Programovací jazyk C++

Cvičení 2

Ačkoliv opakujeme jazyk C, naučíme se již něco z C++

1. Komentáře:

- komentáře v jazyce C: /* */
  - nesmějí se vnořovat
  - je možné je využívat i v C++
  - používají se hlavně jako víceřádkové komentáře
- navíc v C++: jednořádkové komentáře

  // platí do konce řádku

2. Konzolový vstup a výstup pomocí streamů

- v jazyce C++ jsou definovány tři streamy (proudy) v knihovně `iostream`:
  - výstupní (na stdout) `cout`
  - vstupní (ze stdin) `cin`
  - chybový(na stderr) `cerr`
- k proudům se váží dva operátory <<, >>, jejichž význam je pro tento účel předefinován (tzv. **přetížení operátorů**)

- proudy `cout, cin, cerr` jsou objekty (tři proměnné objektového typu `ostream` a `istream`, již deklarované v knihovnách)
  - nejde tedy o příkazy
  - funkcionalita je skryta v přetížení operátorů (tj. k přiřazení funkce k operátorům << a >> vzhledem k `ostream` a `istream`
- nic nebrání nadále používat funkce `printf, scanf`
- často se plete << a >>

- příklad:

```
#include <iostream>
int main(void)
{
   int pl,pz;
   cout << "Zadejte pocet lidi a
  zvirat: ";
   cin >> pl >> pz;
   cout << "Pocet lidi je " << pl <<
   ", pocet zvirat je " << pz << '.';
}
```

Mnemotechnická pomůcka

při výstupu data směřují
do streamu

cout << "Ahoj";

při vstupu data směřují **ze**
streamu do proměnné

cin >> pz;

Manipulátory

- slouží pro řízení vstupní a výstupní konverze
- jsou definovány v `iomanip`

```
// nový řádek
cout << "Ahoj" << endl;
// nastaví šestnáctkový výpis
cout << hex << x;
// vstup čísla v šestn. soustavě
cin >> hex >> a;
```

`dec`	dekadická konverze
`hex`	šestnáctková konv.
`oct`	osmičková konverze
`endl`	konec řádku + “flush”
`setw(int n)`	šířka položky *n* znaků
`setfill(int c)`	plnící znak *c*
`setprecision(int n)`	*n* desetinných míst
`showpos`	vypíše znaménko
`boolalpha`	vypíše true/false

- více v nápovědě

Jmenné prostory

Prostory jmen

- při vkládání několika hlavičkových souborů může vzniknout kolize

zeleznice.h

```
const int x=10;
typedef enum
{
   STOP,CONT
} Stavy;
```

logika.h

```
const int x=5;
typedef enum
{
   DELI,NEDELI
} Stavy;
```

```
#include "zeleznice.h"
#include "logika.h"
void main(void)
{

  Stavy s1,s2;
  int a = x+3;
}
```

Které Stavy?

ze zeleznice.h
nebo z hradlo.h

- řešením je vnoření deklarace do prostoru jmen

zeleznice.h

```
namespace Zeleznice {
   const int x=10;
   typedef enum
   {
      STOP,CONT
   } Stavy;
}
```

logika.h

```
namespace Logika {
   const int x=5;
  typedef enum
   {
      DELI,NEDELI
   } Stavy;
}
```

- mimo deklarovaný prostor není jméno přístupné, musíme se odkázat na příslušný prostor jmen

```
Logika::Stavy
```

```
#include "zeleznice.h"
#include "logika.h"

void main(void)
{
  Zeleznice::Stavy h1;
  Logika::Stavy h2;
  int a = Logika::x+3;
}
```

- abychom nemuseli psát stále odkaz na prostor jmen, lze dosáhnout přímé viditelnosti pomocí direktivy **`using`**

```
#include "zeleznice.h"
#include "logika.h"

using namespace Zeleznice;
void main(void)
{
  Stavy h1;
  Logika::Stavy h2;
  using namespace Logika;
  int a = x+3;
}
```

- deklarace jmenných prostorů lze vnořovat
- deklarace jsou otevřené, tj. deklarace lze přidávat
  - v prvním souboru: `namespace A { … }`
  - ve druhém souboru: `namespace A { … }`
- pro běžné objekty, např. `cout`, je definován jmenný prostor `std`
  - nesmíme tedy zapomenout na počátku programu uvést

```
using namespace std;
```

- jména lze zkracovat pomocí aliasů:
  - `namespace Zel=Zeleznice;`

Vyzkoušejte…

```
#include <iostream>
#include <iomanip>
int main(int argc, char **argv)
{ int x;
  cout << "Ahoj, svete!" << endl;
  cout << hex << 65 << endl;
  cout << "Zadej cislo v osm. soustave: ";
  cin >> oct >> x;
  cout << dec << x << endl;
  cout << showpos << x << endl;
  cout << setw(5) << setfill('0') << x << endl;
  cout << boolalpha << true;
  return 0;
}
```

- tento kód nepůjde přeložit (překladač napíše, že nezná `cout`)

Vyzkoušejte…

```
#include <iostream>
#include <iomanip>
int main(int argc, char **argv)
{ int x;
  std::cout << "Ahoj, svete!" << std::endl;
  std::cout << std::hex << 65 << std::endl;
  …
  return 0;
}
```

Vyzkoušejte…

```
#include <iostream>
#include <iomanip>

using namespace std;
int main(int argc, char **argv)
{ int x;
  cout << "Ahoj, svete!" << endl;
  cout << hex << 65 << endl;
  cout << "Zadej cislo v osm. soustave: ";
  cin >> oct >> x;
  cout << dec << x << endl;
  cout << showpos << x << endl;
  cout << setw(5) << setfill('0') << x << endl;
  cout << boolalpha << true;
  return 0;
}
```

Úkol č. 1

Přepište program z webových stránek pro výpočet nejmenšího společného násobku s využitím proudů `cin, cout.`

Otestujte správnost zadání vstupu pomocí metody `fail()`:

```
cin >> a;
if (cin.fail()) { cerr << ...;
return -1; }
```

Pole

- statické

```
Typ identifikator[velikost];
int a[20];
```

- definice typu pole

```
typedef int TPole20[20];
```

pak mohu nadeklarovat pole:

```
TPole20 pole;
```

- přístup k prvkům `a[i], pole[10]`
- rozsah indexů: `0 .. n-1`
- meze polí se nekontrolují
- pole je v jazyce C i C++ reprezentováno ukazatelem na počátek pole

Otázka:

Co to znamená, napíši-li do programu identifikátor `pole` a jaká je jeho hodnota ?

`pole` je konstantní ukazatel, obsahem je adresa počátku pole.

Načítání do pole

```
int a[10];
```

- pomocí `scanf:`

  ```
  scanf("%d",&a[i]);
  ```

  C-čkovštěji: `scanf("%d",a+i);`

- pomocí `cin:`

  ```
  cin >> pole[i];
  ```

- v jazyce C je možnost definovat prvky pole při deklaraci (konstruktor pole):

```
int a[3] = {5,-6,10};
```

Vícerozměrné pole

- deklarace

```
int x[10][10];
```

- konstruktor pole

```
int m[2][2] = {{1,2},{-5,6}};
```

- vícerozměrné pole je v jazyce C uloženo po řádcích
- `x[3][5]` je stejné jako `*(*(x+3)+5)`

Ukazatelé, dynamická alokace paměti

- typ ukazatel – speciální datový typ

Otázka:
Jaká je hodnota proměnných typu ukazatel?

Je to adresa do paměti, tedy obsahem není přímo hodnota, ale ukazatel (odkaz) na místo, kde se hodnota nachází

- deklarace

```
int *pi;
typedef float* Pfloat;
Pfloat pf;
```

Kam ukazují pi a pf?

- alokace paměti

```
void *malloc(size_t  size)
void *calloc(size_t num, size_t size)
pi = (int*)malloc(sizeof(int)*n);
pi = (int*)calloc(n,sizeof(int));
```

```
void *realloc(void *block,
              size_t size);
```

- při nedostatku paměti vrací `malloc()` `NULL`
- co znamená:
  - `void*`
  - `sizeof()`
- přístup k hodnotě `*pi = 3;`
- dealokace paměti `free(pi);`
- co znamená: operátor `&`

```
float f;
pf = &f;
```

- ukazatelová aritmetika - co znamenají:

```
pi++;
```
pozor, abychom neztratili informaci o původní adrese!

```
pi = pi+2;
pi-1
*(pi-3)      int x=*(pi-3);
(char*)(pi)
char c;
c = *((char*)(pi))
```

- přístup ke statickým a dynamickým polím jsou v jazyce C (C++) stejný
- tyto zápisy jsou totožné:

```
pi[3]       *(pi+3)
pi[0]       *pi
pi[-1]      *(pi-1)
&pi[2]      pi+2
```

Dynamická alokace dvourozměrného pole

- alokace (`n` řádek, `m` sloupců)

```
int **matice;
matice = (int**)malloc(n*sizeof(int*));
for(int i=0;i<n;i++)
  matice[i] = (int*)malloc(m*sizeof(int));
```

- dealokace

```
for(int i=0;i<n;i++) free(matice[i]);
free(matice);
```

```
int main(void)
{
  int i,j,n,m,min; int **matice;
  printf("Zadej pocet radku matice: ");
  scanf("%d",&n);
  printf("Zadej pocet sloupcu matice: ");
  scanf("%d",&m);
  matice = (int**)malloc(n*sizeof(int*));
  for(i=0;i<n;i++)
    matice[i] = (int*)malloc(m*sizeof(int));

  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
      scanf("%d",&matice[i][j]);
```

```
  /* hledam minimum */
  min = matice[0][0];
  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
      if (matice[i][j]<min) min=matice[i][j];

  printf("Minimum je: %d\n",min);
  /* dealokace */
  for(i=0;i<n;i++) free(matice[i]);
  free(matice);
}
```

Alokace paměti v C++

- kromě `malloc` existuje v C++ operátor new
  - oproti funkci `malloc` je operátor new přetížen, tj. je předefinovaná jeho funkce pro určitý datový typ
- dynamická alokace

```
int *p;
p = new int;
```

- dynamická alokace pole

```
int *pi;
pi = new int[velikost];
```

- **velikost je v položkách, nikoliv ve slabikách!**
  - důvod: new je přetížen, tj. uvažování velikosti typu je „schováno“ v přetěžování
- pomocí new dynamicky alokujeme i objekty
  - dynamicky se alokovat objekty pomocí malloc nedají)
- paměť alokovaná pomocí new se dealokuje pomocí delete

```
delete p;
delete [] pi;
```

příznak rušení pole

- operátorem `delete` lze dealokovat jen to, co bylo alokováno operátorem **`new`**

```
int *pa = p+1;
delete pa;
```

- při nemožnosti alokovat paměť:
  - vyhodí se výjimka (rys C++, který poznáme později)
    - původní verze operátoru `new`
  - od verze jazyka 98 existuje i operátoru vracející NULL (nevyhazující výjimku)
  - `p = new (std::nothrow) int[vel]`

```
int main(void)
{
  int i,j,n,m,min; int **matice;
  cout << "Zadej pocet radku matice: ";
  cin >> n;
  cout << "Zadej pocet sloupcu matice: ";
  cin >> m;
  matice = new int*[n];
  for(i=0;i<n;i++)
    matice[i] = new int[m];

  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
      cin >> matice[i][j];
```

```
  /* hledam minimum */
  min = matice[0][0];
  for(i=0;i<n;i++)
    for(j=0;j<m;j++)
      if (matice[i][j]<min) min=matice[i][j];

  cout << "Minimum je " << min << endl;
  /* dealokace */
  for(i=0;i<n;i++) delete [] matice[i];
  delete matice;
}
```

Procedury a funkce v C

- deklarace funkce
  - `typ identifikator(parametry)`
  - `int vypocet(int x, int y)`
    `{ tělo }`
  - hodnota se vrací pomocí **`return`**
- deklarace hlavičky procedury v C:
  - **`void`** `tiskni(`**`void`**`)`
  - `tiskni()` – v C-čku jde o funkci s návratovou hodnotou **`int`**, o parametrech se nic neříká

  - **v C++** se musí explicitně deklarovat vždy návratový typ
  - **v C++** deklarace `char vrat_znak()` znamená, že funkce parametry nemá
- v C++ funkce musí vracet hodnotu (**`return`** nesmí chybět)
- v C++ musí existovat prototyp funkce

- v jazyce C se přeloží, ale v C++ ne:

```
void main(void)
{ int x;
  x= sum(3,4);
}
int sum(int a, int b)
{
  return a+b;
}
```

- v jazyce C++ musím zapsat takto:

```
int sum(int a, int b)
{
  return a+b;
}
void main(void)
{ int x;
  x= sum(3,4);
}
```

- nebo takto:

```
int sum(int a, int b); //prototyp

void main(void)
{ int x;
  x= sum(3,4);
}

int sum(int a, int b)
{
  return a+b;
}
```

Předávání parametrů

Otázka:

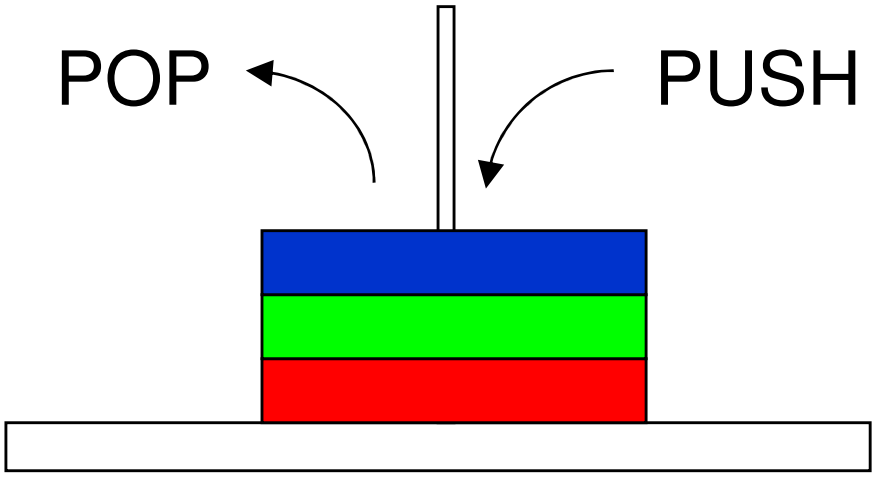
Jak se předávají skutečné parametry funkcí?

Odpověď: Přes **zásobník**.

- v jazyce C se parametry předávají jen hodnotou, tj. kopií na zásobník
- „výstupní“ proměnné procedur se realizují pomocí ukazatelů

Zásobník (stack)

- datová struktura LIFO (last in – first out)
- odebírá se naposledy uložená hodnota
- operace:
  - PUSH (uložení hodnoty na vrchol zásobníku)
  - POP (odebrání hodnoty z vrcholu zásobníku)



- zásobník se využívá v programech také pro odložení mezivýsledků, automaticky se využívá při volání procedur a funkcí
- instrukce CALL *adr*
  - automaticky se uloží na vrchol zásobníku návratová adresa, provede se skok na adresu *adr*
- instrukce RET
  - návratová adresa se odebere ze zásobníku, na ní se provede návrat

```
int obsah(int a,int b)
{
  return a*b;
}
```

funkce obsah

1234: ...

1240: MUL EAX,EBX

1242: RET

```
void main(void)
{
  x = obsah(3,4);
}
```

hlavní program

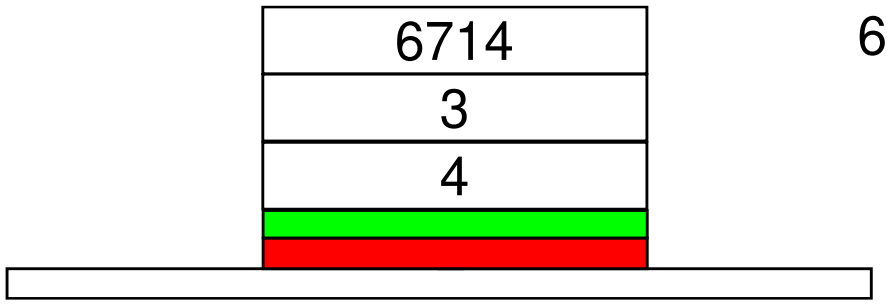
6709: push 4

6710: push 3

6711: call 1234

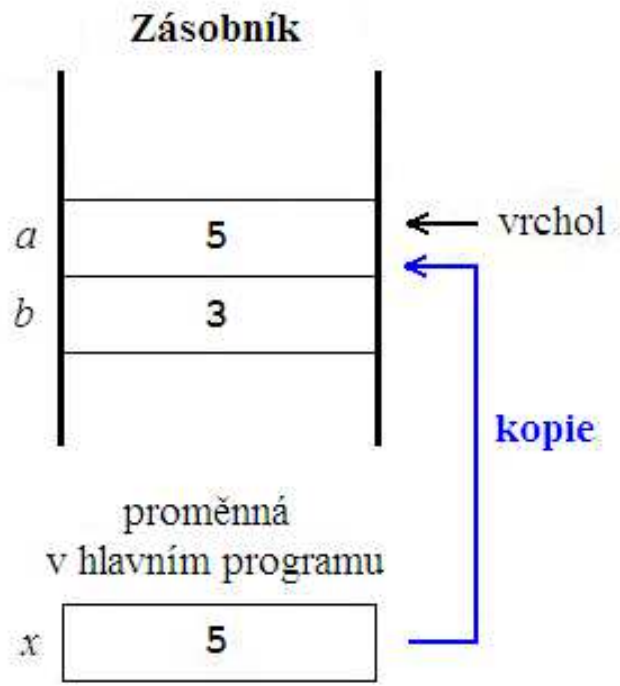
6714: pop

6716: pop

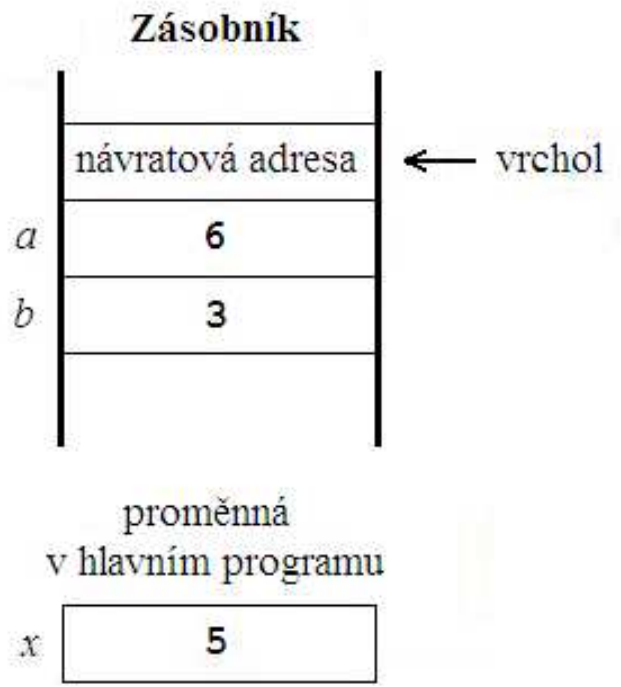


```
void tisk(int a, int b)
{  a++;
   printf("%d\t%d",a,b);
 }
void main(void)
{  int x = 5;
   tisk(x,3);
}
```

Před voláním procedury



Před ukončením procedury



```
void vypocet(int a, int b, int *v1,
  int *v2)
{
 *v1 = a*b;
 *v2 = 2*(a+b);
}
void main(void)
{
  int obsah,obvod;
  vypocet(3,4,&obsah,&obvod);
}
```

- předávání polí – přes ukazatel
- deklarace

```
int max(int n, int *pole);
```

- nebo

```
int max(int n, int pole[]);

void main(void)
{
   int p[10];
   vysl = max(10,p);
}
```

- v jazyce C++ je zaveden typ **reference**, který umožňuje psát v C++ výstupní parametry funkcí podobně, jako v Pascalu parametry volané odkazem (var parametry)
- jazyk C++ dovoluje ještě tzv. přetěžování funkcí (funkce se rozlišují nejen identifikátorem, ale i počtem a typem parametrů)

Typ reference

- proměnná typu reference na *typ T* je synonymem pro jinou proměnnou

```
int  i = 0;
int& ir = i;          // ir ~ i
ir = 2;               // i = 2
```

- proměnná typu reference na *T* musí být inicializovaná, a to proměnnou typu *T*

```
int &ir;  // chyba, ir není inicializována
```

```
float f;
const int ci = 10;

int &ir1 = f;   // chyba, f není typu int
int &ir2 = ci;  // chyba, ci je konstanta
```

- proměnná typu reference po celou dobu existence referencuje stejnou proměnnou (referencuje = odkazuje se na ni)
- typ reference se používá pro parametry procedur a funkcí (nahrazuje parametry volané odkazem)

Konstanta typu reference

- konstanta typu reference na *T* ~ synonymum pro neměnnou hodnotu typu *T*
- konstanta typu reference na *T* může být inicializována konstantou nebo proměnnou typu *T* nebo typu *T1*, jestliže *T1* je kompatibilní vzhledem k přiřazení s *T*. Ve druhém případě se vytvoří dočasný objekt, do kterého se zkonvertuje hodnota daná inicializačním výrazem

```
const int max = 100;
float f = 3.14;
const int& rmax = max;
const int& rf = f;
```
//referencuje se dočasný objekt s hodnotou 3

```
rmax = 10;
```
// chyba, rmax je konstanta

```
rf = 5;
```
// chyba, rf je konstanta

```
int& rmax1 = max;
```
// chyba, rmax1 není konstanta

```
int& rf1 = f;
```
// chyba, rf1 není konstanta

Parametr typu reference

- v procedurách a funkcích nahrazuje parametry volané odkazem
- parametr typu reference představuje ve funkci synonymum *skutečného* parametru a umožňuje hodnotu *skutečného* parametru *změnit*

```
void vypocet1(int a, int b, int *v1, int *v2)
{
  *v1 = a*b;
  *v2 = 2*(a+b);
}


void vypocet2(int a, int b, int &v1, int &v2)
{
  v1 = a*b;
  v2 = 2*(a+b);
}
```

```
void main(void)
{
  int obsah, obvod;

  vypocet1(3,4,&obsah,&obvod);
  vypocet2(3,4,obsah,obvod);
}
```

- předávání polí musíme provádět stylem klasického C

Přetěžování funkcí

- v klasickém jazyku C se funkce rozlišují jen identifikátorem, tj. nelze definovat více funkcí se stejným identifikátorem, ale jinými parametry
- nevýhoda - existence různě pojmenovaných podobných funkcí, např.:

```
int abs(int x);
double fabs(double x);
```

- v C++ můžeme funkci *přetížit*, tj. rozlišit i počtem a typem parametrů

- tj. lze deklarovat funkci se stejným jménem, ale různými parametry (různými typy, s různým počtem)

```
int abs(int x);

double abs(double x);
```

- *funkci nelze přetížit* jen návratovou hodnotou - proč?

```
void tisk(int x)
{
  printf("%d“,x);
}

void tisk(char x)
{
  printf("%c“,x);
}
```

- při volání přetížené funkce se vyvolá ta, jejíž parametry se nejlépe spárují se skutečnými parametry
- párování (matching) parametrů:
  1. přesná shoda typů
  2. shoda typů po roztažení (promotion)

     char $\rightarrow$ int
     short $\rightarrow$ int
     enum $\rightarrow$ int
     float $\rightarrow$ double

3. shoda typů po standardní konverzi

int $\rightarrow$ float
float $\rightarrow$ int
int $\rightarrow$ unsigned
…
int $\rightarrow$ long

4. shoda typů po uživatelské konverzi (přetypování)

- pokud nejlepší párování má více než jedna funkce, ohlásí se chyba (při překladu) - potíže nastávají hlavně, jsou-li skutečným parametrem konstanty

Příklady:

```
void f(float, int);
void f(int, float);
f(1,1);                                 // chyba
f(1.2,1.3);                             // chyba

void f(long);
void f(float);
f(1.1);                                         // chyba
f(1.1F);                                        //
f(float)

void f(unsigned);
void f(int);
void f(char);
unsigned char uc;
f(uc);                                          // f(int)
```

Úkol č. 2

- přetižte funkce z příkladu na webových stánkách a upravte pomocí parametrů nahrazovaných odkazem

Úkol
Napište program pro řazení pole

- implementujte statické a dynam. pole
- v případě dynamického pole zadejte z klávesnice jeho maximální velikost
- – udržujte dále informaci o aktuálním počtu prvků v poli
- napište proceduru vloz pro vkládání dat za poslední prvek (viz další blána)
- pole řaďte metodou Bubble Sort a výběrem maximálního (minimálního prvku)
- využijte procedury a funkce, malloc(), parametry předávejte v klasickém C stylu
- přepište funkce do stylu C++ (operátor new)

```
bool vloz(int n, int& akt,int a[],int prvek);

// n je celkovy pocet prvku pole
// akt je aktualni pocet prvku pole, ktery je po
  vlozeni zvysen o 1
// a je pole
// prvek je vkladany prvek
// funkce vraci tru, jestlize se prvek podarilo
// vlozit, false, je-li pole jiz plne
```