Наставак вектора

Промена величине вектора

```
vector v(n); // v.size() == n
```

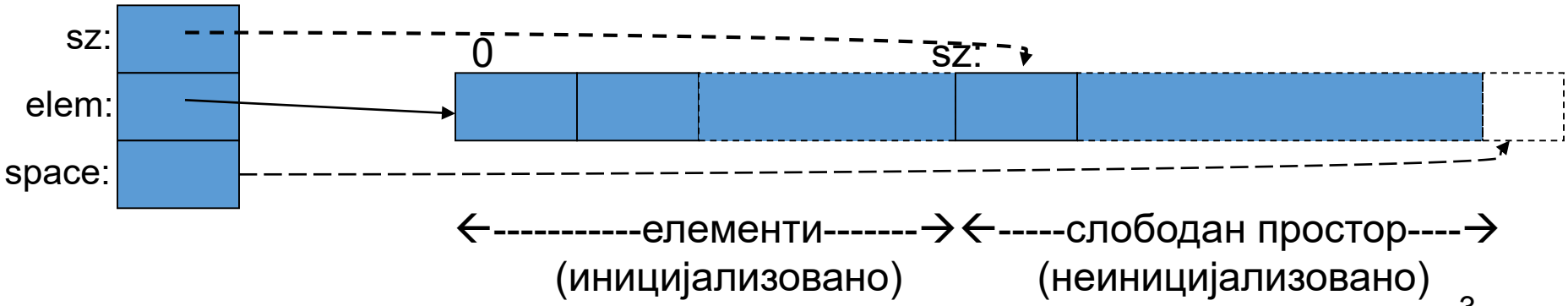
Величину можемо променити на три начина

- Директно
  - **`v.resize(10);`**
- Додавањем елемента
  - **`v.push_back(7);`**
- Доделом
  - **`v = v2; // v`** је копија **`v2`**
    `// v.size()` је сада једнако **`v2.size()`**

Представа вектора

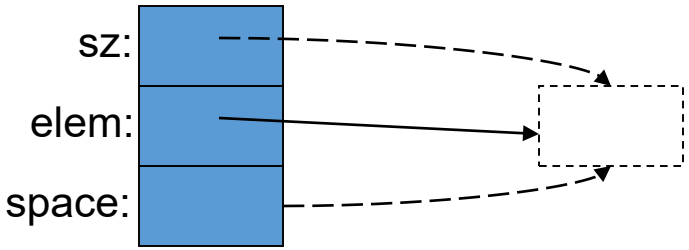
- Претпоставка (која можда није увек тачна, али верујемо да је већином тачна): уколико користите **resize()** или **push_back()** једном, вероватно ћете то урадити поново

```
class vector
{
  int sz;
  double* elem;
  int space;    // број елемената, плус „слободан простор“

public:
  // …
};
```

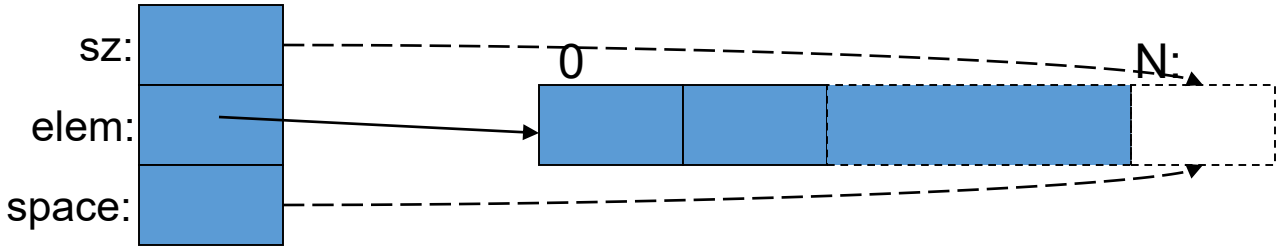


Представа вектора

- Празан вектор:



- `vector(N)` (нема празног простора):



vector::reserve()

- Метода за експлицитно заузимање меморије
  - **`reserve()`** не мења величину или вредности елемената

```
void vector::reserve(int newalloc)
{
 if (newalloc <= space)
   return;

 double* p = new double[newalloc];
 for (int i = 0; i < sz; ++i)
   p[i] = elem[i];

 delete[] elem;

 elem = p;
 space = newalloc;
}
```

vector::resize()

- Тривијална имплементација на основу методе **reserve()**

```
void vector::resize(int newsize)
{
  reserve(newsize);
  for(int i = sz; i < newsize; ++i)
    elem[i] = 0;
  sz = newsize;
}
```

vector::push_back()

```
void vector::push_back(double d)
{
  reserve(sz + 1);

  elem[sz] = d;
  ++sz;
}
```

vector::push_back()

```
void vector::push_back(double d)
{
 if (sz == 0)
     reserve(8);
 else if (sz == space)
     reserve(2 * space);

 elem[sz] = d;
 ++sz;
}
```

resize() и push_back()

```
class vector {
  int sz;
  double* elem;
  int space;
public:
  vector() : sz(0), elem(nullptr), space(0) { }
  explicit vector(int s) : sz(s), elem(new double[s]), space(s) { }
  vector(const vector&);
  vector& operator=(const vector&);
  ~vector() { delete[] elem; }

  double& operator[](int n) { return elem[n]; }
  int size() const { return sz; }

  void resize(int newsize);
  void push_back(double d);

  void reserve(int newalloc);
  int capacity() const { return space; }
};
```

resize() и push_back()

```
class vector {
  int sz = 0;
  double* elem = nullptr;
  int space = 0;
public:
  vector() { }
  explicit vector(int s) : sz(s), elem(new double[s]), space(s) { }
  vector(const vector&);
  vector& operator=(const vector&);
  ~vector() { delete[] elem; }

  double& operator[](int n) { return elem[n]; }
  int size() const { return sz; }

  void resize(int newsize);
  void push_back(double d);

  void reserve(int newalloc);
  int capacity() const { return space; }
};
```

resize() и push_back()

```
class vector {
  int sz = 0;
  double* elem = nullptr;
  int space = 0;
public:
  vector() { }
  explicit vector(int s) : sz(s), elem(new double[s]), space(s) { }
  vector(const vector&);
  vector& operator=(const vector&);
  ~vector() { delete[] elem; }

  double& operator[](int n) { return elem[n]; }
  int size() const { return sz; }

  void resize(int newsize);
  void push_back(double d);

  void reserve(int newalloc);
  int capacity() const { return space; }
};
```

explicit

```
struct mint {
    mint();
    mint(int x);
    ...
};

mint fib(mint n);

void main() {
    mint x, y;
    int a;
    ...
    x = fib(300_mi);
    x = fib(y);
    x = fib(300);
    x = fib(a);
}
```

explicit

```
struct mint {
    mint();
    explicit mint(int x);
    ...
};

mint fib(mint n);

void main() {
    mint x, y;
    int a;
    ...
    x = fib(300_mi);
    x = fib(y);
    x = fib(300);
    x = fib(a);
}
```

explicit

```
struct mint {
    mint();
    explicit mint(int x);
    ...
};

mint fib(mint n);

void main() {
    mint x, y;
    int a;
    ...
    x = fib(300_mi);
    x = fib(y);
    x = fib(mint(300)); // или static_cast<mint>(300) или, ређе, (mint)300
    x = fib(mint(a)); // или static_cast<mint>(a) или, ређе, (mint)a
}
```

explicit

```
class vector {
  int sz = 0;
  double* elem = nullptr;
  int space = 0;
public:
  vector() { }
  /*explicit*/ vector(int s) : sz(s), elem(new double[s]), space(s) { }
  ...
};

double sumOfElements(vector x);


double a = sumOfElements(5); // ???
```

explicit

```
struct mint {
    mint();
    explicit mint(int x); // int -> mint
    operator int() const; // mint -> int
    ...
};

mint fib(mint n);

void main() {
    mint x, y;
    int a, b;
    ...
    b = fib(300_mi);
    b = fib(y);
    b = fib(300);
    b = fib(a);
}
```

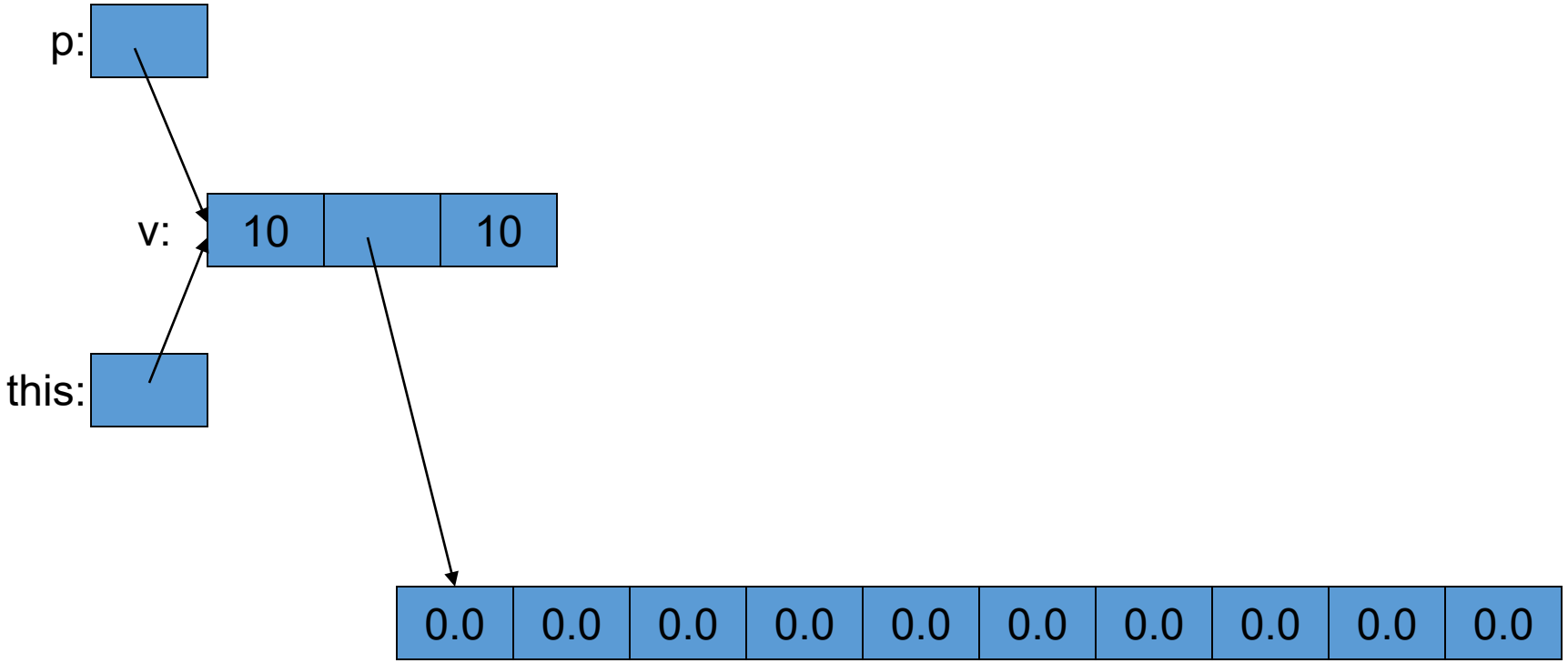
Додела

```
vector& vector::operator=(const vector& a)
{
 // Копирај
 double* p = new double[a.sz];
 for (int i = 0; i < a.sz; ++i)
   p[i] = a.elem[i];

 delete[] elem;

 // Замени
 sz = a.sz;
 elem = p;

 return *this;
}
```

this показивач

- `vector v(10);`
- `vector* p = &v;`

- У неким случајевима потребно је да се из метода класе обрати показивачу на тај објекат.
  - Назив тог „показивача на себе“ је **this**



this показивач

```
vector& vector::operator=(const vector& a)
{
  // …
  return *this; // уобичајено је да додела враћа референцу на себе
                // то је потребно за уланчавање додела
}

void f(vector v1, vector v2, vector v3)
{
  // …
  v1 = v2 = v3;
  // …
}
```

Додела

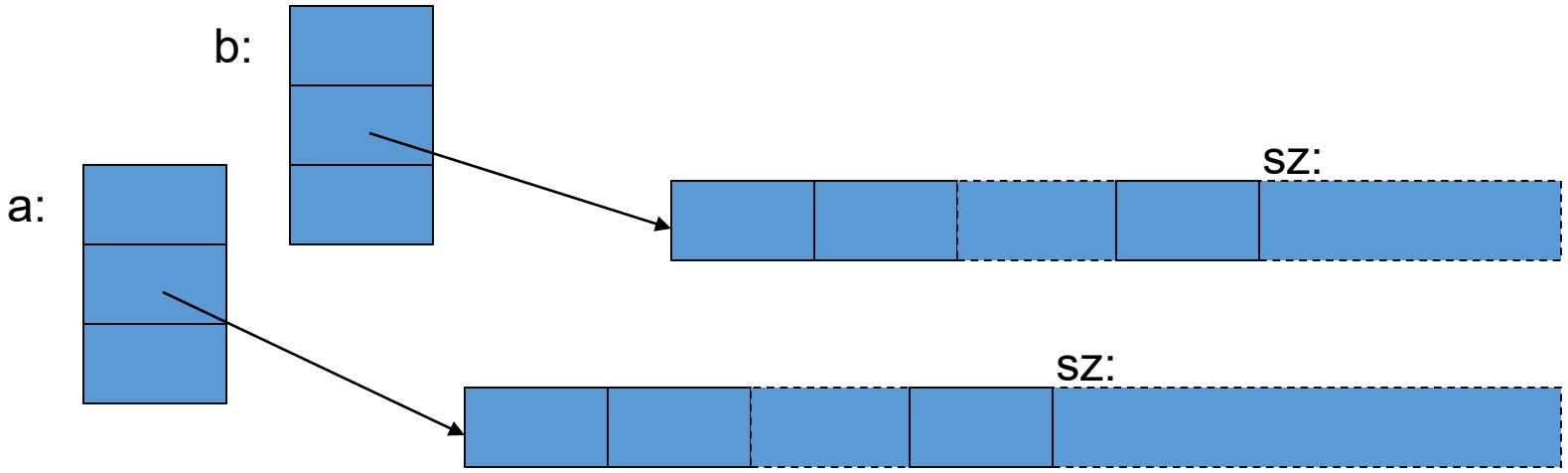
```
vector& vector::operator=(const vector& a)
{ // Код са експлицитним this->, али то није уобичајено
  // Копирај
  double* p = new double[a.sz];
  for (int i = 0; i < a.sz; ++i)
    p[i] = a.elem[i];

  delete[] this->elem;

  // Замени
  this->sz = a.sz;
  this->elem = p;

  return *this;
}
```

Додела

```
vector& vector::operator=(const vector& a)
{
  delete[] elem; // Зашто није добро да буде овде?

  // Копирај
  double* p = new double[a.sz];
  for (int i = 0; i < a.sz; ++i)
    p[i] = a.elem[i];

  delete[] elem;

  // Замени
  sz = a.sz;
  elem = p;

  return *this;
}
```

Оптимизована додела

- „Копирај и замени“ је добар општи концепт
  - али није увек најефикаснији
  - Шта ако у одредишном вектору већ има довољно места?
    - Онда напросто ископирај.
  - Шта ако вектор додељујемо самом себи?
    - Онда не ради ништа.



Оптимизована додела

```
vector& vector::operator=(const vector& a) {
  if (this == &a) return *this;

  if (a.sz <= space) {
    for (int i = 0; i < a.sz; ++i)
      elem[i] = a.elem[i];
    sz = a.sz;
    return *this;
  }

  double* p = new double[a.sz];
  for (int i = 0; i < a.sz; ++i)
    p[i] = a.elem[i];
  delete[ ] elem;
  sz = a.sz;
  space = a.sz;
  elem = p;
  return *this;
}
```

Вектор чији су елементи другог типа

- До сада смо свашта научили што нам је потребно за изградњу вектора:
  - Функције
  - Класе, кориснички типови
  - Показивачи и низови
  - Трајност (животни век)
  - Досег
  - Конструктор, деструктор
  - Динамичко заузимање меморије (new и delete)
  - Преклапање оператора
  - Референце
  - Изузеци
  - ...

Вектор чији су елементи другог типа

- До сада смо свашта научили што нам је потребно за изградњу вектора:
  - Функције
  - Класе, кориснички типови
  - Показивачи и низови
  - Трајност (животни век)
  - Досег
  - Конструктор, деструктор
  - Динамичко заузимање меморије (new и delete)
  - Преклапање оператора
  - Референце
  - Изузеци
  - ...
  - **+ Шаблони („темплејти“)**

vector_of_double

```
class vector_of_double
{
  int sz;
  double* elem;
  int space;
public:
  vector() : sz(0), elem(0), space(0) {}
  explicit vector(int s)
    : sz(s), elem(new double[s]), space(s) {}
  vector(const vector&);
  vector& operator=(const vector&);
  ~vector() { delete[] elem; }

  double& operator[](int n) { return elem[n]; }
  int size() const { return sz; }

  // …
};
```

vector_of_char

```
class vector_of_char
{
  int sz;
  char* elem;
  int space;
public:
  vector() : sz(0), elem(0), space(0) {}
  explicit vector(int s)
    : sz(s), elem(new char[s]), space(s) {}
  vector(const vector&);
  vector& operator=(const vector&);
  ~vector() { delete[] elem; }

  char& operator[](int n) { return elem[n]; }
  int size() const { return sz; }

  // …
};
```

Неформални шаблон класе вектор

```
class Name
{
  int sz;
  T* elem;
  int space;
public:
  vector() : sz(0), elem(0), space(0) {}
  explicit vector(int s)
    : sz(s), elem(new T[s]), space(s) {}
  vector(const vector&);
  vector& operator=(const vector&);
  ~vector() { delete[] elem; }

  T& operator[](int n) { return elem[n]; }
  int size() const { return sz; }

  // …
};
```

Неформални шаблон класе вектор

- Када имамо шаблон класе вектор онда можемо применити следећу логику:
- Ако ми треба променљива типа вектор чији су елементи **double**:
  - Ископирам шаблон
  - Заменим свако **T** са **double**
  - Дам класи (типу) неки јединствени назив
  - Ставим да је споменута променљива тог типа
- Ако ми треба променљива типа вектор чији су елементи **char**:
  - Ископирам шаблон
  - Заменим свако **T** са **char**
  - Дам класи (типу) неки јединствени назив
  - Ставим да је споменута променљива тог типа
- Ако ми треба још једна променљива типа вектор чији су елементи **double**:
  - Пошто сам већ једном направио жељени тип по шаблону, сада могу одмах да направим само променљиву тог типа.

Формални шаблон класе вектор

```
template<typename /*или class*/ T> class vector
{
  int sz;
  T* elem;
  int space;
public:
  vector() : sz(0), elem(0), space(0) {}
  explicit vector(int s)
    : sz(s), elem(new T[s]), space(s) {}
  vector(const vector&);
  vector& operator=(const vector&);
  ~vector() { delete[] elem; }

  T& operator[](int n) { return elem[n]; }
  int size() const { return sz; }

  // …
};
```

Параметризација са типом елемената

```
template<typename T> class vector
{
  // …
};

vector<double> vd;
vector<int> vi;
vector<vector<int>> vvi;
vector<char> vc;
vector<double*> vpd;
vector<vector<double>*> vvpd;
```

Формални шаблон класе вектор

- Механизам темплејта у Це++-у обавља исте активности као код неформалног шаблона, само што се то дешава аутоматски.

```
vector<double> x;
vector<char> y;
vector<double> z;
...
```

Формални шаблон класе вектор

- Механизам темплејта у Це++-у обавља исте активности као код неформалног шаблона, само што се то дешава аутоматски.

```
class vector<double> {
  ...
  double* elem;
  double& operator[](int n) { return elem[n]; }
  ...
};
class vector<char> {
  ...
  char* elem;
  char& operator[](int n) { return elem[n]; }
  ...
};
...
vector<double> x;
vector<char> y;
vector<double> z;
...
```

Темплејти (шаблони)

- Основа генеричког програмирања у Це++-у
  - Велика флексибилност и перформансе
- До сада смо видели да су функција и класа два важна појма у Це++-у
- И једно и друго може бити направљено на основу шаблона.
- Дефиниција шаблона класе:

```
template<class T, int N> class Buffer { /* … */ };
```

- Дефиниција шаблона функције:

```
template<class T, int N> void foo(T x) { /* … */ };
```

- Специјализација (инстанцирање)

```
// за темплејт класе:
Buffer<char, 1024> buf; // за buf, T је char и N је 1024
// за темплејт функције:
foo<char, 1024>(c); // за foo(), T је char и N је 1024
```

// мада, функције имају једну посебност код инстанцирања коју ћемо споменути за неколико слајдова, тако да ћете ретко виђати овакво инстанцирање

Темплејти (шаблони)

- Постоје још и
  - шаблони променљивих
- Као и
  - шаблони надимака (алијаса), тј. других назива типове
    - То је оно са употребом кључне using, нпр.: **using Tip = int;**
- Али о томе неки други пут...

Појмовник

- Најбоље је прихватити овакву употребу појмова:
  - Исправно: шаблон класе, class template, темплејт класе, класни шаблон
  - Неисправно: шаблонска класа, template class, тамплејт класа
  - Исто је и за шаблон функције.
  - „vector је шаблон класе.“ – исправно.
  - „vector је шаблонска класа.“ – неисправно.

- Врло често ћете наилазити на неисправну употребу појмова. Није само по себи страшно, али може подсвесно створити погрешну слику о томе шта се заправо дешава.

- Сам шаблон није класа, или функција, већ се по том шаблону могу правити класе, или функције (множина!)
- У том смислу, појмови као што су „шаблонска класа“ могу се употребити да означе појединачну инстанцу шаблона („класа која је настала од шаблона“), нпр. „vector<int> је шаблонска класа“.

Темплејти (шаблони)

- Проблеми
  - Лоше пријављивање грешака
    - Али доста се поправило и поправља се
  - Касно пријављивање грешака
    - Некада тек током повезивања
  - Сви шаблони морају бити дефинисани у свакој јединици превођења (у свакој датотеци)
    - Могући екстерни шаблони од Це++11
    - Дефиниција шаблона мора бити у заглављу
- Препорука
  - Са разумевањем користите библиотеке шаблона
    - рецимо Це++ стандардна библиотека
      - *Нпр.,* **vector**, **sort()**
  - У почетку пишите само једноставне шаблоне
    - док не стекнете више искуства

Провера опсега

```
struct out_of_range { /* … */ };

template<class T> class vector
{
  // …
  T& operator[](int n);
  // …
};

template<class T> T& vector<T>::operator[](int n)
{
  if (n < 0 || sz <= n) throw out_of_range();
  return elem[n];
}
```

Провера опсега

```
void fill_vec(vector<int>& v, int n)
{
  for (int i = 0; i < n; ++i) v.push_back(factorial(i));
}

int main()
{
  vector<int> v;
  try {
      fill_vec(v, 10);
      for (int i = 0; i <= v.size(); ++i)
            cout << "v[" << i << "]==" << v[i] << '\n';
  }
  catch (out_of_range) {          // дођи ћемо овде... зашто?
      cout << "out of range error";
      return 1;
  }
}
```

Руковање изузецима (примитивно)

```
vector<int>* some_function()
{
 vector<int>* p = new vector<int>;

 try {
     fill_vec(*p, 10);
     // …
     return p;
 }
 catch (…) {
     delete p;
     throw;
 }
}
```

Руковање изузецима

```
// Када користимо промењљиве аутоматске трајности
// ослобађање ресурса је аутоматско (кроз деструктор)

vector<int> glob;

void some_other_function()
{
  vector<int> v;

  fill_vec(v, 10);
  // ...
  fill_vec(glob, 10);
  // …
}
```

Шта стандард гарантује

```
// не гарантује проверу опсега у функцији operator[ ]
// шта више, обично је нема:
template<class T> class vector {
  // …
  T& at(int n);        // гарантује проверу опсега
  T& operator[](int n);
};

template<class T>  T& vector<T>::at (int n)
{
  if (n<0 || sz<=n) throw out_of_range();
  return elem[n];
}

template<class T>  T& vector<T>::operator[](int n)
{
  return elem[n];
}
```

Приступ const вектору

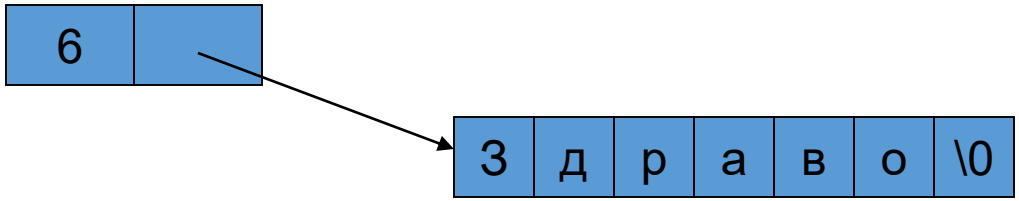
```
template<class T> class vector {
  // …
  T& at(int n);


  T& operator[ ](int n);

  // …
};

void f(const vector<double> cvd, vector<double> vd)
{
  // …
  double d1 = cvd[7];
  double d2 = vd[7];
  cvd[7] = 9;
  vd[7] = 9;
}
```

Приступ const вектору

```
template<class T> class vector {
  // …
  T& at(int n);
  const T& at(int n) const;

  T& operator[ ](int n);
  const T& operator[ ](int n) const;
  // …
};

void f(const vector<double> cvd, vector<double> vd)
{
  // …
  double d1 = cvd[7]; // позови const верзију [ ]
  double d2 = vd[7];  // позови не-const верзију [ ]
  cvd[7] = 9;         // грешка: позови const верзију [ ]
  vd[7] = 9;          // позови не-const верзију [ ]: ОК
}
```

Стринг (string)

- **string** је врло сличан класи vector<char>
  - Нпр. **`size(), [ ], push_back()`**
  - На исти начин је изграђен као и вектор
- **string** је оптимизован за манипулисање знаковним низовима
  - Спајање низова (**+**)
  - Нуди коришћење Цеовског низа (**c_str()**)
  - Оператор уноса (>>)



Од конкретног ка апстрактном

```
int r1 = 3 * 3 + 4 * 4 * 4;

int a, b;
int r2 = a * a + b * b * b;
```

Од конкретног ка апстрактном

```
int r1 = 3 * 3 + 4 * 4 * 4;

int a, b;
int r2 = a * a + b * b * b;

int c, d;
int r3 = c * c + d * d * d;
```

Од конкретног ка апстрактном

```
int sumPow23(int x, int y) {
    return x * x + y * y * y;
}

int r1 = sumPow23(3, 4);

int a, b;
int r2 = sumPow23(a, b);

int c, d;
int r3 = sumPow23(c, d);
```

Од конкретног ка апстрактном

```
int sumPow23(int x, int y) {
    return x * x + y * y * y;
}

int r1 = sumPow23(3, 4);

int a, b;
int r2 = sumPow23(a, b);

double c, d;
double r3 = ?
```

Од конкретног ка апстрактном

```
int sumPow23(int x, int y) {
    return x * x + y * y * y;
}

double sumPow23(double x, double y) {
    return x * x + y * y * y;
}

int r1 = sumPow23(3, 4);

int a, b;
int r2 = sumPow23(a, b);

double c, d;
double r3 = sumPow23(c, d);
```

Од конкретног ка апстрактном

```
int sumPow23(int x, int y) {
    return x * x + y * y * y;
}
```

```
mov eax, DWORD PTR [rbp-4]
imul eax, DWORD PTR [rbp-4]
mov edx, eax
mov eax, DWORD PTR [rbp-8]
imul eax, DWORD PTR [rbp-8]
imul eax, DWORD PTR [rbp-8]
add eax, edx
```

```
double sumPow23(double x, double y) {
    return x * x + y * y * y;
}
```

```
movsd xmm0, QWORD PTR [rbp-8]
movapd xmm1, xmm0
mulsd xmm1, QWORD PTR [rbp-8]
movsd xmm0, QWORD PTR [rbp-16]
mulsd xmm0, QWORD PTR [rbp-16]
mulsd xmm0, QWORD PTR [rbp-16]
addsd xmm0, xmm1
```

Тип је важан, јер смисао и извршавање зависе од тога.
Машински код је врло различит за различите типове!

(Корисна страница на интернету: Compiler Explorer godbolt.org

Од конкретног ка апстрактном

```
template<typename T> T sumPow23(T x, T y) {
    return x * x + y * y * y;
}

int r1 = sumPow23<int>(3, 4);

int a, b;
int r2 = sumPow23<int>(a, b);

double c, d;
double r3 = sumPow23<double>(c, d);
```

Али...

Од конкретног ка апстрактном

```
template<typename T> T sumPow23(T x, T y) {
    return x * x + y * y * y;
}

int r1 = sumPow23(3, 4);

int a, b;
int r2 = sumPow23(a, b);

double c, d;
double r3 = sumPow23(c, d);
```

...шаблони функције се могу инстанцирати и без експлицитног навођења стварних параметара шаблона, јер се шаблонски параметри могу закључити на основу функцијских стварних параметара.

(Могу се на сличан начин инстанцирати и шаблони класе, али то је маааало сложенија тема, па ћемо оставити за касније)

Од конкретног ка апстрактном

```
template<typename T> T sumPow23(T x, T y) {
    return x * x + y * y * y;
}

// али:

int x;
float y;
int z = sumPow23(x, y);


string e, f;
string r4 = sumPow23(e, f);
```

Од конкретног ка апстрактном

```
template<typename T> T sumPow23(T x, T y) {
    return x * x + y * y * y;
}

// али:

int x;
float y;
int z = sumPow23(x, y); // грешка!!! x и y нису истог типа
                        // проблем код закључивања параметара

string e, f;
string r4 = sumPow23(e, f); // грешка!!! * недефинисано
                            // проблем код инстанцирања шаблона
                            // (тј. код превођења функције
                            // настале инстанцирањем шаблона)
```

Од конкретног ка апстрактном

```
template<typename T> T sumPow23(T x, T y) {
    return x * x + y * y * y;
}
```

Овај шаблон се може инстанцирати за било који тип T над којим је:
- дефинисана операција множења
- дефинисана операција сабирања

(Такође, претпоставка је да је резултат израза у return наредби таквог типа који се може имплицитно конвертовати у T, и сл.)

Ови услови нису експлицитно наведени и то је главни разлог зашто када погрешимо при раду са шаблонима врло често добијемо тешко разумљиве поруке о грешкама.

Али, сада постоји механизам који омогућава експлицитно исказивање оваквих услова код шаблона. То би требало да побољша пријаву грешака и учини шаблоне разумљивијим. Тај механизам се назива „концепти“. (Ово је само информативно, за оне који гледају иза хоризонта, али неће бити на испиту :))

cppinsights.io

Source:

```
1  #include <cstdio>
2  #include <vector>
3
4  template<typename T> T sumPow23(T x, T y) {
5      return x * x + y * y * y;
6  }
7
8  int main()
9  {
10     int r1 = sumPow23<int>(3, 4);
11
12     int a, b;
13     int r2 = sumPow23<int>(a, b);
14
15     double c, d;
16     double r3 = sumPow23<double>(c, d);
17 }
```

Insight:

```
1  #include <cstdio>
2  #include <vector>
3
4  template<typename T> T sumPow23(T x, T y) {
5      return x * x + y * y * y;
6  }
7
8  /* First instantiated from: insights.cpp:14 */
9  #ifdef INSIGHTS_USE_TEMPLATE
10 template<>
11 int sumPow23<int>(int x, int y)
12 {
13   return (x * x) + ((y * y) * y);
14 }
15 #endif
16
17
18 /* First instantiated from: insights.cpp:20 */
19 #ifdef INSIGHTS_USE_TEMPLATE
20 template<>
21 double sumPow23<double>(double x, double y)
22 {
23   return (x * x) + ((y * y) * y);
24 }
25 #endif
26
27
```