Konstruktory a destruktory

Nevýhoda

- atributy po vytvoření objektu nejsou automaticky inicializovány
  - hodnota atributů je náhodná
- vytvoření metody pro inicializaci, kterou musí programátor explicitně zavolat, není spolehlivé
  - *není zaručeno, že ji programátor, či spíše další uživatel knihovny, nezapomene zavolat*

Bezpečné pole z minulé hodiny

```
class BezpPole
{
 private:
  int n,akt,*pole;
 public:
  void vloz(int prvek);
  int vrat_prvek(int index);
  int vrat_akt_pocet();
  void init();
  void deinit();
};
```

Bezpečné pole z minulé hodiny

```
void main()
{
  int i;
  BezpPole a;
  a.init(); // na init nesmíme zapomenout
  a.vloz(3); a.vloz(4); a.vloz(2);
  for(i=0;i<a.vrat_akt_pocet();i++)
    cout << a.vrat_prvek(i) << endl;
  a.deinit(); // na deinit nesmíme zapomenout
}
```

Konstruktor

- OOP poskytuje prostředek pro počáteční inicializaci (nejen) atributů tzv. *konstruktor*

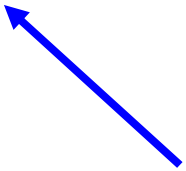
Konstruktor

- jde o speciální metodu, která je automaticky vyvolána při vzniku objektu
  - konstruktor musí přirozeně programátor naprogramovat
- slouží pro počáteční inicializaci objektu
- pokud není konstruktor definován, překladač jej vytvoří automaticky (implicitní), který ale hodnoty atributů neinicializuje (je prázdný, nedělá nic)

- konstruktor má stejný identifikátor jako je jméno (identifikátor) třídy
  - t.j. musíme deklarovat a vytvořit metodu se stejným názvem jako jméno třídy a ta je automaticky konstruktorem
- konstruktor nevrací žádná data, není deklarován ani jako **`void`**!
- parametry může mít libovolné, lze jej přetížit
  - je to dokonce i vhodné, abychom mohli inicializovat objekt různými způsoby
  - je vhodné mít vždy implicitní konstruktor (bez parametrů)

```
class Komplex
{
 public:
    float re,im; //reálná a imag. část
    float velikost();
    Komplex();
    Komplex(float real, float imag);
};
```

implicitní konstruktor



přetížený konstruktor

- **implementace**

```
Komplex::Komplex()
{
  re = im = 0;
}

Komplex::Komplex(float real, float imag)
{
  re = real; im = imag;
}
```

- použití

```
void main()
{
  Komplex k_cislo;
  Komplex c1(1,3);
  Komplex *pc1,*pc2;

  pc1 = new Komplex(5,10);
  pc2 = new Komplex();
  float vel = k_cislo.velikost();
  vel = pc1 -> velikost();
  delete pc1; delete pc2;
}
```

zde se volá implicitní konstruktor

zde se volá přetížený konstruktor

zde se volá implicitní konstruktor

- pokud deklarujeme konstruktor s parametry, musíme deklarovat a implementovat i implicitní, chceme-li jej využívat
  - pak již implicitní konstruktor není generován automaticky
- při vzniku objektu se nejprve alokuje paměť pro objekt (pro atributy) a pak se volá konstruktor
- v konstruktoru zpravidla inicializujeme atributy nebo alokujeme potřebnou dynamickou paměť, jestliže ji objekt využívá
  - ruší se pak v *destruktoru*

- konstruktor nad objektem nelze dodatečně vyvolat

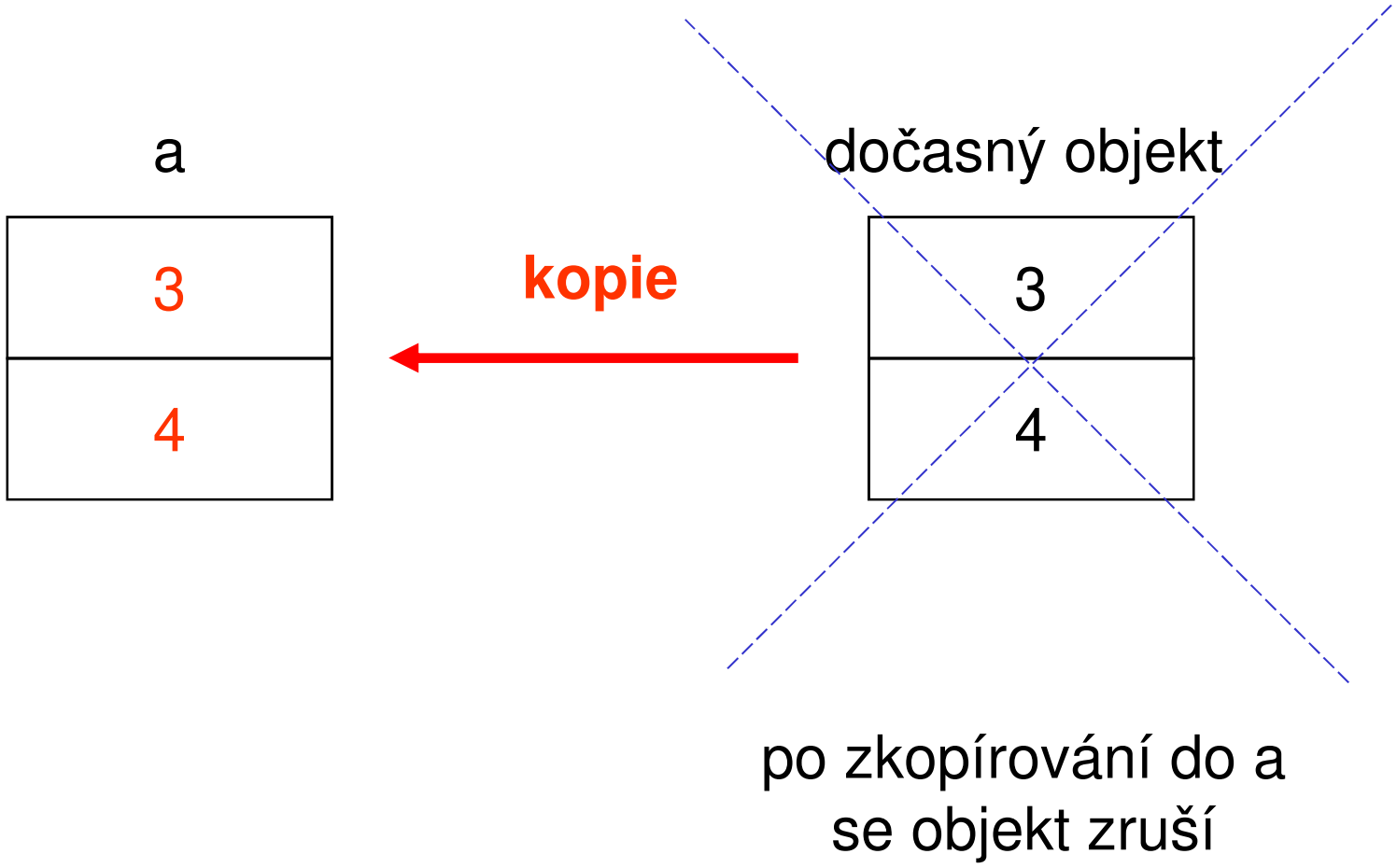
```
void main(void)
{
  Komplex c;

  c.Komplex(2,2);
}
```

chyba

- explicitní volání konstruktoru znamená vytvoření dočasného (nepojmenovaného) objektu, který se po použití automaticky zruší

```
Komplex a;
// zavolal se impl. konstruktor

a = Komplex(3,4);
```

  – do proměnné `a` se zkopíruje hodnota dočasného objektu, ve kterém je re=3 a im=4


a
3
4
kopie
dočasný objekt
3
4
po zkopírování do a
se objekt zruší

Poznámka:

- inicializaci atributů lze v konstruktoru provést pomocí *inicializátorů* (při deklaraci třídy jako in-line konstruktor nebo při implementaci v souboru .cpp)
  - u atributů typu reference, konstantních atributů a objektových atributů, kde není ve třídě definován implicitní konstruktor, je to jediná možnost

```
class Komplex {
  float re, im;
public:
  Komplex(): re(0),im(0) {};
  Komplex(float x, float y): re(x),im(y) {};
}
```

- konstruktor s inicializátory v souboru .cpp:

```
Komplex::Komplex(): re(0),im(0)
{

}
Komplex::Komplex(float x, float y):
  re(x),im(y)
{

}
```

```
class A
{
    int x;
    bool barevne;
public:
  // máme jen konstruktor s parametry
  A(bool b);
};

A::A(bool b)
{
  x = 0; barevne = b;
}
```

```
class B
{
    int z;
    A a;
public:
  B();
  B(bool b);
};

B::B(): a(true)
{
  z = 0;
}
B::B(bool b): a(b),z(0)
{

}
```

Destruktor

- speciální metoda, která je automaticky volána při rušení objektu
- nejprve je zavolán destruktor nad objektem a pak je objekt zrušen (dealokována paměť)
- typicky:
  - pokud je v konstruktoru dynamicky alokována paměť, pak je v destruktoru provedena dealokace
  - změna statických (třídních) atributů

- destruktor má stejný identifikátor jako je jméno třídy a od konstruktoru je odlišen úvodním znakem „~” (vlnovka)
  - t.j. musíme deklarovat a vytvořit metodu se stejným názvem jako jméno třídy uvozenou vlnovkou a ta je automaticky destruktorem
- destruktor nesmí mít žádné parametry a nevrací žádnou hodnotu
- destruktor lze vyvolat přímo

Bezpečné pole
s konstruktorem/destruktorem

```
class BezpPole
{
 private:
  int n,akt,*pole;
 public:
  void vloz(int prvek);
  int vrat_prvek(int index);
  int vrat_akt_pocet();
  BezpPole();
  ~BezpPole();
};
```

```
BezpPole::BezpPole()
{
 n = 0; akt = 0; pole = NULL;
}

BezpPole::~BezpPole()
{
  if (pole != NULL) delete [] pole;
}
```

Příklad - fronta

- datová struktura typu **FIFO**
  - First In - First Out
- prvky se odebírají v pořadí, jak byly vkládány

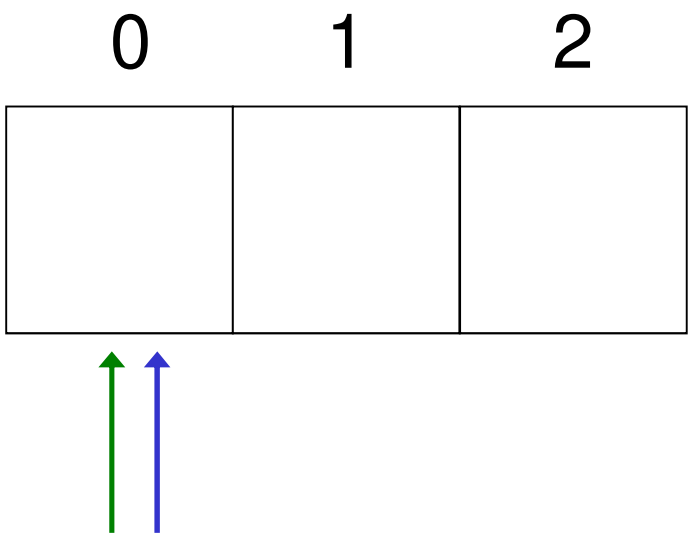
Možné implementace

- jako spojový seznam
  - „neomezená“ velikost fronty (omezená pouze velikostí dostupné paměti)
  - operace vložení prvku znamená vložení prvku na konec seznamu
  - operace výběr prvku je výběr z čela fronty
- polem pevné délky
  - fronta s omezenou velikostí
  - implementuje se jako tzv. **kruhová fronta**

Implementace pomocí pole pevné délky

- fronta je reprezentována polem a indexem čela a konce fronty
- fronta má pevnou délku a je implementována jako **kruhová**
  - indexy se zvyšují modulo délka pole
- je nutné implementovat ještě dotaz, zda je fronta plná a prázdná

Fronta délky $n = 3$



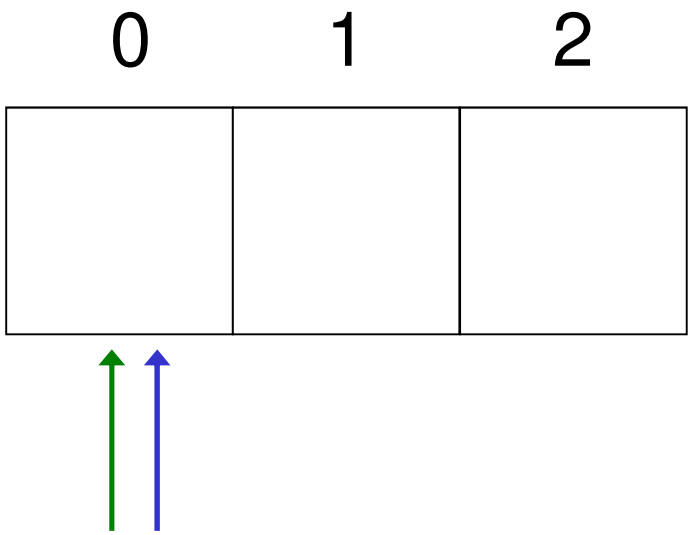
celo: 0

konec: 0

prázdná

plná

Fronta délky $n = 3$



vloz(3)

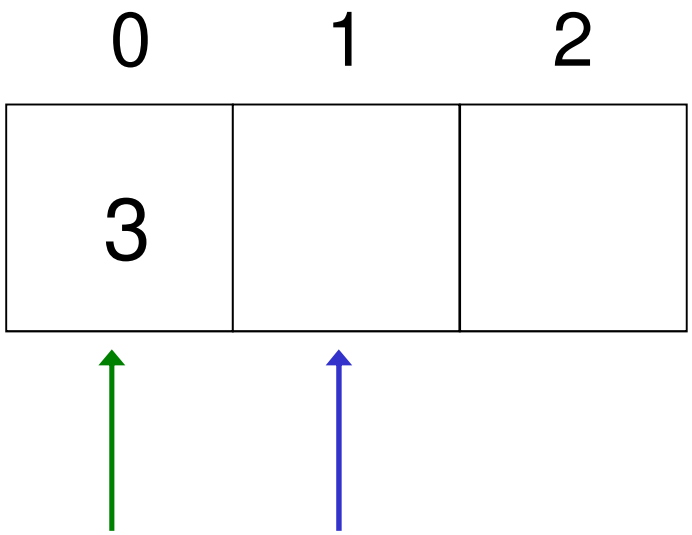
celo: 0

konec: 0

prázdná

plná

Fronta délky $n = 3$



vloz(3)

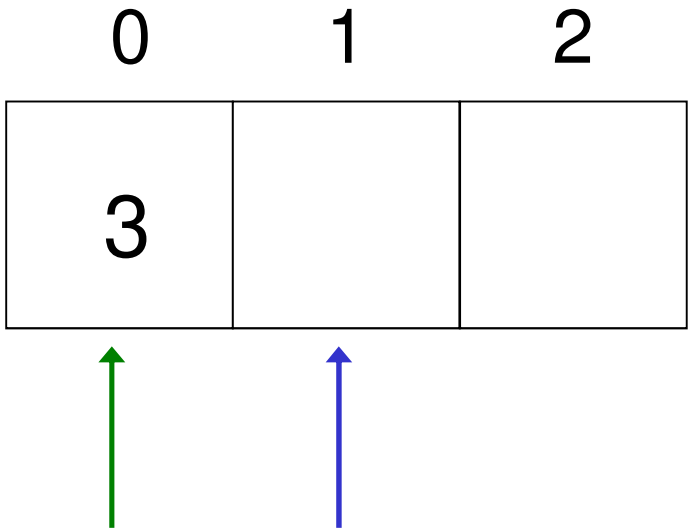
celo: 0

konec: 1

prázdná

plná

Fronta délky $n = 3$



celo: 0

konec: 1

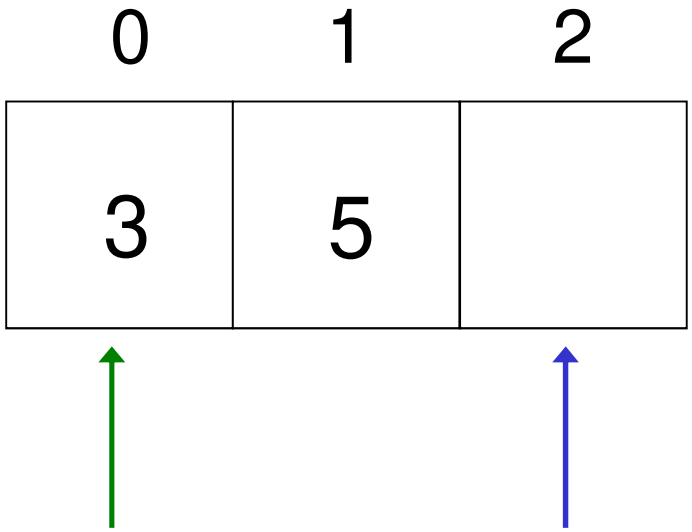
prázdná

plná

vloz(3)

vloz(5)

Fronta délky $n = 3$



celo: 0

konec: 2

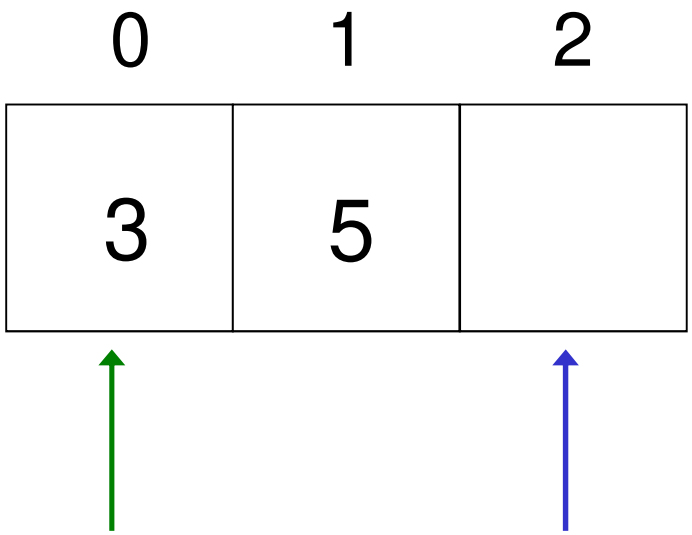
prázdná

plná

vloz(3)

vloz(5)

Fronta délky $n = 3$



celo: 0

konec: 2

prázdná

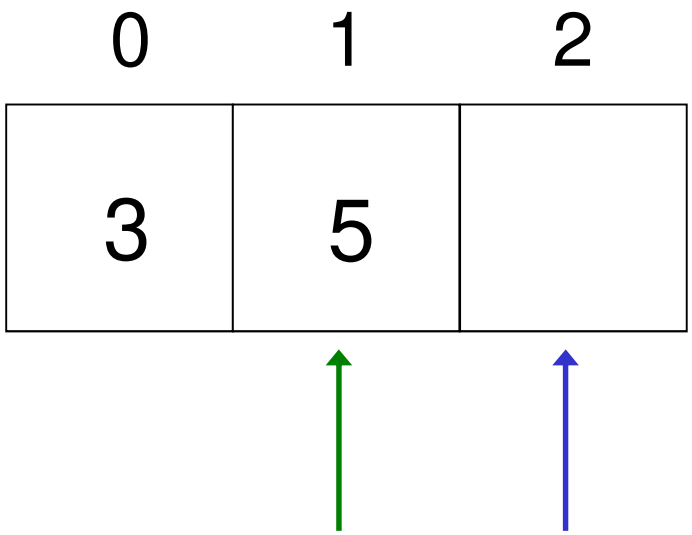
plná

vloz(3)

vloz(5)

vyjmi() – vrátí hodnotu z čela fronty - 3

Fronta délky $n = 3$



celo: 1

konec: 2

prázdná

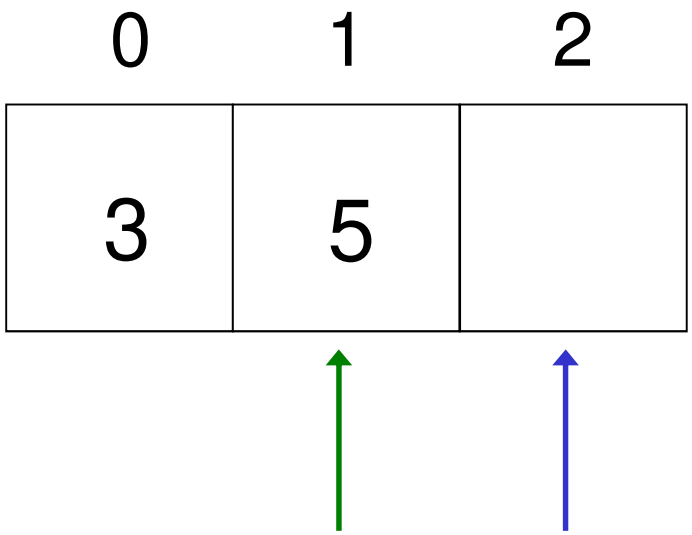
plná

vloz(3)

vloz(5)

vyjmi() – vrátí hodnotu z čela fronty - 3

Fronta délky $n = 3$



celo: 1

konec: 2

prázdná

plná

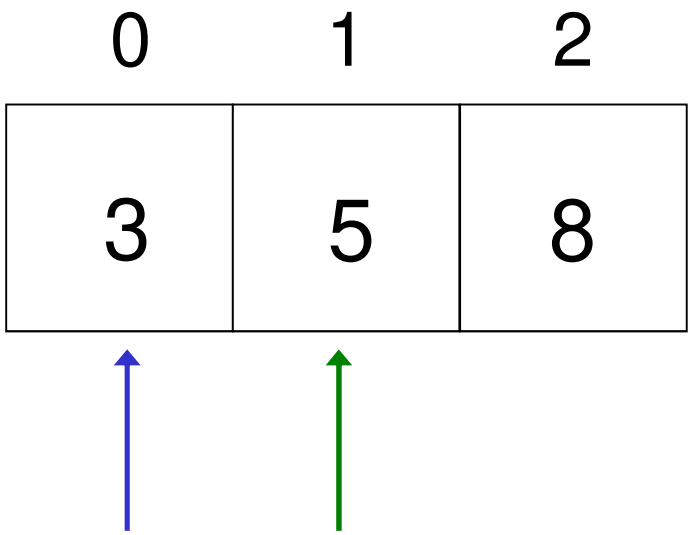
vloz(3)

vloz(5)

vyjmi() – vrátí hodnotu z čela fronty - 3

vloz(8)

Fronta délky $n = 3$



celo: 1

konec: 0

prázdná

plná

vloz(3)

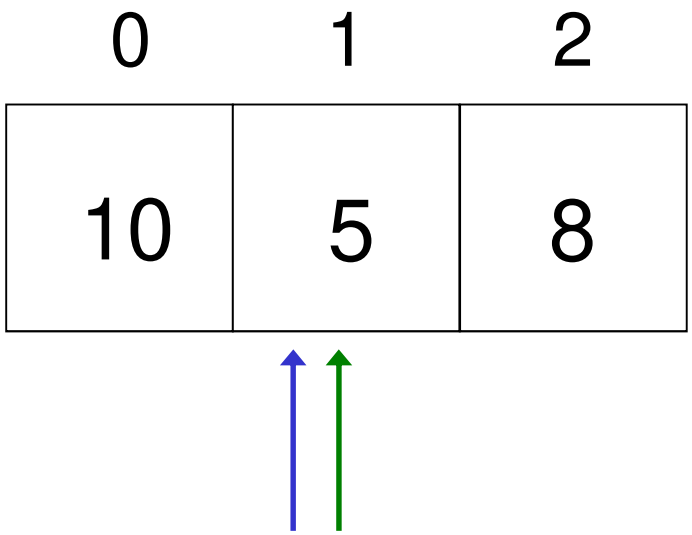
vloz(5)

vyjmi() – vrátí hodnotu z čela fronty - 3

vloz(8)

vloz(10)

Fronta délky $n = 3$



celo: 1

konec: 1

prázdná

plná

vloz(3)

vloz(5)

vyjmi() – vrátí hodnotu z čela fronty - 3

vloz(8)

vloz(10)

- index *celo* ukazuje na prvek pole na čele fronty, který bude odebrán
- index *konec* ukazuje na prvek v poli, kam se zapíše nový prvek
- pokud se indexy *celo* a *konec* rovnají, je fronta buď plná nebo prázdná
  - podle rovnosti indexů není možné rozlišit stav fronty
  - musím tedy tyto stavy uchovávat zvlášť
  - *pravidlo*: pokud se po přidání prvku indexy rovnají, je fronta plná (obdobně: prázdná)

Příklad - kruhová fronta pevné délky

```
class TFronta {
  int delka,celo,konec;
  bool plna,prazdna;
  int *fronta;
 public:
  bool je_prazdna();
  bool je_plna();
  bool vloz(int prvek);
  int vyber();
  TFronta();
  TFronta(int velikost);
  ~TFronta();
};
```

destruktor

```
TFronta::TFronta()
{
  delka = 50;
  celo = konec = 0;
  plna = false; prazdna = true;
  fronta = new int[50];
}
TFronta::TFronta(int velikost)
{
  delka = velikost;
  celo = konec = 0;
  plna = false; prazdna = true;
  fronta = new int[velikost];
}
```

```
bool TFronta::je_prazdna()
{
  return prazdna;
}

bool TFronta::je_plna()
{
  return plna;
}
```

```
bool TFronta::vloz(int prvek)
{
  prazdna = false;
  if (plna) return false;
  fronta[konec] = prvek;
  konec = (konec+1)%delka;
  if (konec == celo) plna = true;
  return true;
}
```

```
int TFronta::vyber()
{
  plna = false;
  if (prazdna) return -1;
  int prvek = fronta[celo];
  celo=(celo+1)%delka;
  if (celo==konec) prazdna = true;
  return prvek;
}

TFronta::~TFronta()
{
  delete [] fronta;
}
```

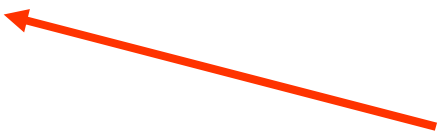
Poznámky:

- spoléhat se na návratovou hodnotu -1 metody vyber(), která signalizuje pokus o výběr z prázdné fronty, není vhodné
  - nepoznám, zda byla vyjmuta -1 nebo bylo vybíráno z prázdné fronty
    - možné řešení: např. nastavovat chybovou proměnnou
    - poznáme i lepší možnost než chybová proměnná

Odstranění problému spočívá:

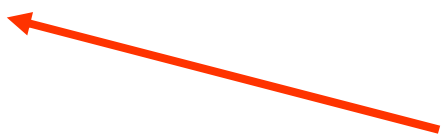
- ve správném používání knihovních funkcí
  - před každým voláním funkce vyjmi() aplikace otestuje, zda není fronta prázdná
  - např. `while (!f.je_prazdna())`
- v používání **výjimek**
  - rys objektových jazyků, který poznáme později

```
void main()
{
  int x;
  TFronta f1;

  f1.vloz(10);
  f1.vloz(-3);
  x = f1.vyber();
  if (f1.je_prazdna()) cout << "Prazdna";
  else cout << "Jeste tam neco je!";
};
```



zde je vyvolán destruktor

```
void main()
{
  int x;
  TFronta *f1;
  f1 = new TFronta(100);
  f1 -> vloz(10);
  f1 -> vloz(-3);
  x = f1 -> vyber();
  if (f1->je_prazdna()) cout << "Prazdna";
  else cout << "Jeste tam neco je!";
  delete f1;
};
```

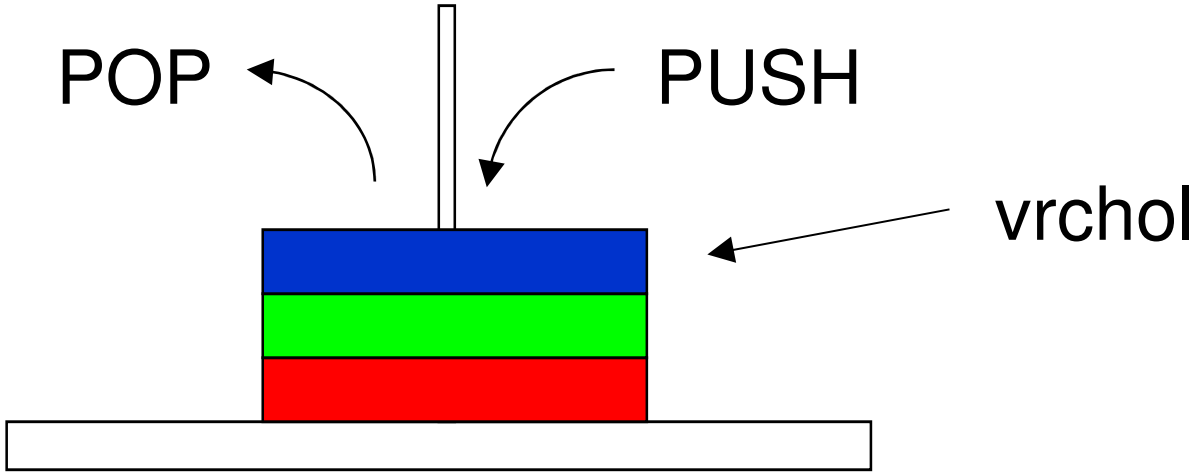


zde je vyvolán destruktor

Zásobník (Stack)

- datová struktura typu **LIFO**
  - Last In - First Out
- nejprve se vybírá prvek, který byl vložen na *vrchol (top)* zásobníku jako poslední
- operace:
  - **PUSH** (uložení hodnoty na vrchol zásobníku)
  - **POP** (odebrání hodnoty z vrcholu zásobníku)

- někdy bývá implementována také operace **TOP**
  - zjištění hodnoty na vrcholu zásobníku bez odebrání prvku



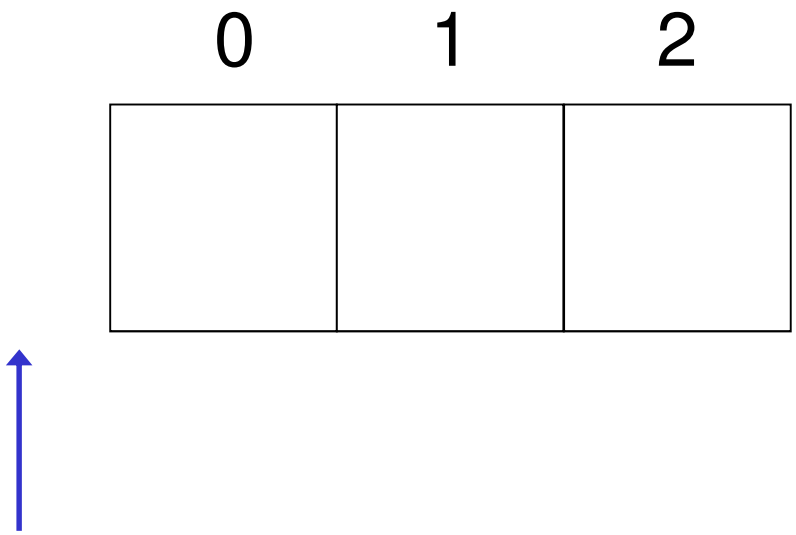
Možné implementace

- na principu spojového seznamu
  - „neomezená“ velikost zásobníku (omezená pouze velikostí dostupné paměti)
  - operace vložení prvku znamená vložení prvku na konec seznamu
  - operace výběr prvku je výběr z konce seznamu
  - zásobník je reprezentován ukazatelem na vrchol
  - prvek seznamu nese hodnotu a ukazatel na předchozí prvek v zásobníku
- polem (pevné nebo proměnné délky)
  - kruhová implementace není potřebná
  - stačí index na vrchol

Zásobník pomocí pole

- dvě varianty implementace
  - vrchol obsahuje index uloženého posledního prvku
    - při vložení nového prvku nejprve zvednu index a zapíši nový, při výběru vezmu prvek na tomto indexu a pak snížím index vrcholu; počáteční hodnota indexu vrcholu je -1
  - vrchol obsahuje index, kam mám vložit nový prvek
    - při vložení nového prvku zapíši prvek do pole na vrchol a následně jej zvednu o 1; při výběru nejprve snížím index o 1 a vrátím prvek; počáteční hodnota vrcholu je 0

Zásobník délky $n = 3$ (varianta 1)

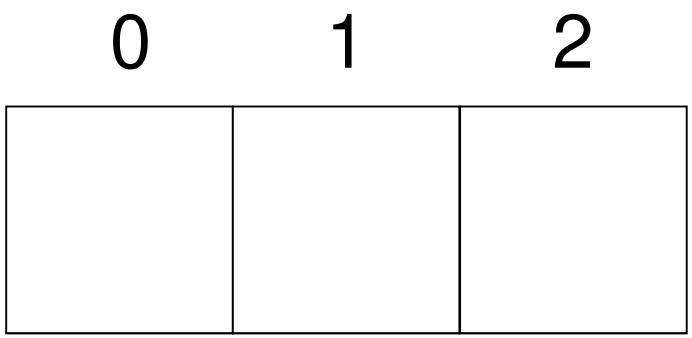


vrchol: -1

prázdný

plný

Zásobník délky $n = 3$ (varianta 1)



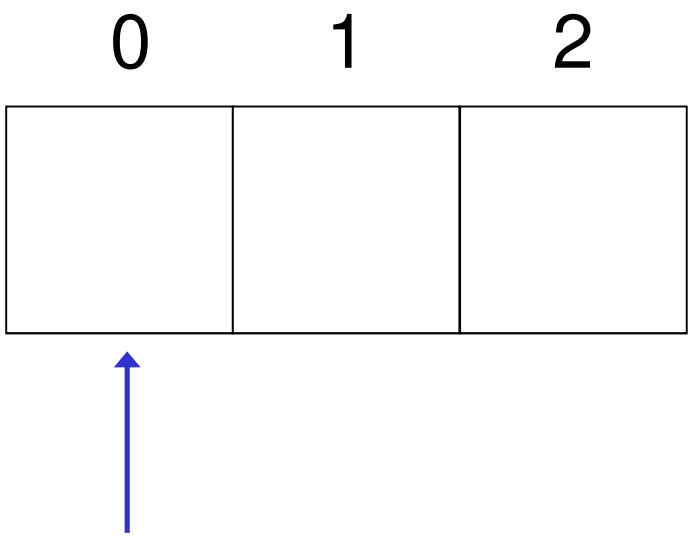
push(3)

vrchol: -1

prázdný

plný

Zásobník délky $n = 3$ (varianta 1)



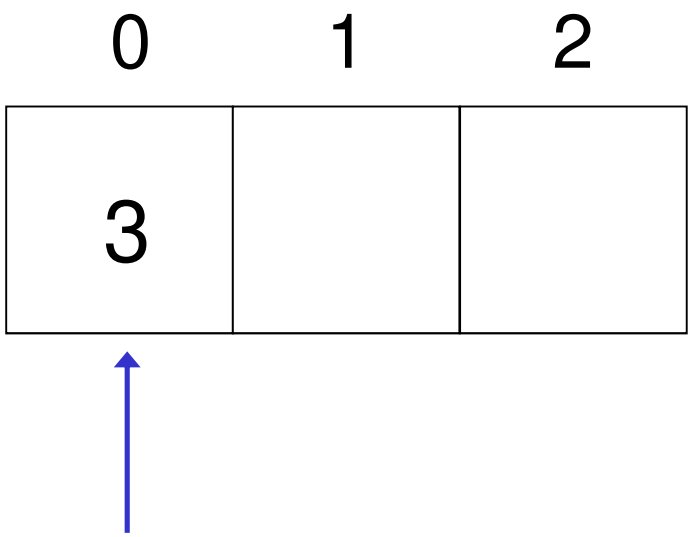
push(3)

vrchol: 0

prázdný

plný

Zásobník délky $n = 3$ (varianta 1)



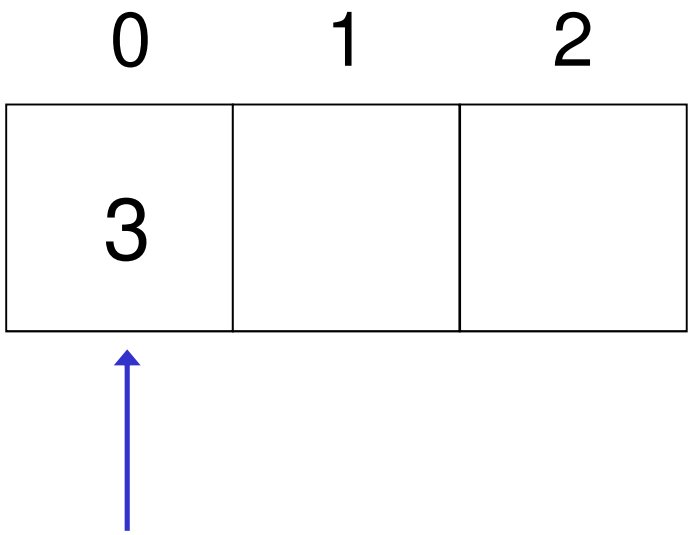
push(3)

vrchol: 0

prázdný

plný

Zásobník délky $n = 3$ (varianta 1)



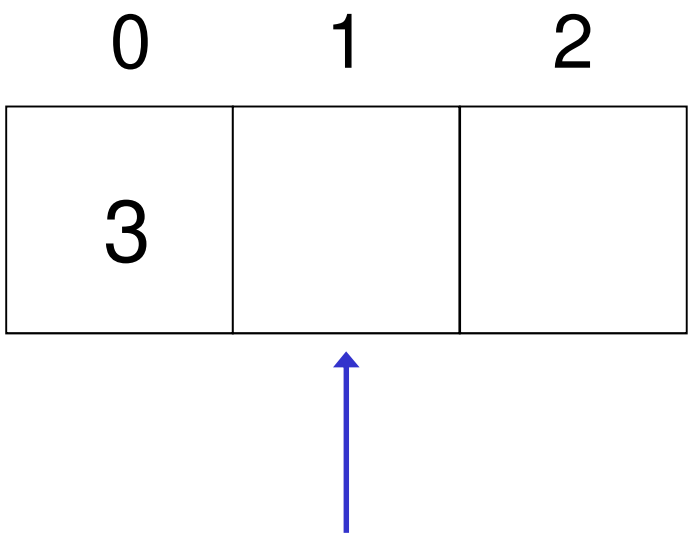
vrchol: 0

prázdný

plný

push(3)
push(8)

Zásobník délky $n = 3$ (varianta 1)



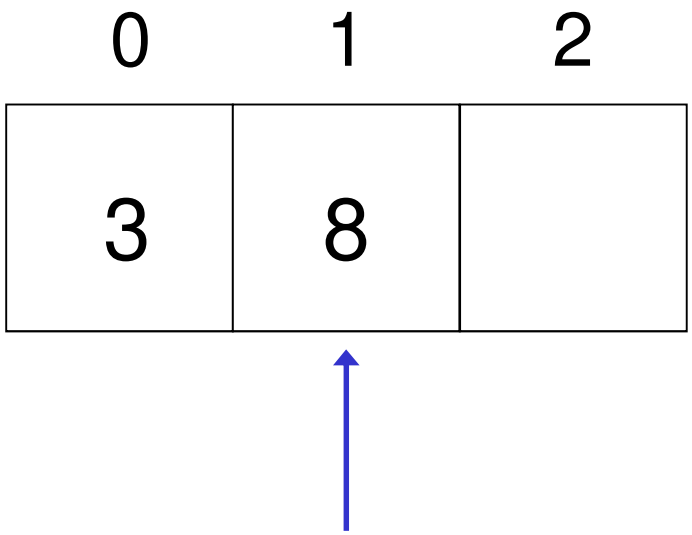
vrchol: 1

prázdný

plný

push(3)
push(8)

Zásobník délky $n = 3$ (varianta 1)



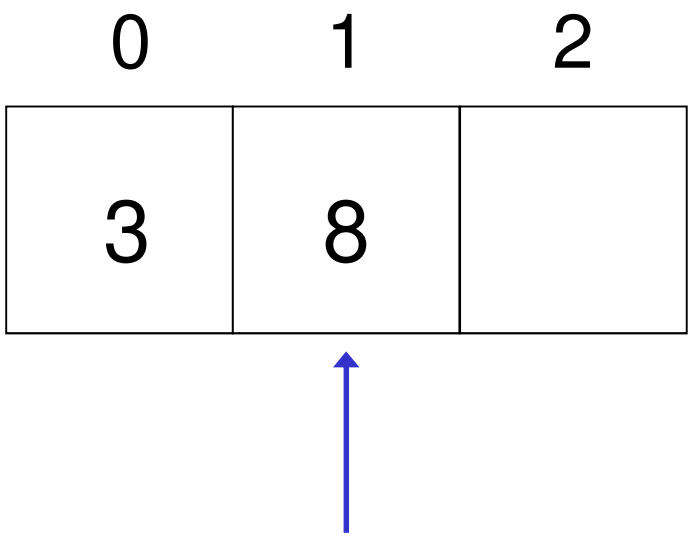
vrchol: 1

prázdný

plný

push(3)
push(8)

Zásobník délky $n = 3$ (varianta 1)



vrchol: 1

prázdný

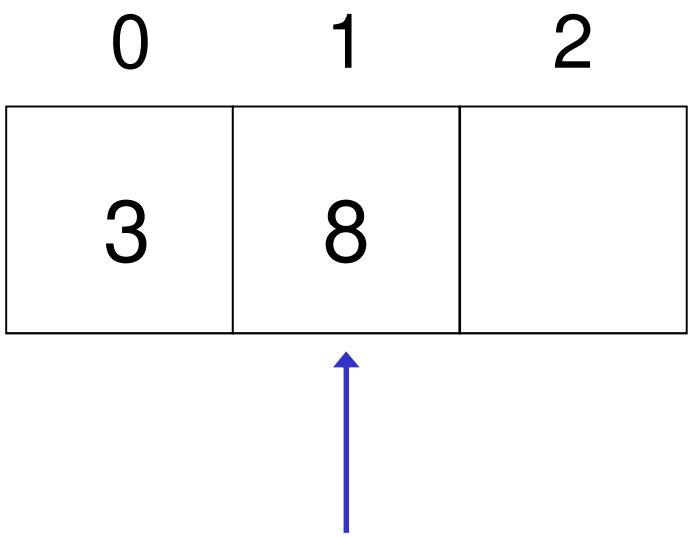
plný

push(3)

push(8)

top() vrátí hodnotu z vrcholu, tj. 8, ale neodebere ji

Zásobník délky $n = 3$ (varianta 1)



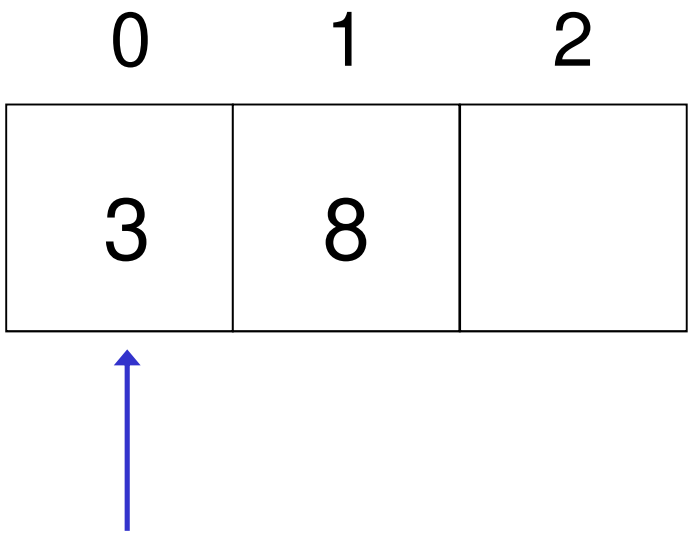
vrchol: 1

prázdný

plný

push(3)
push(8)
top()
pop() vrátí hodnotu z vrcholu, tj. 8, a odebere ji

Zásobník délky $n = 3$ (varianta 1)



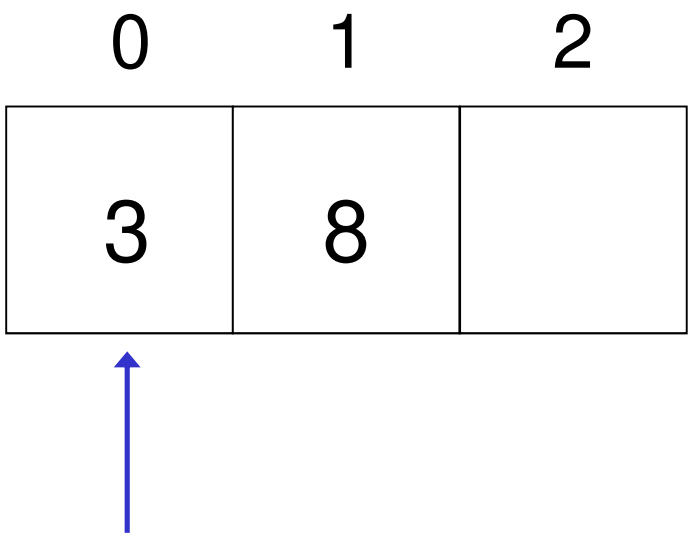
vrchol: 0

prázdný

plný

push(3)
push(8)
top()
pop()

Zásobník délky $n = 3$ (varianta 1)



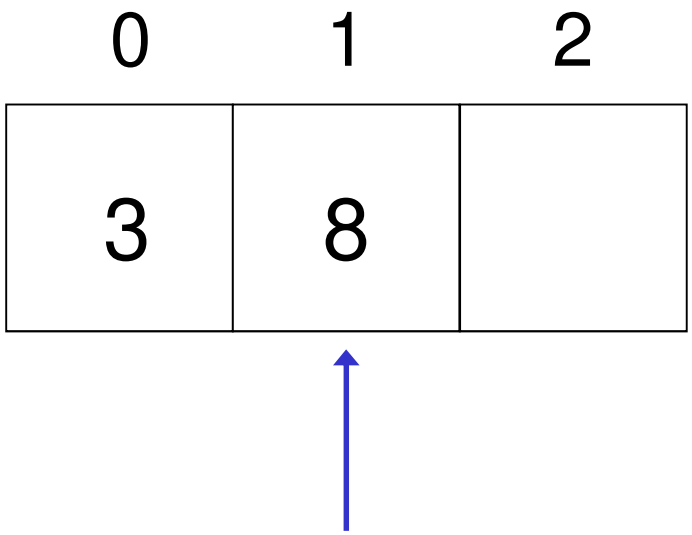
vrchol: 0

prázdný

plný

push(3)
push(8)
top()
pop()
push(10)

Zásobník délky $n = 3$ (varianta 1)



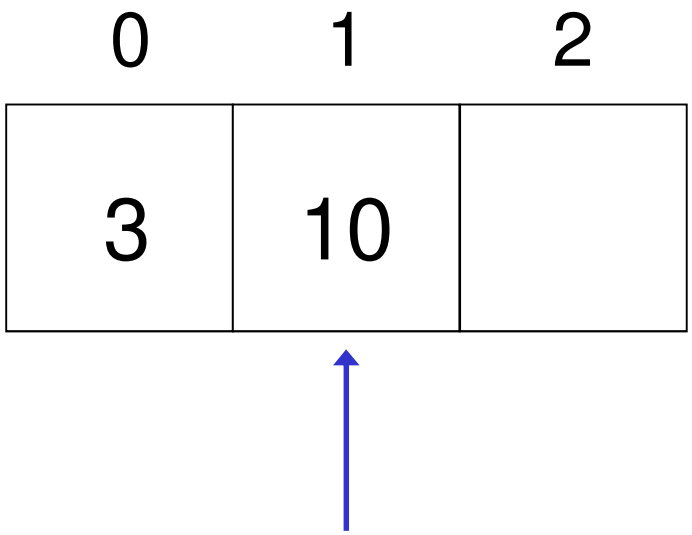
vrchol: 1

prázdný

plný

push(3)
push(8)
top()
pop()
push(10)

Zásobník délky $n = 3$ (varianta 1)



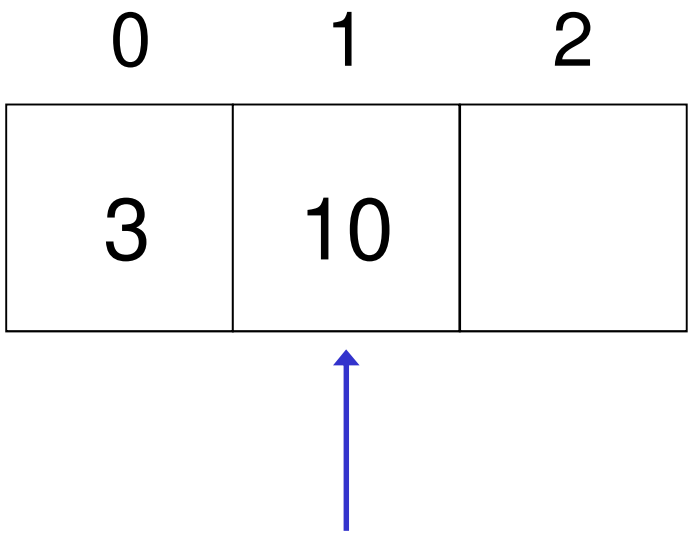
vrchol: 1

prázdný

plný

push(3)
push(8)
top()
pop()
push(10)

Zásobník délky $n = 3$ (varianta 1)



vrchol: 1

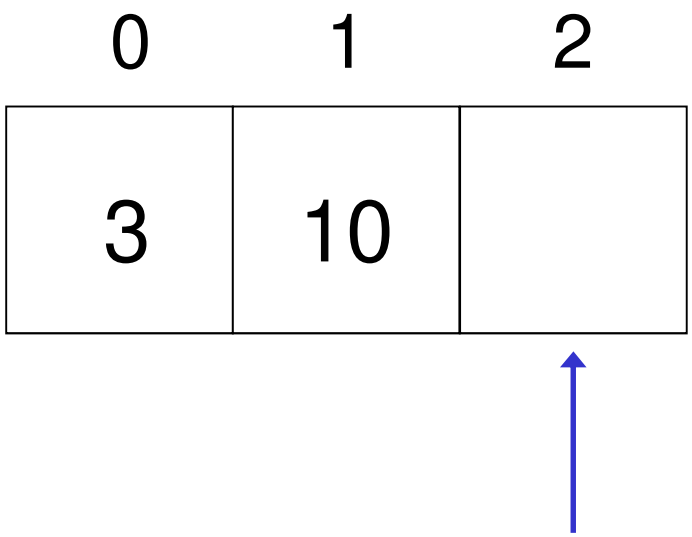
prázdný

plný

push(3)
push(8)
top()
pop()
push(10)

push(13)

Zásobník délky $n = 3$ (varianta 1)



vrchol: 2

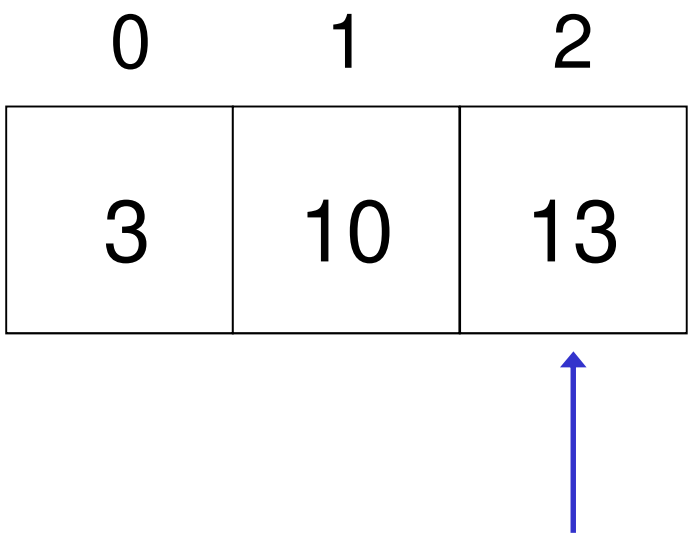
prázdný

plný

push(3)
push(8)
top()
pop()
push(10)

push(13)

Zásobník délky $n = 3$ (varianta 1)



vrchol: 2

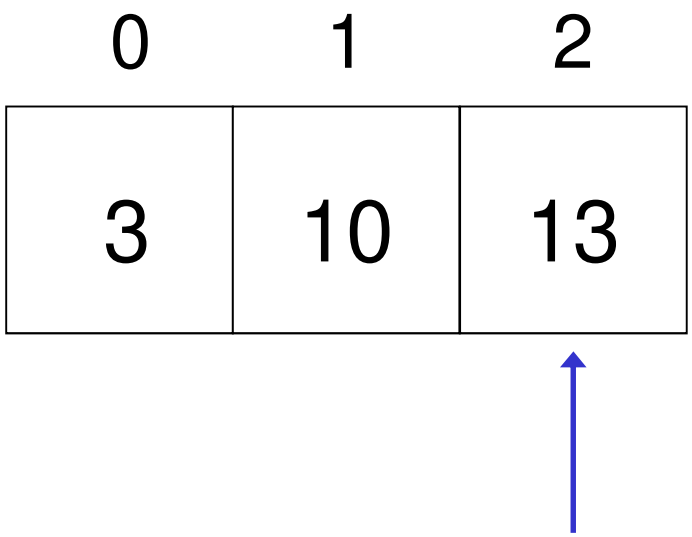
prázdný

plný

push(3)
push(8)
top()
pop()
push(10)

push(13)

Zásobník délky $n = 3$ (varianta 1)



vrchol: 2

prázdný

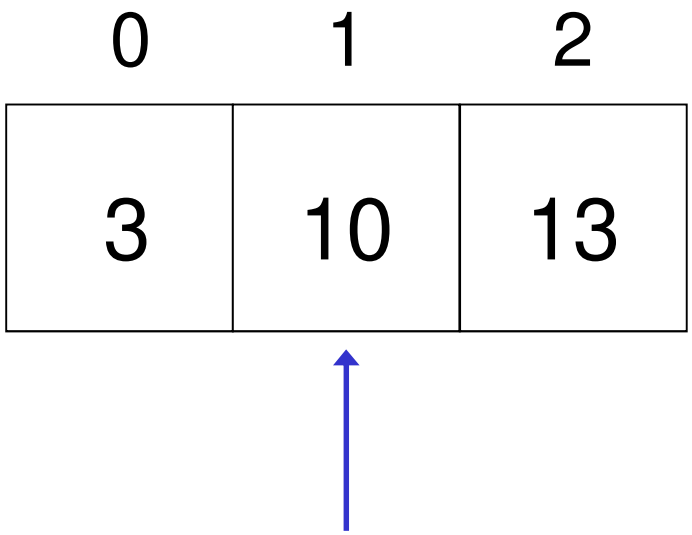
plný

push(3)
push(8)
top()
pop()
push(10)

push(13)
pop()

vrátí hodnotu 13 a odebere ji

Zásobník délky $n = 3$ (varianta 1)



vrchol: 1

prázdný

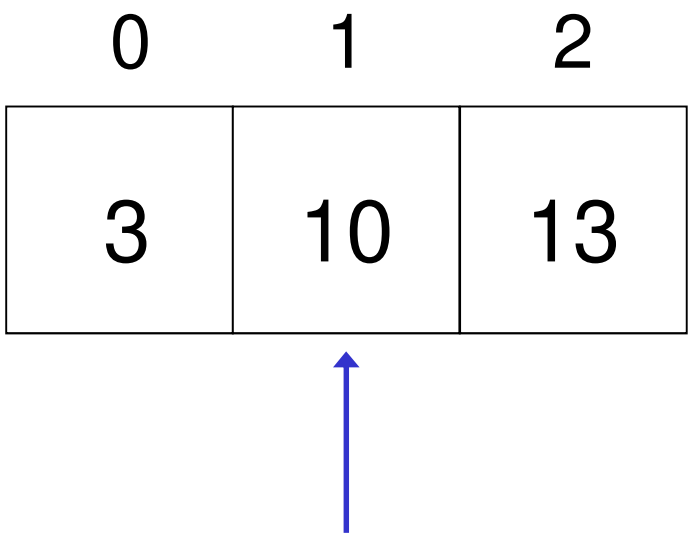
plný

push(3)
push(8)
top()
pop()
push(10)

push(13)
pop()

vrátí hodnotu 13 a odebere ji

Zásobník délky $n = 3$ (varianta 1)



vrchol: 1

prázdný

plný

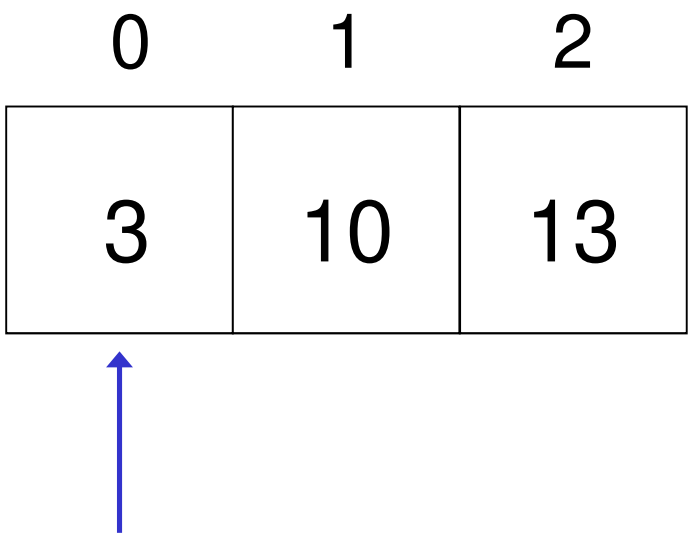
push(3)
push(8)
top()
pop()
push(10)

push(13)
pop()
pop()

vrátí hodnotu 13 a odebere ji

vrátí hodnotu 10 a odebere ji

Zásobník délky $n = 3$ (varianta 1)



vrchol: 0

prázdný

plný

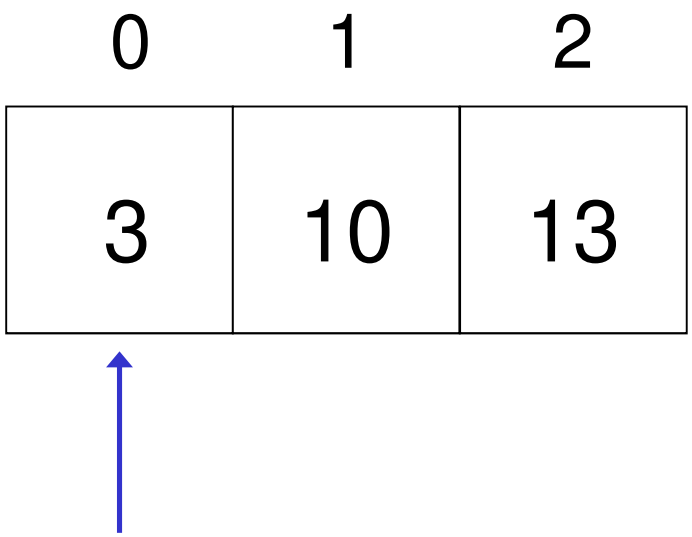
push(3)
push(8)
top()
pop()
push(10)

push(13)
pop()
pop()

vrátí hodnotu 13 a odebere ji

vrátí hodnotu 10 a odebere ji

Zásobník délky $n = 3$ (varianta 1)



vrchol: 0

prázdný

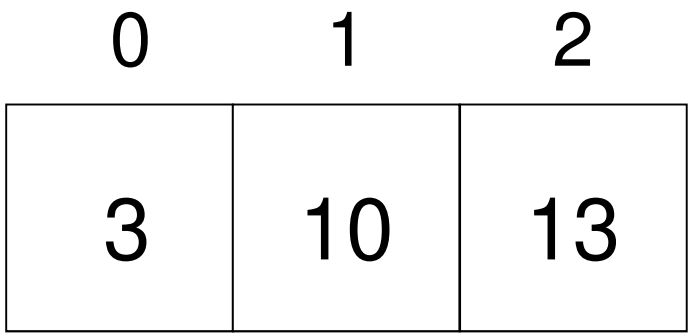
plný

push(3)
push(8)
top()
pop()
push(10)

push(13)
pop()
pop()
pop()

vrátí hodnotu 13 a odebere ji
vrátí hodnotu 10 a odebere ji
vrátí hodnotu 3 a odebere ji

Zásobník délky $n = 3$ (varianta 1)



vrchol: -1

prázdný

plný

push(3)
push(8)
top()
pop()
push(10)

push(13)
pop()
pop()
pop()

vrátí hodnotu 13 a odebere ji
vrátí hodnotu 10 a odebere ji
vrátí hodnotu 3 a odebere ji

Fronta na principu spojového seznamu

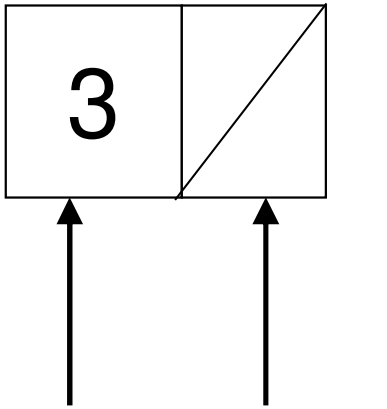
celo konec

```
NULL
```

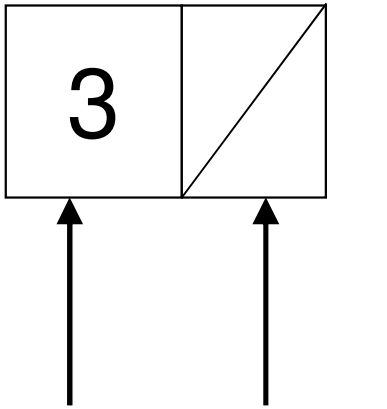
celo konec

```
NULL
```

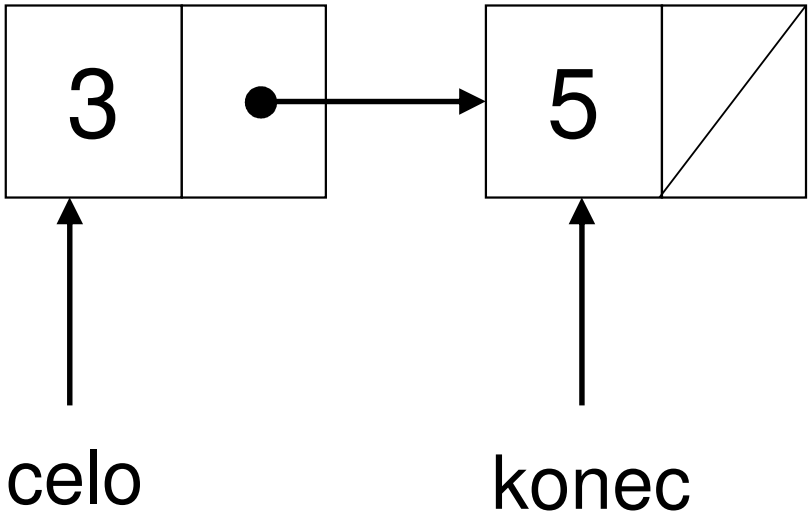
vloz(3)



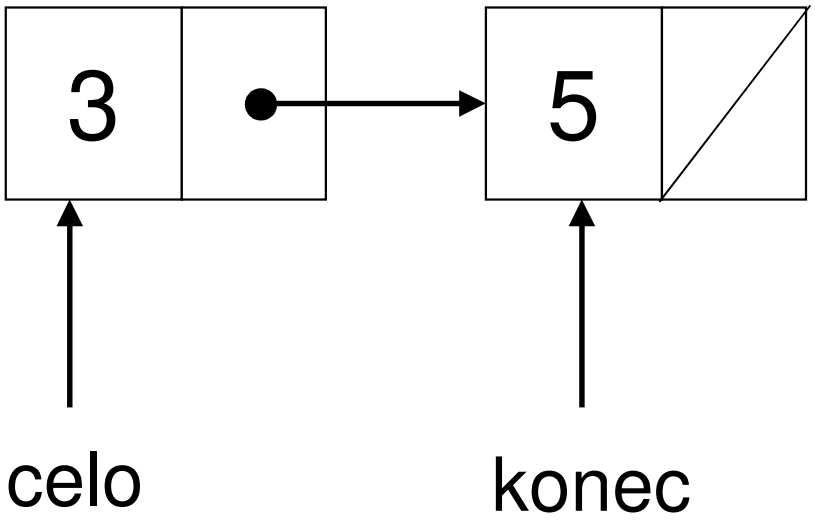
vloz(3)



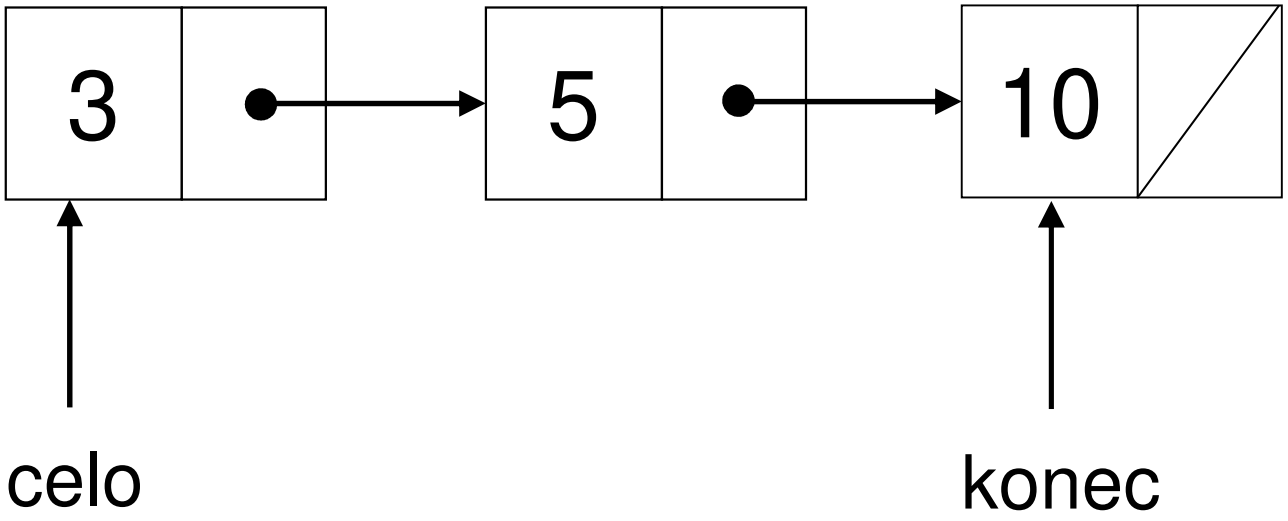
vloz(3)
vloz(5)



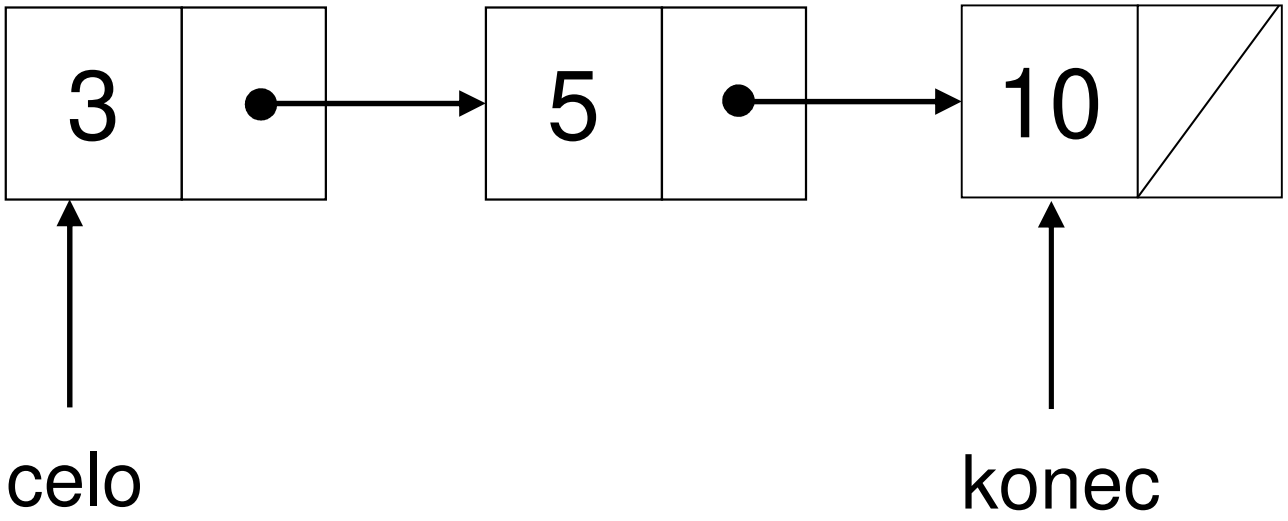
vloz(3)
vloz(5)



vloz(3)
vloz(5)
vloz(10)



vloz(3)
vloz(5)
vloz(10)

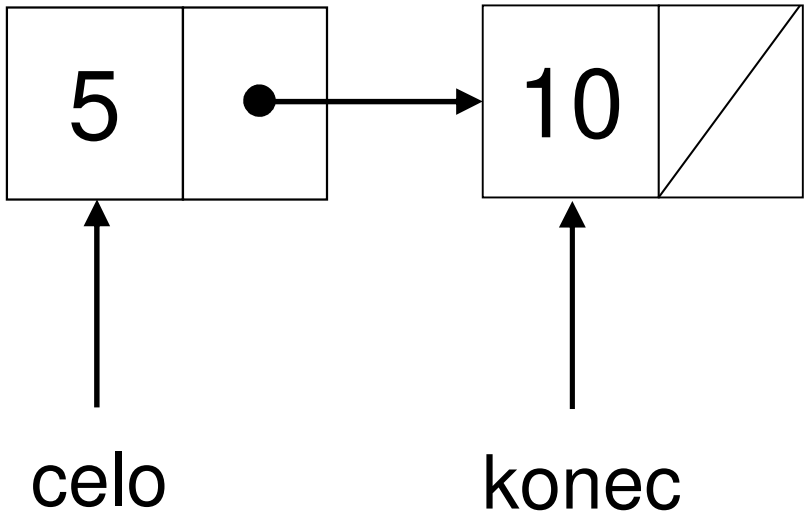


vloz(3)

vloz(5)

vloz(10)

vyjmi() – vrátí hodnotu z čela fronty - 3

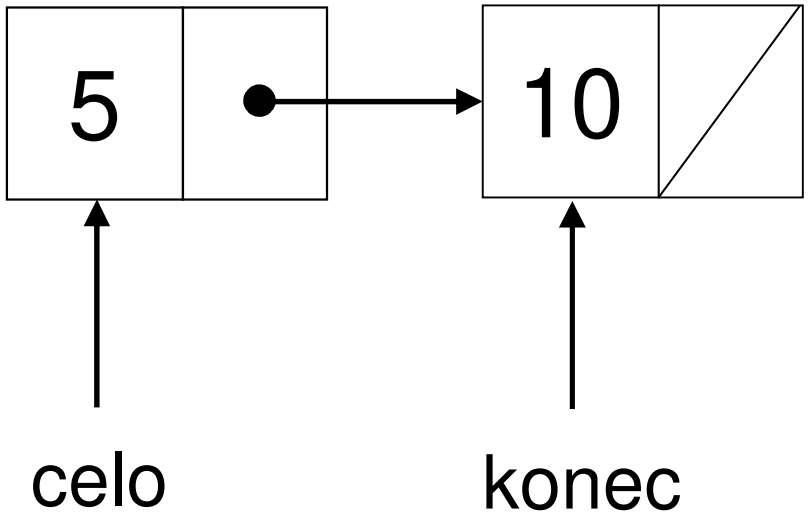


vloz(3)

vloz(5)

vloz(10)

vyjmi() – vrátí hodnotu z čela fronty - 3



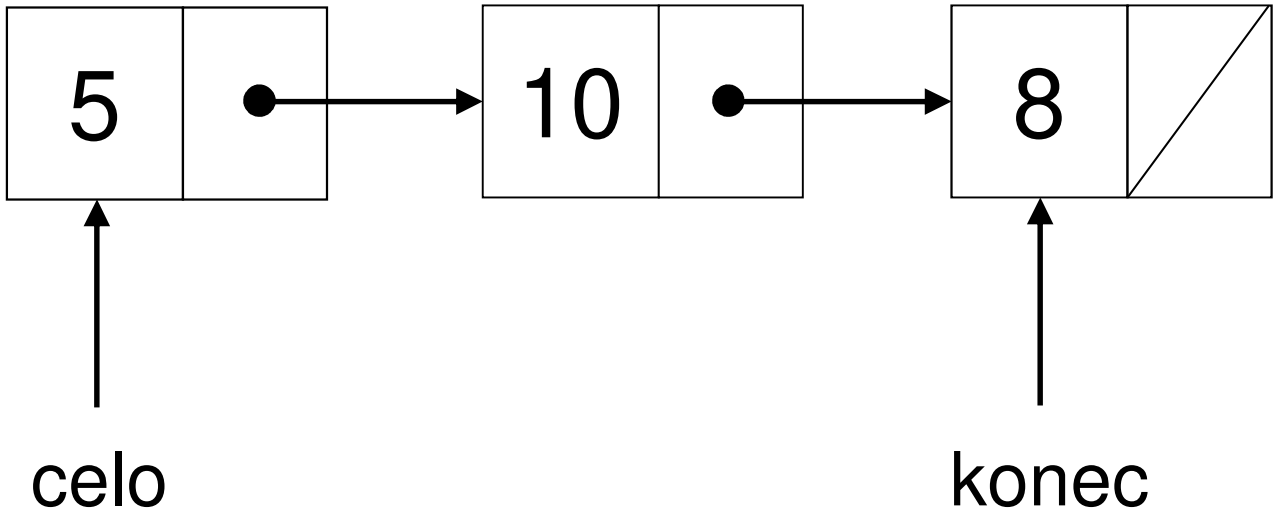
vloz(3)

vloz(5)

vloz(10)

vyjmi() – vrátí hodnotu z čela fronty - 3

vloz(8)



vloz(3)

vloz(5)

vloz(10)

vyjmi() – vrátí hodnotu z čela fronty - 3

vloz(8)

Deklarace třídy:

```
class TFronta
{
  class TPolozka {
   int prvek; TPolozka *dalsi;
   TPolozka(int novy_prvek);
  }
  TPolozka *celo, *konec;
 public:
  int je_prazdna();
  atd.
}
```

Poznámka:

- konstruktor vložené třídy TPolozka implementujeme:

```
TFronta::TPolozka::TPolozka(int novy_prvek)
{
  prvek = novy_prvek;
  dalsi = NULL;
}
```

Více o konstruktorech a o přiřazení ...

- inicializovat objekt lze i pomocí jiného objektu
  - konstruktor se nespouští
  - provádí se bitová kopie všech atributů
- lze provést přiřazení mezi objekty
  - provádí se bitová kopie všech atributů

```
#include "TKomplex.h"
void main(void)
{
  Komplex c1(3,4);
  Komplex c2 = c1;
  Komplex c3;

  c1.re = -4;
  c1.im = 5;

  c3 = c1;
  }
```

zde se provádí bitová kopie objektů, nevolá se konstruktor

zde se provádí bitová kopie objektů

Bitová kopie může někdy způsobovat problémy!

- uvažujme třídu, pomocí které implementujeme n-rozměrný vektor
- vektor je představován dynamicky alokovaným polem

```
class TVektor
{
  int velikost;
  int *vektor;
 public:
  TVektor();
  TVektor(int vel);
  ~TVektor();
  int vrat_vel();
  int pricti(TVektor v);
  void zmen_velikost(int nova_vel);
  void nastav_slozku(int index, int hodn);
  int vrat_slozku(int index);
};
```

```
#include "TVektor.h"
void main(void)
{
  TVektor v1(2), v2(2);
  v1.nastav_slozku(0,-4);
  v1.nastav_slozku(1,2);
  v2.nastav_slozku(0,1);
  v2.nastav_slozku(1,2);
  v1.pricti(v2); //ve v1 bude (-3,4)
  int i;
  for(i=0;i<v1.vrat_vel();i++)
   cout << v1.vrat_slozku(i) << endl;
}
```

```
TVektor::TVektor()
{
  velikost = 10;
  vektor = new int[10];
}
TVektor::TVektor(int vel)
{
  velikost = vel;
  vektor = new int[vel];
}
TVektor::~TVektor()
{
  delete [] vektor;
};
```

```
void TVektor::nastav_slozku(int index, int
  hodn)
{
  if (index < velikost && index >= 0)
    vektor[index] = hodn;
}

int TVektor::vrat_slozku(int index)
{
  if (index < velikost && index >= 0)
    return vektor[index];
  else return -1;
}
```

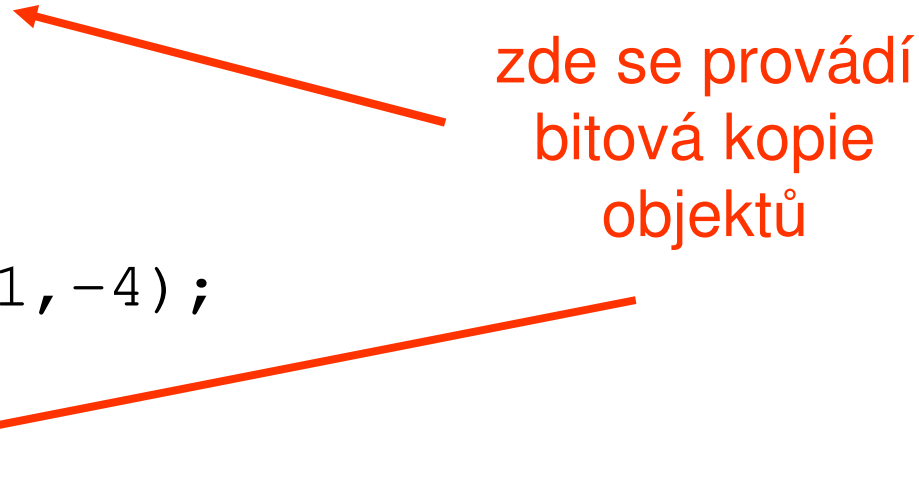
```
int TVektor::vrat_vel()
{
  return velikost;
}
int TVektor::pricti(TVektor v)
{
  if (velikost==v.velikost)
  {
    for(int i=0;i<velikost;i++)
      vektor[i] += v.vektor[i];
    return 1;
  }
   return 0;
}
```

POZOR!

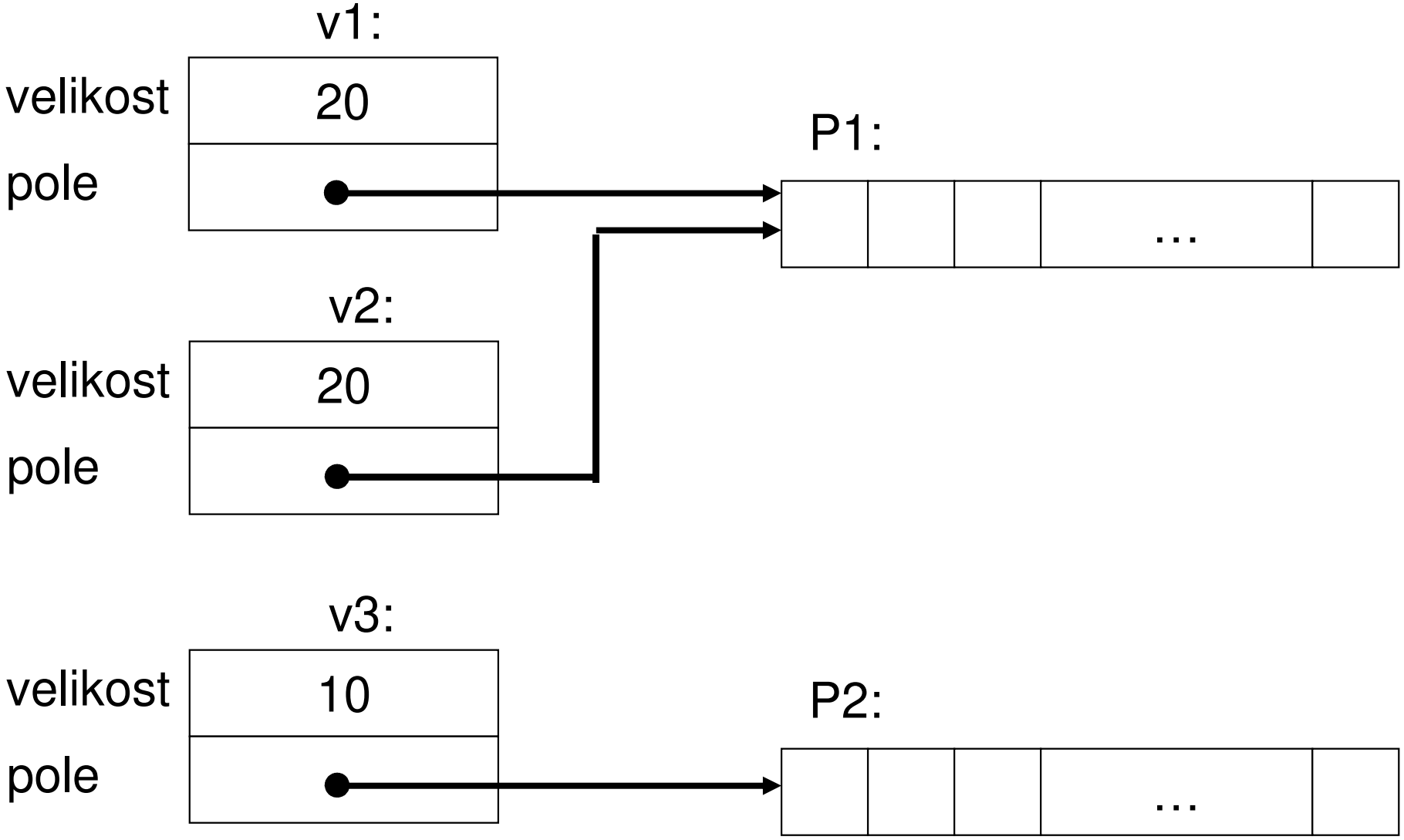
viz dále

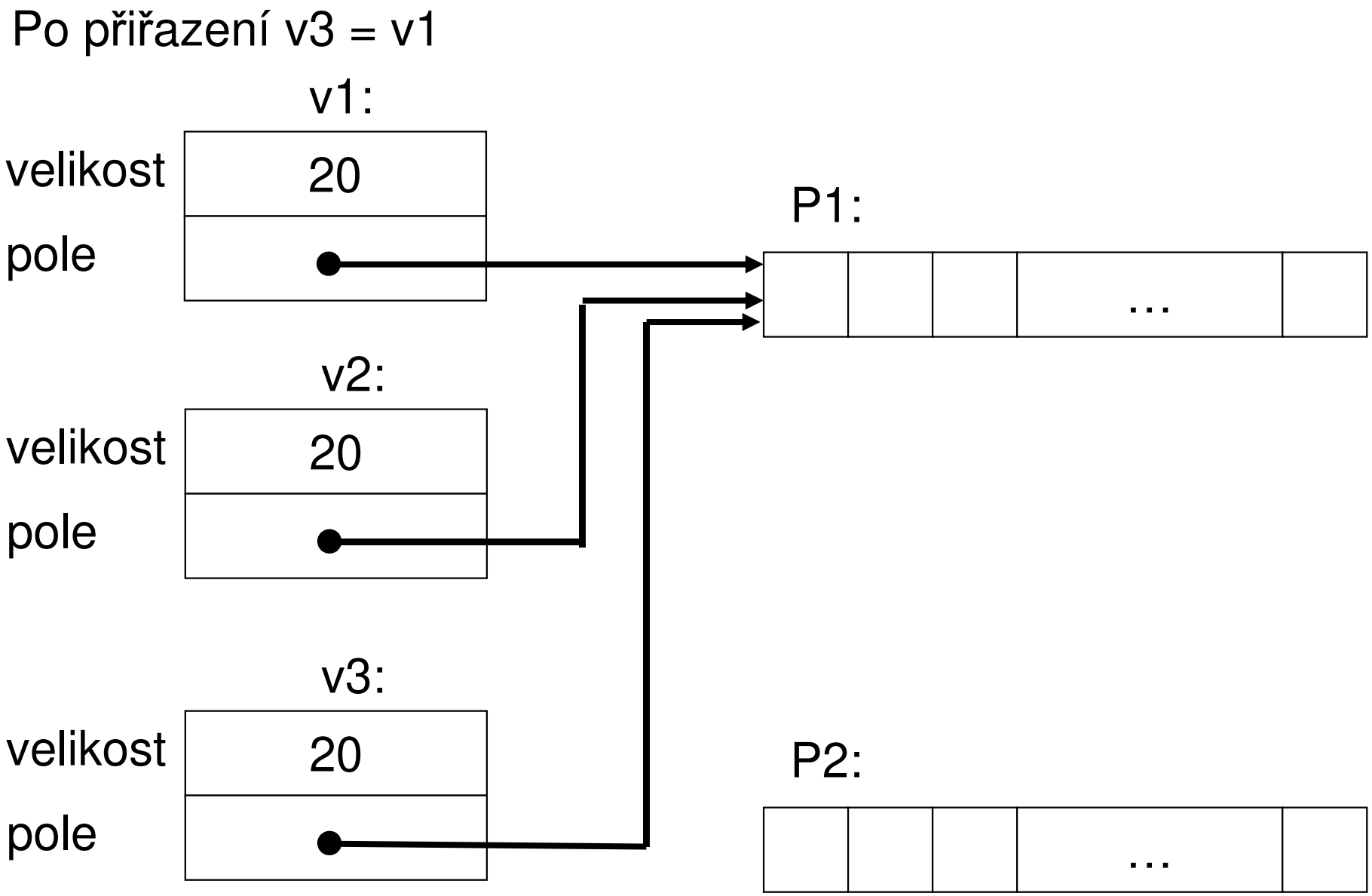
```
void TVektor::zmen_velikost(int nova_vel)
{
  if (nova_vel > velikost)
  {
    int *novy = new int[nova_vel];
    memcpy(novy,vektor,sizeof(int)*velikost);
    delete []vektor;
    vektor = novy;
  }
  velikost = nova_vel;
}
```

```
#include "TVektor.h"
void main(void)
{
  TVektor v1(20);
  TVektor v2 = v1;
  TVektor v3(10);

  v1.nastav_slozku(1,-4);

  v3 = v1;
  v1.pricti(v2);
}
```



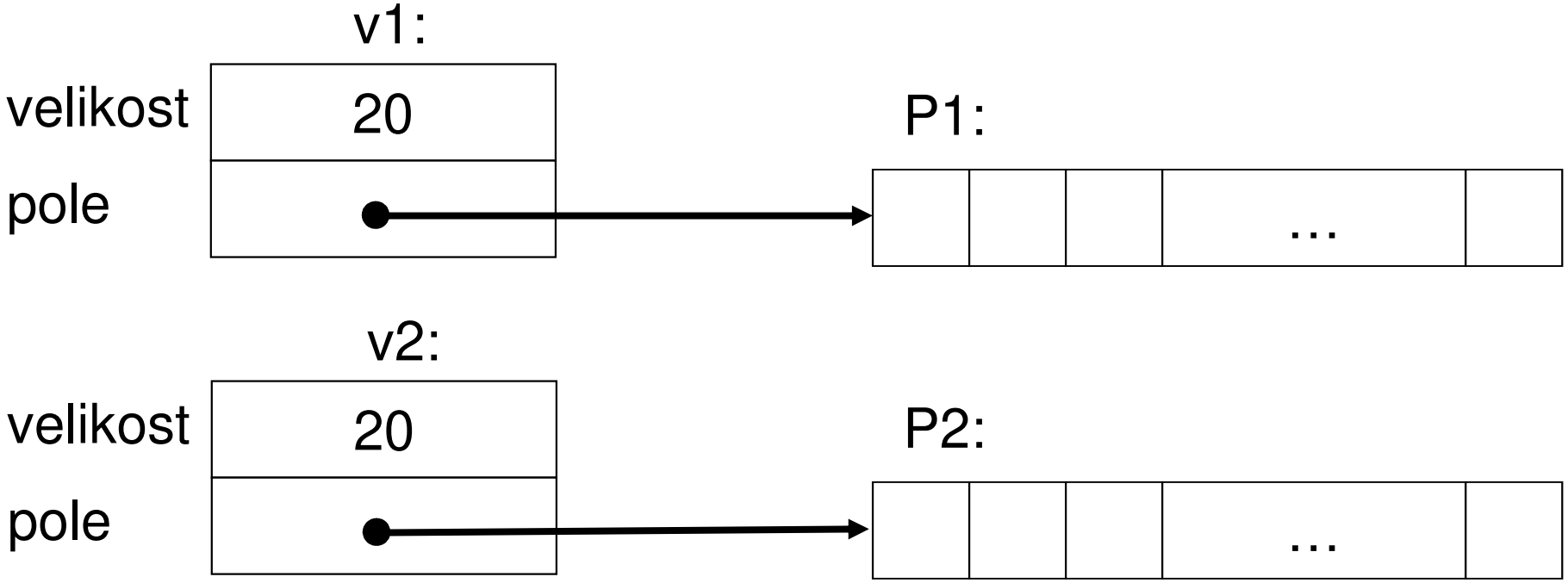
Po vytvoření objektů …




Po přiřazení v3 = v1
v1:
velikost
20
pole
P1:
...
v2:
velikost
20
pole
v3:
velikost
20
pole
P2:
...

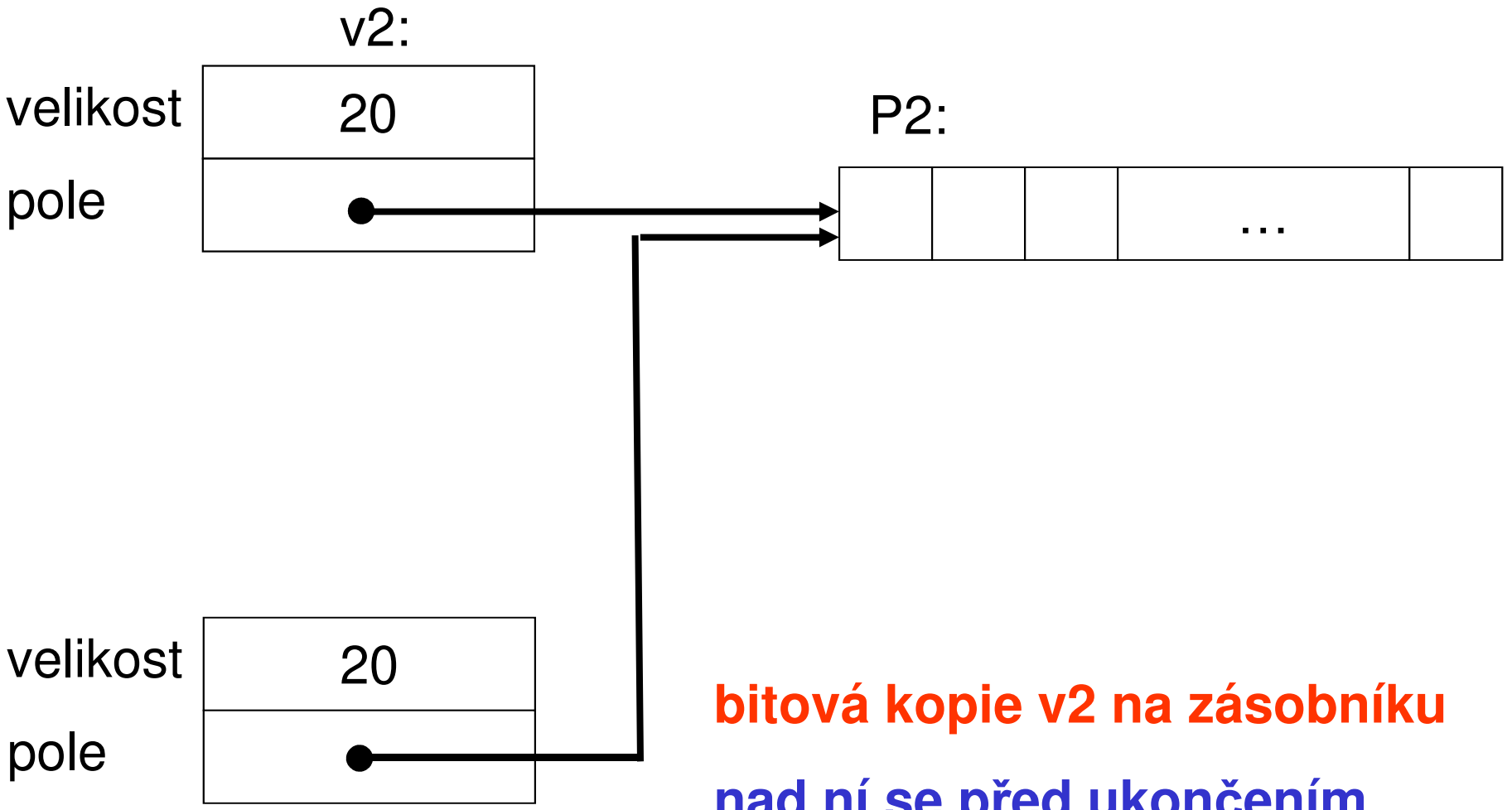
- ukazatel `vektor` ukazuje u instancí `v1`, `v2` a `v3` na stejné dynamické pole
  - při změně hodnot v2 se mění i v1
- u instance v3 jsme ztratili ukazatel na alokované pole P2; paměť již programátor nemůže vrátit operačnímu systému

Mějme dva vektory



Při volání metod může nastat ještě závažnější problém…

- při volání metody `v1.pricti(v2)` se na zásobník uloží lokální kopie objektu v2:
  `{20,vektor}`
  - ukazatel `vektor` ukazuje na pole P2
- při ukončení metody se ruší lokální kopie objektu v2, tedy je nad touto kopií zavolán **destruktor**
- protože kopie ukazatele ukazuje na P2, je toto **pole P2 dealokováno!**
  - **paměť pro P2 může být pak přidělena jinému programu a obsah přepsán!**



bitová kopie v2 na zásobníku

nad ní se před ukončením metody `pricti` zavolá destruktor, tj. dealokuje se P2

Řešení

- předáváme-li objekt jako parametr metodám (obecně procedurám a funkcím), předáme jej odkazem (jako typ reference) nebo pomocí ukazatele:

```
TVektor::pricti(TVektor &v)
```

nebo

```
TVektor::pricti(TVektor *v)
```

- pro inicializaci objektu jiným objektem při vytvoření deklarujeme zvláštní typ konstruktoru, tzv. **kopírující konstruktor (copy constructor)**
- kopírující konstruktor má jediný parametr, a to *(konstantní) referenci na jiný objekt téže třídy*
- kopírující konstruktor se vyvolá vždy, když vytvořený objekt má být inicializován kopií jiného objektu téže třídy (např. také při předání objektu jako parametru do procedur)

```
class TVektor
{
  int velikost;
  int *vektor;
 public:
  TVektor(const TVektor &v);
  atd.
};
TVektor::TVektor(const TVektor &v)
{
  velikost = v.velikost;
  vektor = new int[v.velikost];
  memcpy(vektor,v.vektor,sizeof(int)*velikost);
}
```

kopírující konstruktor

```
#include "TVektor.h"
void main(void)
{
  TVektor v1(20);
  TVektor v2 = v1;
  TVektor v3(10);

  v1.nastav_slozku(-4,1);

  v3 = v1;
}
```

zde se implicitně volá kopírující konstruktor

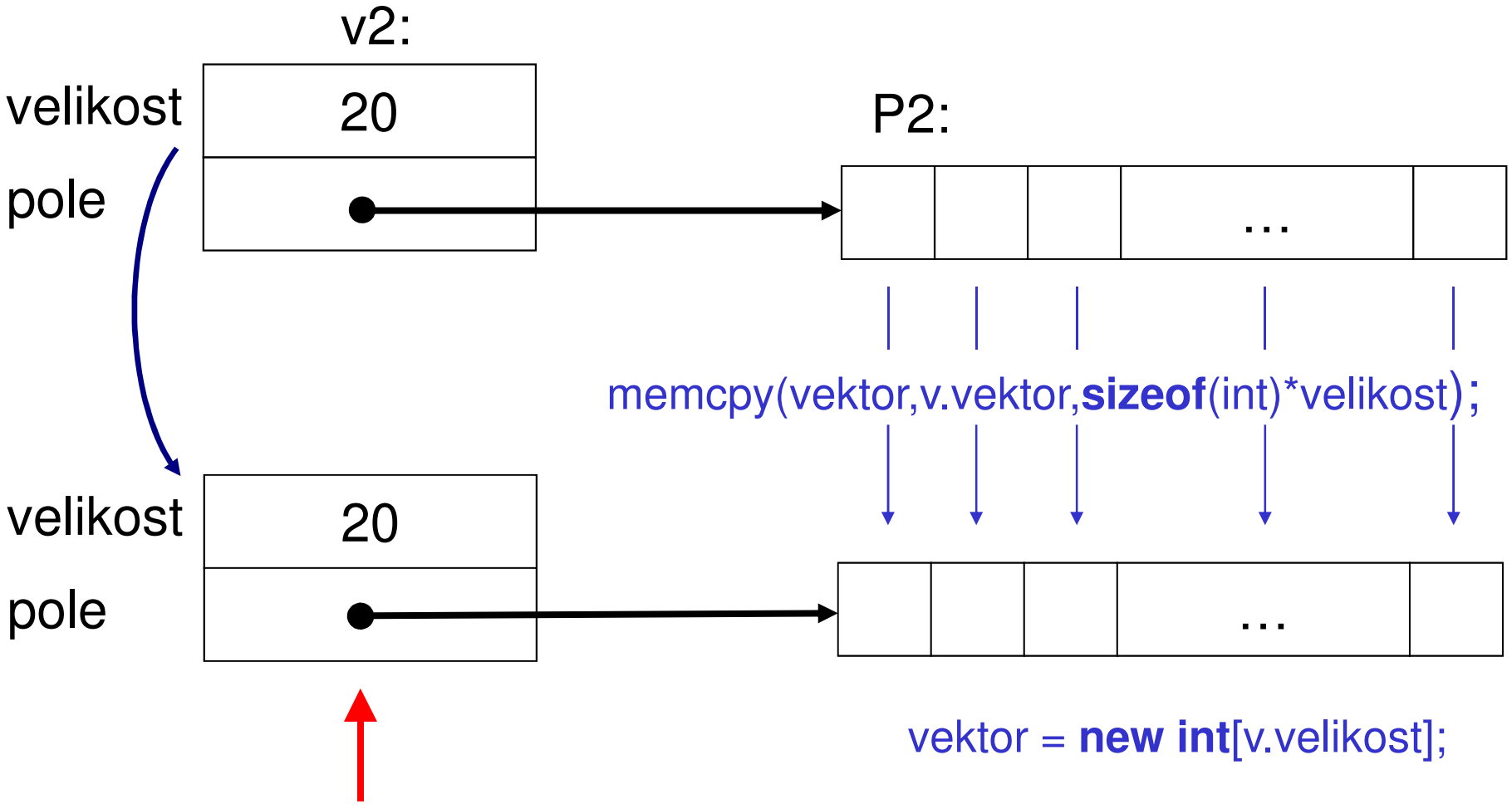
zde se **nevolá** kopírující konstruktor

- máme-li deklarován kopírující konstruktor, pak deklarace funkce

  ```
  pricti(TVektor v)
  ```

  nepřináší problémy, protože při inicializaci lokální kopie objektu na zásobníku skutečným parametrem se provádí voláním kopírujícího konstruktoru
- problém u přiřazení vyřešíme:
  - vytvořením speciální metody `prirad` a operátor = nebudeme používat
  - přetížením operátoru = vzhledem ke třídě TVektor, což se budeme učit někdy příště…

v1.pricti(v2) s kopírujícím konstruktorem



lokální kopie v2 na zásobníku se vytvoří pomocí kopírujícího konstruktoru

Podobné problémy mohou nastat, vrací-li funkce objekt ...

```
#include "Komplex.h"
//funkce vrací objekt
Komplex Komplex::soucet(Komplex &c2)
{
  Komplex c;
  c.re = re + c2.re;
  c.im = im + c2.im;
  return c;
}
```

```
void main()
{
  Komplex c1(3,4),c2(-1,-1),s;

  s = c1.soucet(c2);
  cout << "Vysledek: " << s.re << showpos <<
  s.im << "i" << noshowpos << endl;
}
```

Co se v metodě `soucet` děje ?

- na zásobník se předají jako parametry *odkaz* na objekt c2 a („skrytě“) na c1
- po vstupu se na zásobníku vytvoří lokální objekt c (zavolá se implicitní konstruktor)
- po skončení funkce vrací objekt c, tj. *bitovou kopii atributů*, nad lokálním objektem c je vyvolán *destruktor*

- bitová kopie lokálního objektu c je po ukončení funkce `soucet()` v hlavním programu uložena v dočasném objektu, který je přiřazen (opět bitovou kopií) do objektu s; nad dočasným objektem je opět volán *destruktor*
- zde to nevadí, ale u objektů, kde je např.v konstruktoru alokována dynamická paměť za běhu, jsou problémy zřejmé
  - pomůže kopírující konstruktor

Co z toho vyplývá?

Abychom dobře programovali v jazyku C/C++, musíme znát detaily jazyka a vnitřní mechanismy (volání metod, předávání parametrů). Při tvorbě programů musíme mít stále na zřeteli, co se „děje uvnitř“. Jinak se nám může stát, že program nebude pracovat správně, havaruje apod. a budeme překvapeni jeho chováním. Bez znalosti detailů budeme pracně hledat příčinu.

Řešení

1. kopírující konstruktor + přetížení operátoru "=„

nebo

2. funkce vrací místo objektu ukazatel, resp. referenci na objekt
   - nesmí vracet ukazatel na objekt, který je lokální v proceduře, tj. bude zrušen

- špatně:

```
Komplex *funkce()
{
  Komplex c;

  kód

  return &c;
}
```

- vracíme ukazatel na objekt, který je lokální ve funkci a po jejím skončení zaniká

- správně:

```
Komplex *funkce()
{
  Komplex *c;
  c = new Komplex;

  kód

  return c;
}
```

- o uvolnění paměti se musí postarat např. hlavní program

Použití:

```
void main()
{
  Komplex c1(3,4),c2(-1,-1),*s;

  s = funkce();


  delete s;
}
```

Poznámka – přetížení operátoru

```
class Komplex
{
 public:
    float re, im;
    …
    Komplex Komplex::operator+(Komplex &c2)
    …
};
```

Poznámka – přetížení operátoru

```
#include „Komplex.h"
//operátor vrací objekt
Komplex Komplex::operator+(Komplex &c2)
{
  Komplex c;
  c.re = re + c2.re;
  c.im = im + c2.im;
  return c;
}
```

```
void main()
{
  Komplex c1(3,4),c2(-1,-1),s;

  s = c1+c2;
  // lze napsat c1.operator+(c2);
  cout << "Vysledek: " << s.re << showpos <<
  s.im << "i" << noshowpos << endl;
}
```