Více o konstruktorech a destruktorech …

Více o konstruktorech a o přiřazení ...

- inicializovat objekt lze i pomocí jiného objektu
- lze provést přiřazení mezi objekty
  - v „původním“ C nebylo možné provést přiřazení mezi proměnnými typu struktura, bylo nutné provést kopii po položkách
- v obou případech se provádí bitová kopie všech atributů

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

  c1.set_num(-4,5);

  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;
}

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

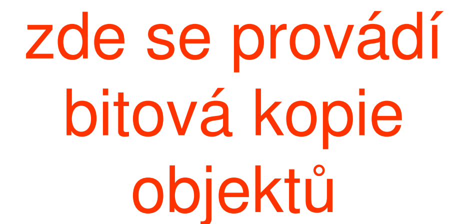
```
int TVektor::vrat_vel()
{
  return velikost;
}
int TVektor::pricti(TVektor v)
{
  if (velikost==v.velikost)
  {
    for(int i=0;i<velikost;i++)
      vektor[i] += v.vektor[i];
    return 1;
  }
   return 0;
}
```

POZOR!

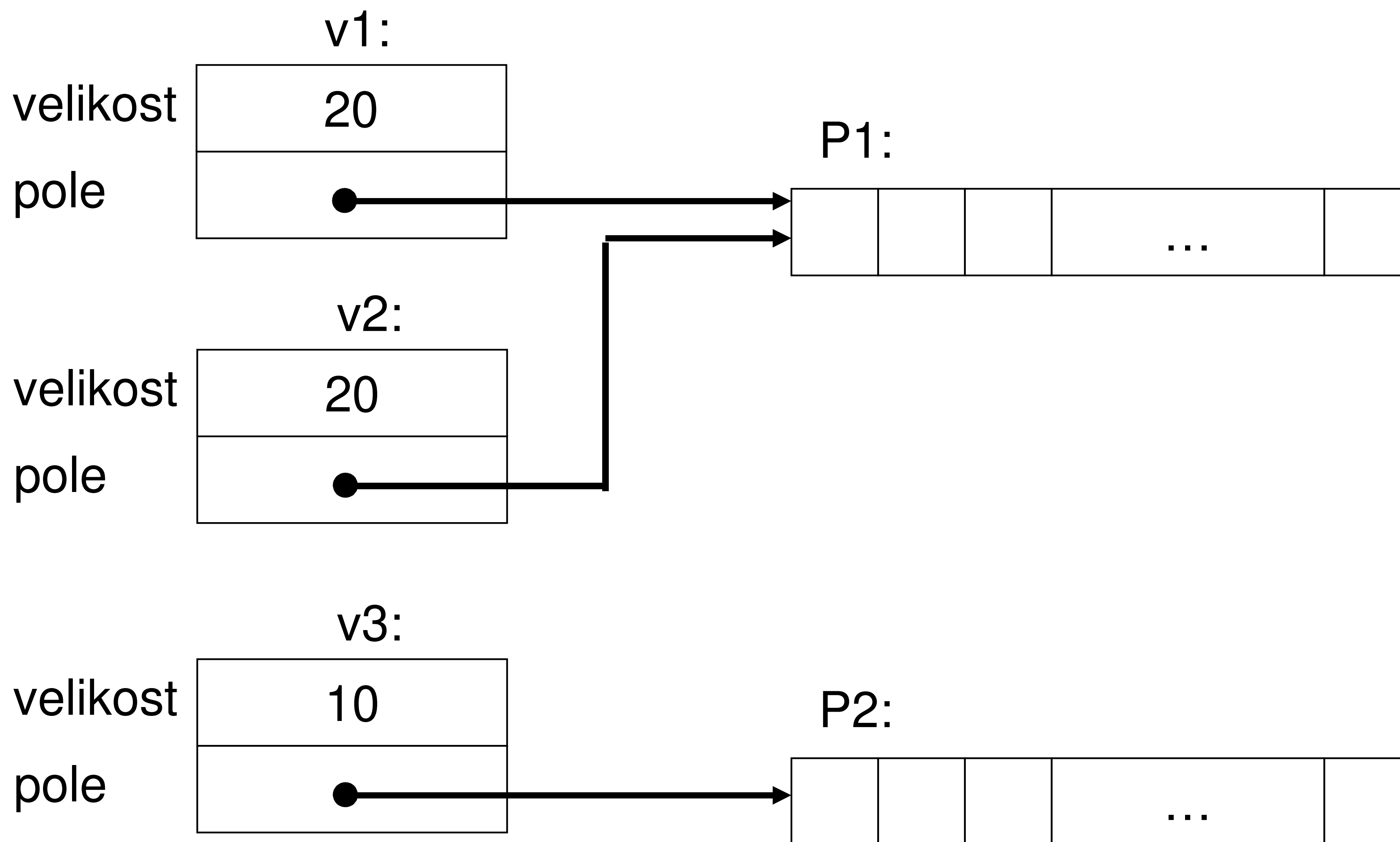
viz dále

```
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;
}
```

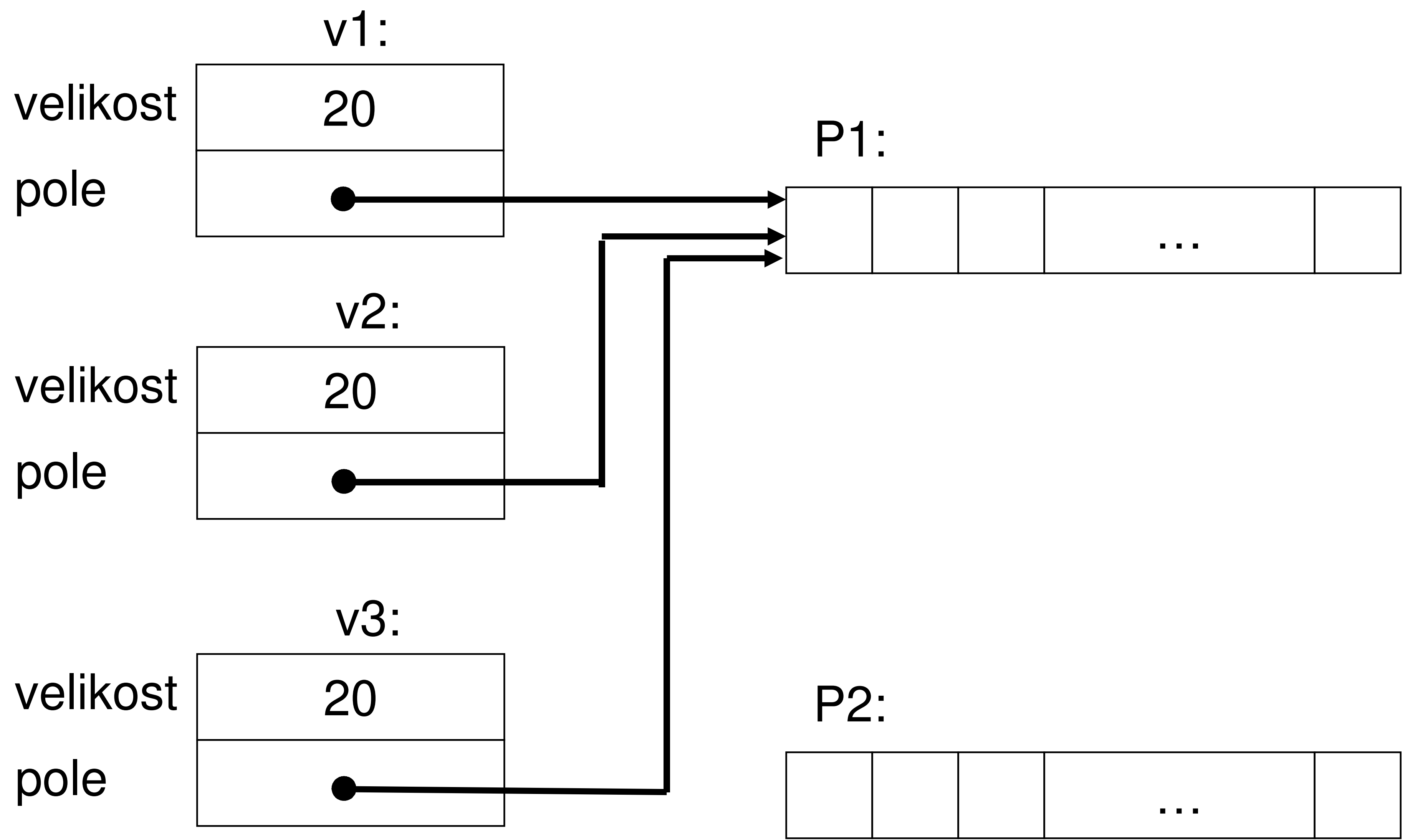
```
#include "TVektor.h"
void main(void)
{
  TVektor v1(20);
  TVektor v2 = v1;
  TVektor v3(10);

  v1.nastav_slozku(-4,1);

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



Po vytvoření objektů …

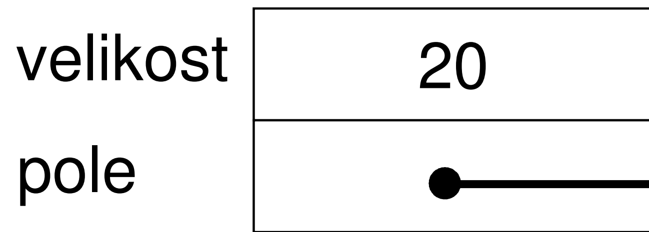


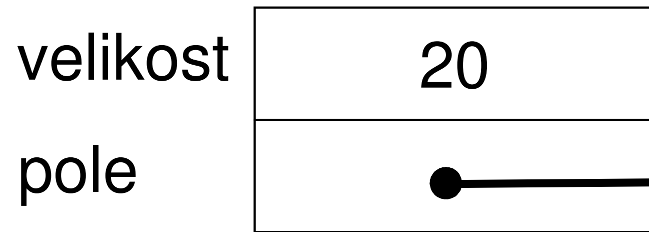
Po přiřazení v3 = v1

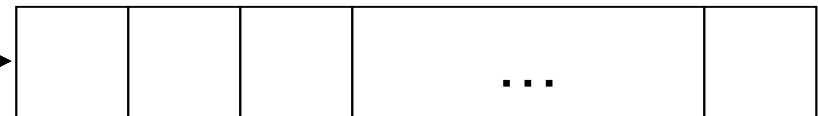


- ukazatel `vektor` ukazuje u instancí `v1`, `v2` a `v3` na stejné dynamické pole
  - při změně hodnot v2 se mění i v1
- u instance v3 jsme ztratili ukazatel na alokované pole P2; paměť již programátor nemůže vrátit operačnímu systému

Mějme dva vektory

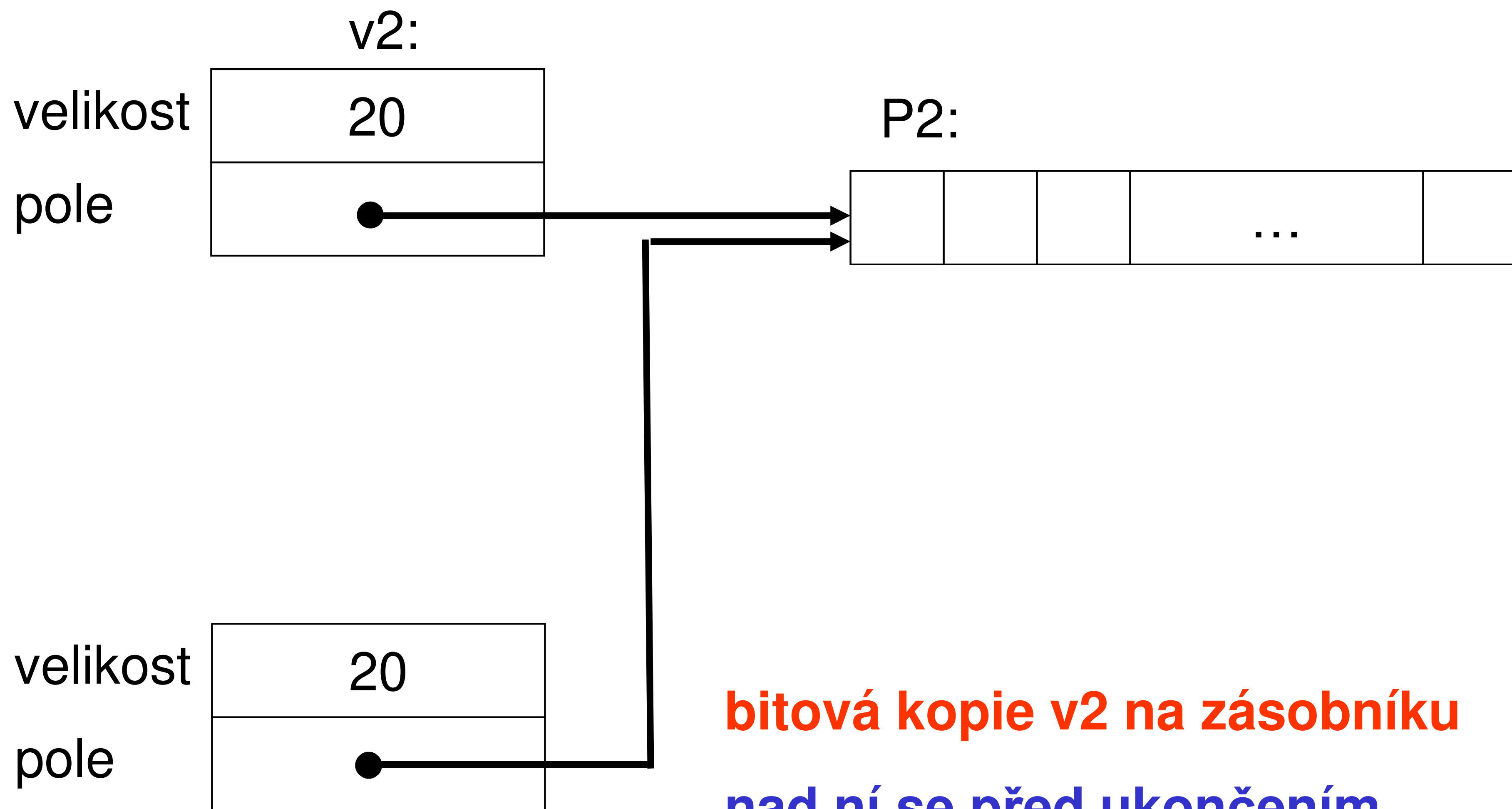






Při volání metod může nastat ještě závažnější problém…

- 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!**



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

```
#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 (řádek `TVektor v2 = v1;`)

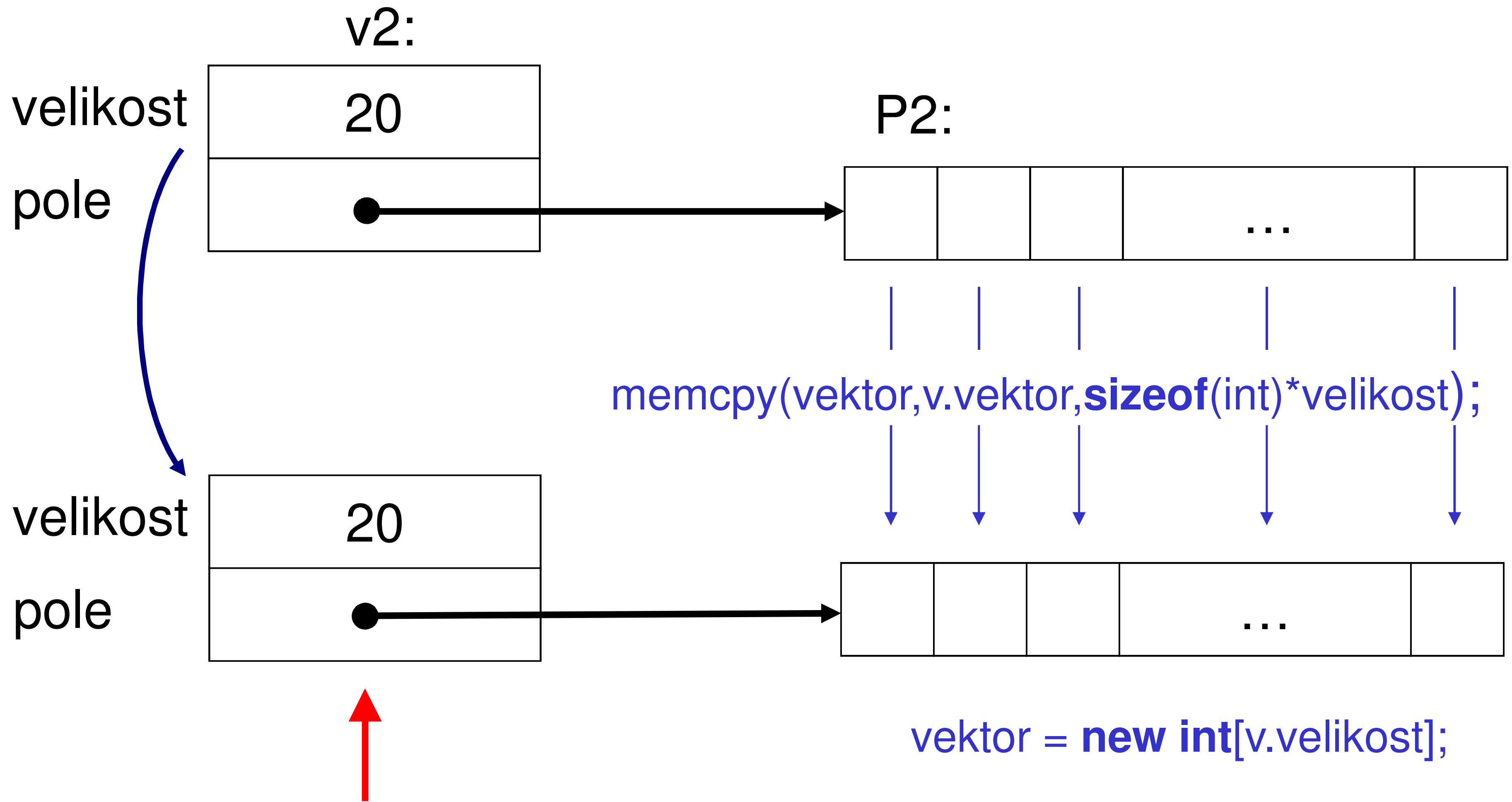
zde se **nevolá** kopírující konstruktor (řádek `v3 = v1;`)

- 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

Úkol

Naimplementujte a vyzkoušejte třídu vektor. Doplňte do konstruktorů a destruktorů jednoduché výpisy typu „Volání konstruktoru“ a vyzkoušejte, zejména u funkce pricti, je-li parametrem objekt nebo reference na objekt. Doplňte kopírující konstruktor a opět pomocí výpisu hlášení vyzkoušejte, kdy se volá.

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

```
#include "TKomplex.h"
//funkce vrací objekt
TKomplex k_soucet(TKomplex &c1, Tkomplex &c2)
{
  TKomplex c;
  c.set_num(c1.vrat_real()+c2.vrat_real(),
            c1.vrat_imag()+c2.vrat_imag());
  return c;
}
```

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

  s = k_soucet(c1,c2);
  printf("Vysledek: %f + %fi",
  s.vrat_real(),s.vrat_imag());
}
```

Co se ve funkci `k_soucet` děje ?

- na zásobník se předají jako parametry *odkazy* na objekty c1 a c2
- 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 `k_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 "="
2. funkce vrací místo objektu ukazatel, resp. referenci na objekt
   - nesmí vracet ukazatel na objekt, který je lokální v proceduře, tj. bude zrušen

- špatně:

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

  kód

  return &c;
}
```

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

- správně:

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

  kód

  return c;
}
```

- o uvolnění paměti se musí postarat např. hlavní program

Použití:

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

  s = funkce();



  delete s;
}
```