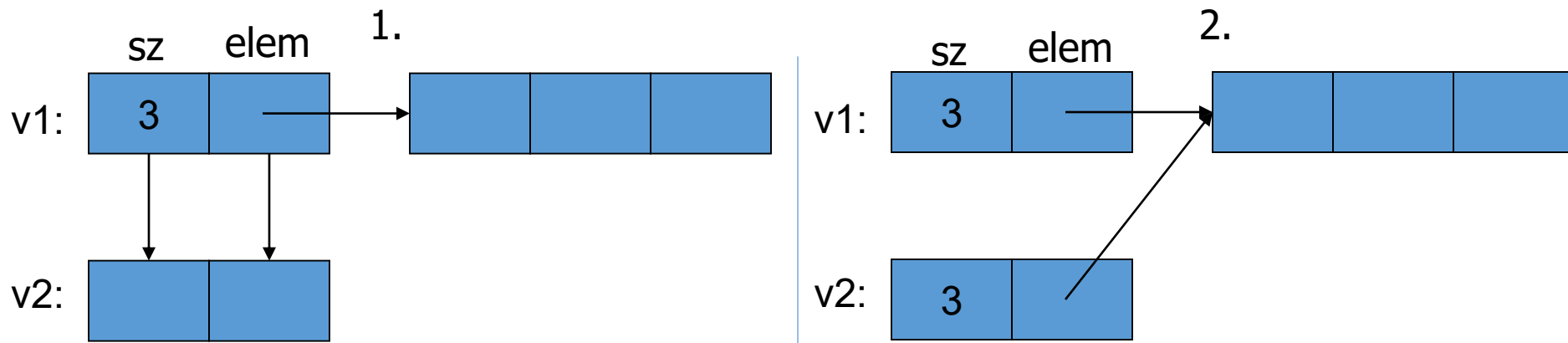
Подсећање: шта нам од операција треба

- 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)

```
class Token {  
public:  
    Token() {}  
    Token(const Token& x) : kind(x.kind), value(x.value) {}  
    Token& operator=(const Token& x) { kind = x.kind;  
        value = x.value; }  
    ~Token() {}  
    char kind;  
    double value;  
};
```

Подразумевано иницијализовање копијом

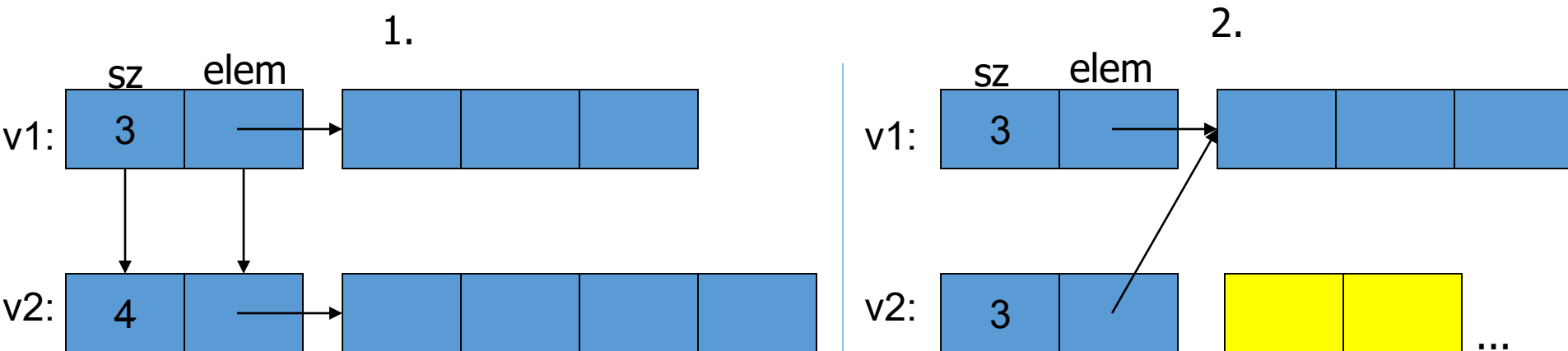
```
void f(int n)
{
    vector v1(n);
    vector v2 = v1; // иницијализација:
                    // подразумевано копира елементе класе
                    // дакле, sz и elem ће бити копирани
}
```



Проблем када напустимо f()! Зашто?

Подразумевана додела копије

```
void f(int n)
{
    vector v1(n);
    vector v2(4);
    v2 = v1; // додела:
              // подразумевано копира елементе класе
              // дакле, sz и elem ће бити копирани
}
```



Опет проблем када напустимо f()!

Сада имамо и цурење меморије.

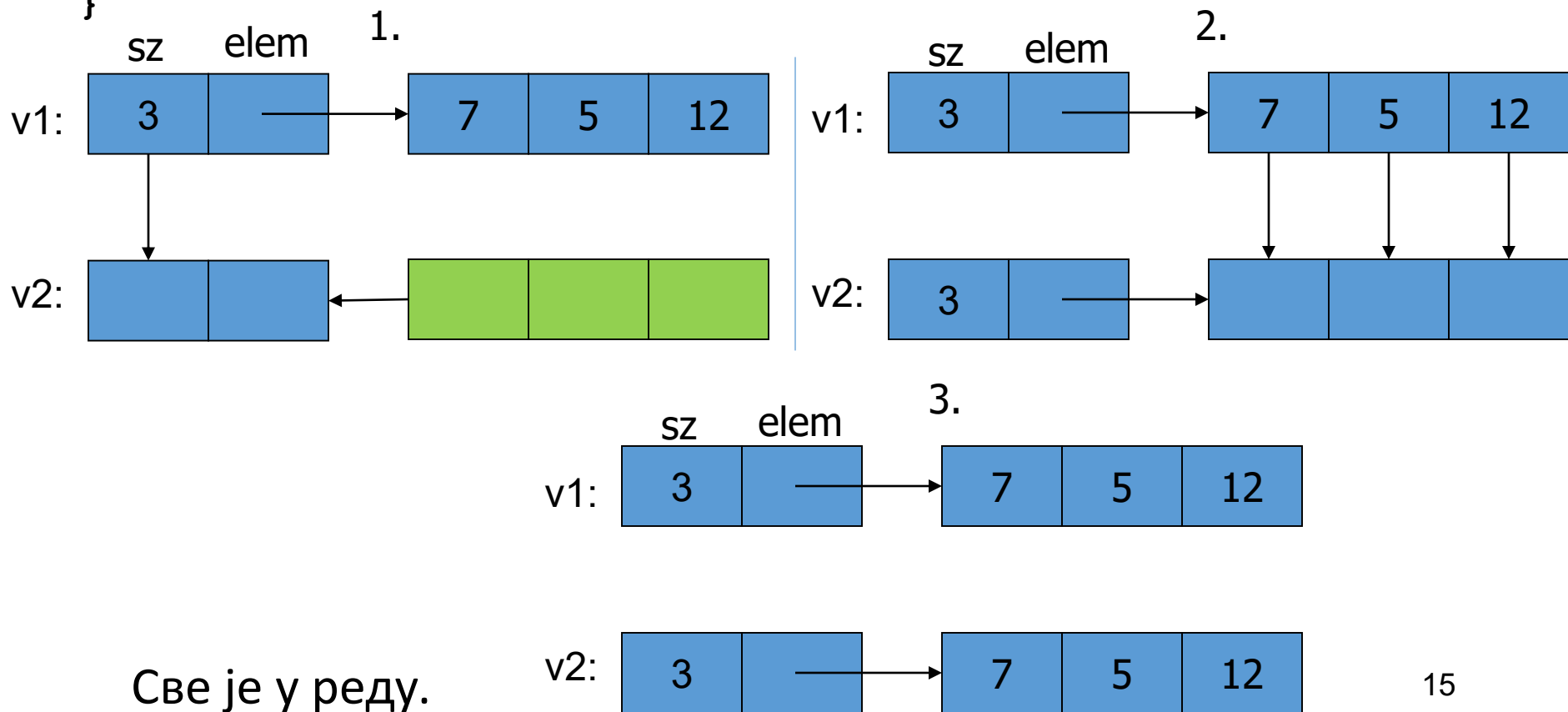
Конструктор копије (иницијализација)

```
class vector
{
    int sz;
    double* elem;
public:
    vector(const vector&);
    // ...
};

vector::vector(const vector& a)
    : sz(a.sz), elem(new double[a.sz])
{
    for (int i = 0; i < sz; ++i)
        elem[i] = a.elem[i];
}
```

Иницијализација помоћу конструктора копије

```
void f(int n)
{
    vector v1(n);
    vector v2 = v1;
}
```



Оператор доделе (додела копије)

```
class vector
{
    int sz;
    double* elem;
public:
    vector& operator=(const vector& a) ;
    // void operator=(const vector& a) ;
};
```

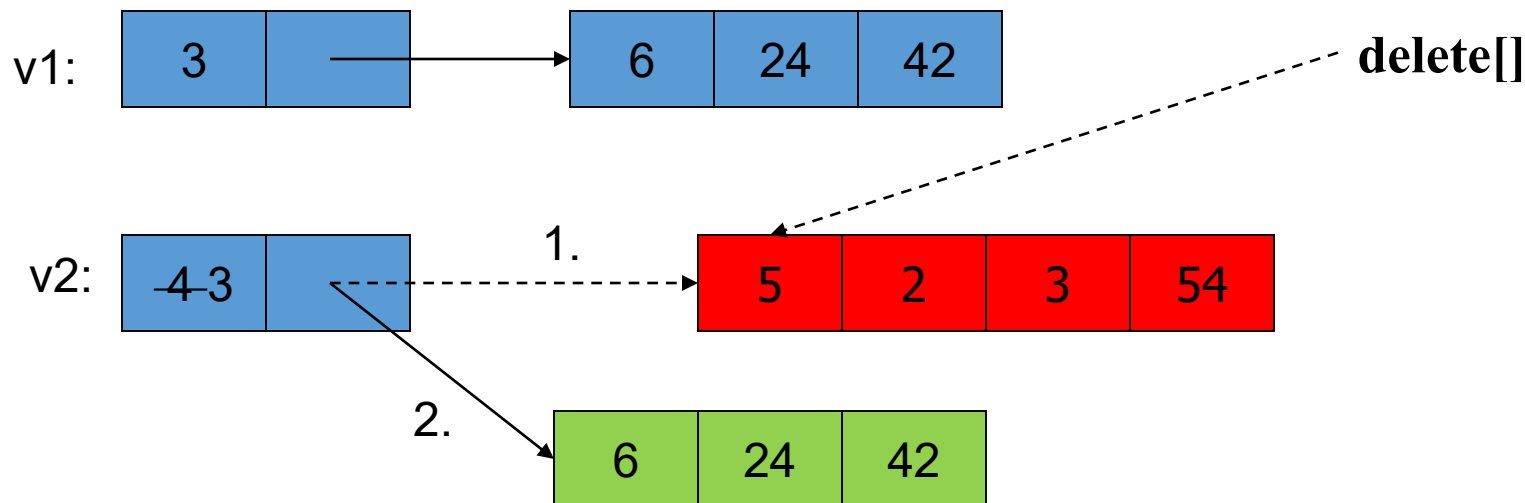
```
y = x = a;
// x = a;
```


Оператор доделе

```
vector& vector::operator=(const vector& a)  
    // као конструктор копије,  
    // али морамо руковати и старим елементима  
{  
    double* p = new double[a.sz];  
    for (int i = 0; i < a.sz; ++i)  
        p[i] = a.elem[i];  
  
    delete[] elem;  
  
    sz = a.sz;  
    elem = p;  
  
    return *this;  
}
```

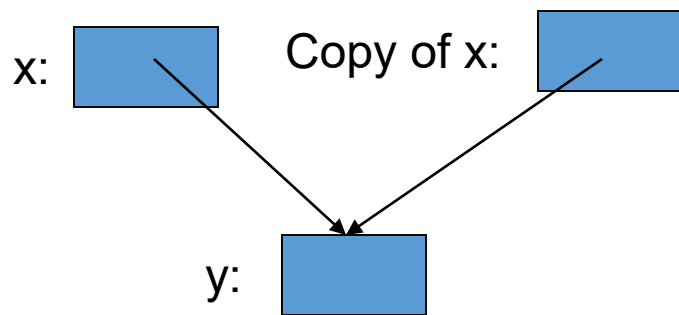
Копирање помоћу оператора доделе

```
void f(int n)
{
    vector v1(n);
    vector v2(4);
    v2 = v1;           // додела
}
```

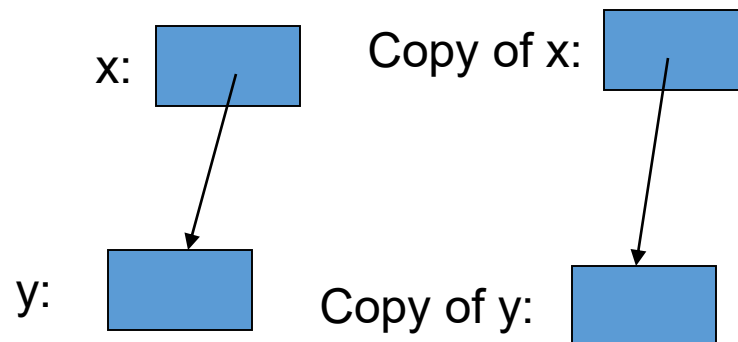


Терминологија у вези са копирањем

- Плитка копија: копира само показивач
- Дубока копија: копира оно на шта показивачи показују
 - То се дешава приликом копирања типова **vector**, **string**, итд.



Плитка копија

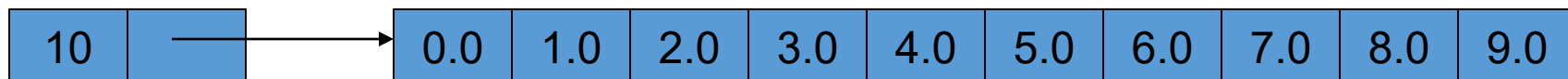


Дубока копија

Вектор (једноставан приступ)

```
vector v(10);  
for (int i = 0; i < v.size(); ++i) // ружњикаво  
{  
    v.set(i, i);  
    cout << v.get(i);  
}
```

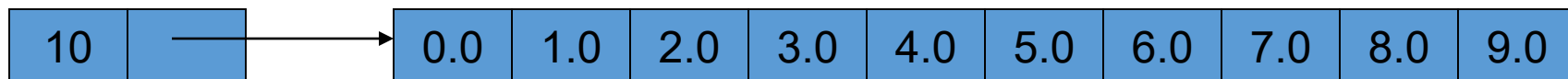
```
for (int i = 0; i < v.size(); ++i) // на ово смо навикли:  
{  
    v[i] = i;  
    cout << v[i];  
}
```



Вектор (преклапамо [] оператор)

```
class vector
{
    int sz;
    double* elem;
public:
    vector(int s) : sz(s), elem(new double[s]) { }
    // ...
    double operator[](int n) { return elem[n]; }
};
```

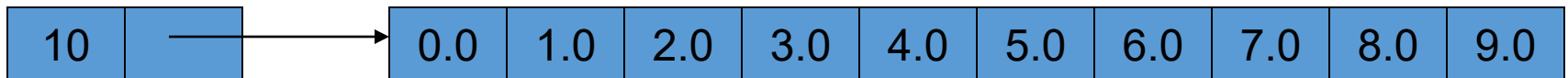
```
vector v(10);
for (int i = 0; i < v.size(); ++i) // ради само за читање
{
    v.set(i, i); ✗
    cout << v[i]; ✓
}
```



Вектор (могли бисмо користи показивач)

```
class vector
{
    int sz;
    double* elem;
public:
    vector(int s) : sz(s), elem(new double[s]) { }
    // ...
    double* operator[](int n) { return &elem[n]; }
};
```

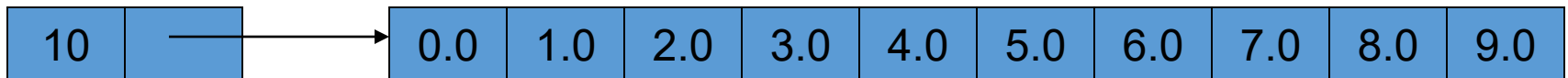
```
vector v(10);
for (int i = 0; i < v.size(); ++i) // ради, али и даље ружњикаво
{
    *v[i] = i; ~
    cout << *v[i]; ~
}
```



Вектор (користимо референцу)

```
class vector
{
    int sz;
    double* elem;
public:
    vector(int s) : sz(s), elem(new double[s]) { }
    // ...
    double& operator[](int n) { return elem[n]; }
};
```

```
vector v(10);
for (int i = 0; i < v.size(); ++i)
{
    v[i] = i; ✓
    cout << v[i]; ✓
}
```



Који алат употребити?

- Нож
- Јер може све!



Који алат употребити?

- А шта ћемо са овим?
- Сличан је однос показивача и референци.
- Као и још неких елемената језика...



Показивачи и референце

Важне особине показивача:

- Додела показивачу мења показивач, а не оно на шта он показује
- Адреса објекта (која се може сместити у показивач) се добија употребом унарног `&` или `new`
- За приступ објекту на који се показује користи се унарно `*`, `[]` или `->`
- Показивач може садржати нул вредност (`nullptr`), што означава да показивач не показује ни на шта.

Важне особине референце:

- Додела референци мења оно на шта референце показује, а не њу саму
- Референца се може доделити само приликом иницијализовања
- Приступ објекту кроз референцу не захтева никакву посебну синтаксу
- Референца увек нешто референцира (не може ништа да не референцира)

Показивачи и референце

- Показивач је променљива која садржи адресу неког објекта у меморији.
- Референцу можемо описати као „константан показивач који се имплицитно дереференцира“, или као „алтернативно име објекта“.
- И референца у многим случајевима се сведе на коришћење адреса у меморији (тј. понашање је испод хаубе је најчешће исто), али је се то дешава имплицитно.
- Показиваче можемо користити за све! (Као што и све можемо написати у асемблеру.)
- Али треба да их користимо јако ретко...
- Само у јако ограниченим блоковима кода, или специјалним функцијама, где је ефикасност критична, и где је врло мала могућност да настану негативне последице (јер је блок мали, па се целокупна употреба показивача може лако сагледати).
- У скоро свим другим употребама, предност треба давати језичким конструкцијама или елементима стандардне библиотеке који те случајеве боље покривају.

Показивачи и референце

- 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

- 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

- 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

- 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 и референце

- 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
- За све друге потребе користите експлицитну индирекцију, тј. показиваче (за сада...).