# Základy programování v C++ 14. cvičení

Zuzana Petříčková

26. listopadu 2018

# Přehled

# Ukazatele v C/C++ – pokračování

**Ukazatel (pointer)**

- proměnná, jejíž hodnotou je adresa v paměti počítače
  - ukazatel na data
  - ukazatel na funkci

**Kdy ukazatele využijeme?**

- předávání parametrů funkcí odkazem
- efektivnější práce s poli
- **dnes:** dynamická alokace paměti

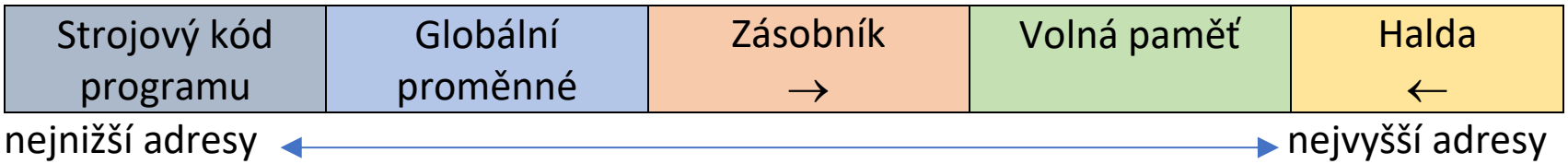
# Proměnné v C/C++

1. **globální**
   - deklarované mimo těla funkcí
   - existují po celou dobu běhu programu (vznikají při jeho spuštění, zanikají při jeho ukončení)
2. **lokální**
   - deklarované uvnitř bloku (mezi { }, např. v těle funkce)
   - existují jen po dobu provádění příkazů daného bloku (vznikají, když program vstoupí do bloku, zanikají při jeho ukončení)
   - uložené v paměti na zásobníku (stack)
3. **novinka: dynamické**
   - vytvoříme je operátorem **new** v místě programu, kde je potřebujeme,
   - zrušíme je operátorem **delete** ve chvíli, kde je přestaneme potřebovat
   - uložené v části paměti zvané halda (heap)

# Proměnné v C/C++

1. **globální**
   - deklarované mimo těla funkcí
   - existují po celou dobu běhu programu
2. **lokální**
   - deklarované uvnitř bloku (mezi { }, např. v těle funkce)
   - existují jen po dobu provádění příkazů daného bloku
   - uložené v paměti na zásobníku (stack)
3. **novinka: dynamické**
   - vytvoříme je v místě programu, kde je potřebujeme, zrušíme je ve chvíli, kde je přestaneme potřebovat
   - uložené v části paměti zvané halda (heap)

Celková paměť vyhrazená pro program a jeho data:



# Dynamické proměnné

**Základní vlastnosti**

- nevznikají deklarací, ale jsou vytvářeny (**alokovány**) a rušeny (**dealokovány**)
- nemají jména, zacházíme s nimi pouze pomocí ukazatelů

**K čemu jsou dobré**

- umožňují práci s daty, o kterých předem nevíme, jak budou velká

# Vytvoření a zrušení dynamické proměnné

```
int *ui = new int;
float *uf;
uf = new float;
...
delete ui;
delete uf;
ui = nullptr;
uf = nullptr;
```

Operátor **new** ... vytvoření (alokace) proměnné

- vyhradí v paměti místo pro proměnnou daného typu a vrátí ukazatel na toto místo

Operátor **delete** ... zrušení (dealokace) jedné proměnné

- uvolní paměť, na kterou ukazuje ukazatel (ale ukazatel nevynuluje → potenciální nebezpečí)

# Vytvoření a zrušení jednorozměrného dynamického pole

```
int dim;
cout << "Zadejte delku pole: " << endl;
cin >> dim;

float *uf = new float[dim];
int   *ui = new int[dim];
...
delete [] ui;
delete [] uf;
ui = nullptr;
uf = nullptr;
```

Operátor **new T[]** ... vytvoření dynamického pole s prvky typu **T**

- vyhradí v paměti místo pro pole daného typu a vrátí ukazatel na toto místo

Operátor **delete []** ... zrušení celého pole

- operátor **delete** by uvolnil paměť jen prvního prvku pole

# Ošetření chybné alokace

operátor **new** v případě neúspěšné alokace paměti

- vrací nulový ukazatel ... v C (a starších verzích C++)
- vyvolá výjimku typu **bad_alloc** ... podle novějších standardů C++

**"Odstrašující" příklad:**

```
#include <iostream>
using namespace std;

int main()
{
    int *u;
    while (true)
        u = new int[100000000];
    // opakovane se alokuje pamet, ale neuvolnuje se
    ...
}
```

# Ošetření chybné alokace

## Správné ošetření situace ... odchycení výjimky

```
#include <iostream>
#include <new> // bad_alloc
using namespace std;

int main()
{
    try
    {
        int * ui = new int[100];
        ... // veskera prace s promennou
        delete [] ui;

    } catch (const bad_alloc& e) {
        cout << "Allocation failed: " << e.what() << endl;
    }
    ...
}
```

# Ošetření chybné alokace bez využití vyjímek

parametr **nothrow**

- pokud chceme, aby operátor **new** v případě neúspěšné alokace v C++ nevyvolal výjimku, ale vrátil nulový ukazatel

**Příklad ... správné ošetření situace:**

```
int main()
{
    int *ui = new (nothrow) int[10];
    if (!ui)
    {
        cout << "Alokace se nezdarila.";
        return -1; // chyba("alokace se nezdarila.");
    }
     ... // veskera prace s promennou
    if (ui)
        delete [] ui;
    ...
}
```

# Uvolňování dynamicky alokované paměti

V jazycích C/C++ není garbage collector
→ **všechny nepotřebné dynamické proměnné musí uvolnit programátor**

- pokud neuvolníme dynamickou proměnnou → napořád nám "sežere" paměť (viz. minulý příklad)
- pokud uvolníme již uvolněnou dynamickou proměnnou → nastane běhová chyba (někdy obtížně odhalitelná)

# Struktury a ukazatele, dynamická alokace

- Přístup ke složkám struktury pomocí operátoru -> nebo **(*u).**

```
...
Zlomek z = {2,3};
Zlomek *uz = &z;
(*uz).citatel = 4; // z.citatel = 4;
uz->citatel = 5;   // z.citatel = 5;
vypis(*uz);        // vypis(z)

uz = new (nothrow) Zlomek;
if (!uz)
    chyba();

(*uz).citatel = 7;
uz->jmenovatel = 6;
vypis(*uz);

delete uz;
...
```

# Odbočka: jemný úvod do výjimek

- ošetření chyb za běhu
- přenos řízení a informace o problému (chybě) z místa, kde problém nastal, do místa, kde ho bude možné řešit / ošetřit
- vyvolání výjimky: **throw výraz**, kde výraz může být libovolného typu (typicky třída exception nebo odvozená)
- zachycení výjimky: blok příkazů **try-catch**

# Výjimky - jemný úvod

```
try
{
    // proved kus kodu, ktery potencialne haze vyjimku
    ...
}
catch (const Typ1 &e) // Typ1 je datovy typ
{
    ... // osetri chybu nebo vypis informaci o chybe
}
catch (const Typ2 &e)
{
    ... // osetri chybu nebo vypis informaci o chybe
}
catch (...) // toto navesti odchyti vsechny zbyle typy vyjime
{
    cout << "Neocekavana chyba" << end;
}
```

# Výjimky - jemný úvod

**Příklad:** odchycení většiny výjimek, které ,,hodí" funkce ze standardních knihoven:

```
try
{
    ...
    int *ux = new int[10];
    ...
}
catch (const bad_alloc &e)
{
    cout << "Chyba alokace: " << e.what() << endl;
}
catch (const exception &e) // jina standardni vyjimka
{
    cout << "Chyba: " << e.what() << endl;
}
```

# Výjimky - jemný úvod

**Příklad: ošetření chyby pomocí výjimky typu int**

```
try
{
    ...
    if (x == 0)
        throw 1;
    ...
}
catch (const int &e)
{
    cout << "Chyba c. " << e << " : ";
    switch(e)
    {
        case 1:
            cout << "nepovolena hodnota x" << endl;
            break;
        ...
    }
}
```

# Výjimky - jemný úvod

**Příklad: ošetření chyby pomocí výjimky typu string**

```
try
{
    ...
    if (y == 0)
        throw (string) "nepovolena hodnota y";
    ...
}
catch (const string &e)
{
    cout << "Chyba: " << e << end;
}
```

# Výjimky - jemný úvod

**Příklad: ošetření chyby pomocí výjimky typu exception**

```
try
{
    ...
    if (y == 0)
        throw exception("nepovolena hodnota y");
    ...
}
catch (const string &e)
{
    cout << "Chyba: " << e.what() << end;
}
```

# Příklad: dynamicky se zvětšující pole

**Zadání: chytré pole**

- cílem je implementovat pole, které se bude dynamicky zvětšovat podle potřeby (ve chvíli, kdy uživatel bude chtít přidat další prvek do již plného pole).

# Příklad: dynamicky se zvětšující pole

## Nástřel

- cílem je implementovat pole, které se bude dynamicky zvětšovat podle potřeby (ve chvíli, kdy uživatel bude chtít přidat další prvek do již plného pole).

```
int n = 1;                  // aktualni delka pole
float *a= new float[n];
a[0] = 3.4;

/* n predame jako referenci (popr. ukazatel)
   abychom ho mohli ve funkci zmenit (zvysit o 1)
*/
a = pridej(a,n,4.5);
a = pridej(a,n,6.7);
...
delete[] a;
```

# Příklad: dynamicky se zvětšující pole

## Nástřel

```
/* Funkce prida prvek na konec jiz plneho dynamickeho pole
* a ... puvodni pole (funkce ho dealokuje)
* n ... delka pole (bude zmenena)
* x ... vkladana hodnota
* funkce vrati nove vytvorene zvetsene pole
*/
float* pridej(float *a, int &n, float x)
{
  // 1. vytvor nove pole delky (n+1) (operator new)
  // 2. prekopiruj do nej vsechny prvky pole a
  // 3. na index n vloz x
  // 4. aktualizuj n
  // 5. dealokuj puvodni pole (operator delete)
  // 6. vrat nove pole
}
```

# Příklad: dynamicky se zvětšující pole

**Nástřel ... alternativně:**

```
/* Funkce prida prvek na konec jiz plneho dynamickeho pole
* a ... reference na ukazatel na pole
*       (ukazatel bude zmenen)
* n ... delka pole (bude zmenena)
* x ... vkladana hodnota
*/
void pridej1(float *&a, int &n, float x)
{
  // 1. vytvor nove pole delky (n+1) (operator new)
  // 2. prekopiruj do nej vsechny prvky pole a
  // 3. na index n vloz x
  // 4. aktualizuj n
  // 5. dealokuj puvodni pole (operator delete)
  // 6. a bude ukazovat na nove pole
}
```

# Příklad: dynamicky se zvětšující pole

**Lépe:**

- pro pole si pamatuji jeho aktuální délku a kapacitu
- při naplnění kapacity pole nezvětším o 1, ale např.
  - o k prvků, kde k je vhodně zvolená konstanta
  - k-krát (typicky dvakrát) ... obecně efektivnější

```
int n = 0; // aktualni delka pole (pocet ulozenych prvku)
int k = 1; // aktualni kapacita pole
float *a= new float[k];
a = pridej(a,n,k,4.5); // n a k predame jako referenci,
                                                // (popr. ukazatel)
a = pridej(a,n,k,6.7);
...
delete[] a;
```

# Příklad: dynamicky se zvětšující pole... lépe:

```
/* Funkce prida prvek na konec jiz plneho dynamickeho pole
* a ... reference na ukazatel na pole
*       (ukazatel bude zmenen)
* n ... pocet prvku pole (bude zmenen)
* k ... kapacita pole (bude zmenena)
* x ... vkladana hodnota
*/
float* pridej(float *&a, int &n, int &k, float x)
{
  // 1. pokud je pole plne (k == n)
  // 1.1. vytvor nove pole delky (2*k) (operator new)
  // 1.2. prekopiruj do nej vsechny prvky pole a
  // 1.3. dealokuj puvodni pole (operator delete)
  // 1.4  a bude ukazovat na nove pole
  // 1.5. aktualizuj promennou k
  // 2. do pole a na index n vloz x
  // 3. aktualizuj promennou n
  // 4. vrat vysledne pole a
}
```

# Příklad: dynamicky se zvětšující pole

**Ještě lépe a přehledněji:**

- pole, jeho aktuální délku a kapacitu obalíme do struktury

```
struct ChytrePole
{
    float *a = nullptr;
    int n = 0;
    int k = 0;
};
int main()
{
    ChytrePole s;

    vytvor(s); // s predame jako referenci
    ...                // (popr. ukazatel)
    pridej(s,4.5);
    pridej(s,6.7);
    ...
    zrus(s);
}
```

# Příklad: dynamicky se zvětšující pole

**Ještě lépe a přehledněji:**

```
/* Funkce prida prvek na konec jiz plneho dynamickeho pole
* a ... reference na strukturu s polem
* x ... vkladana hodnota
*/
void pridej(ChytrePole &s, float x)
{
  // 1. pokud je pole plne (s.k == s.n)
  // 1.1. vytvor nove pole delky (2*s.k) (operator new)
  // 1.2. prekopiruj do nej vsechny prvky pole s.a;
  // 1.3. dealokuj puvodni pole s.a (operator delete)
  // 1.4. s.a = nove pole
  // 1.5. aktualizuj s.k
  // 2. do pole s.a na index n vloz x;
  // 3. aktualizuj promennou s.n;
}
```

# Příklad: dynamicky se zvětšující pole

**Ještě lépe a přehledněji:**

```
void zrus(ChytrePole &s)
{
  // 1. dealokuj pole s.a (operator delete)
  // 2. nastav vsechny parametry struktury na 0
 }

void vytvor(ChytrePole &s)
{
  // 1. pokud pole neni prazdne, tak ho vyprazdni (zrus)
  // 2. do s.a vloz nove pole delky 1
  // 3. nastav s.k = 1, s.n = 0
 }
```

# Příklad: dynamicky se zvětšující pole

**Další oprace nad polem (na procvičení):**

```
// smaz prvek na indexu i
void smaz(ChytrePole &s, int i)
{
  // 1. pokud je i platny index (0 <= i < s.n)
  // 1.1. vsechny prvky za indexem i prekopiruj
  //      o jednu pozici zpet
  // 1.2. aktualizuj s.n (kapacita se nemeni)
}

// najdi index prvku (vrat -1, pokud ho nenajdes)
int najdi(ChytrePole &s, float x)
{
  ...
}
```

# Příklad: dynamicky se zvětšující pole

**Další oprace nad polem (na procvičení):**

```
// pokud je prvek v poli, tak jeho prvni vyskyt smaz
bool smaz(ChytrePole &s, float x)
{
   ... (zavola predchozi dve funkce)
}

 // pokud je prvek v poli, tak vsechny jeho vyskyty smaz
bool smazVse(ChytrePole &s, float x)
{
   ... (napr. s vyuzitim predchozich funkci,
        nebo efektivneji)
}
```

# Příklad: dynamicky se zvětšující pole

**Povídací prográmek nad dynamickým polem:**

```
Pole: 4.5 6.7
 pro pridani prvku na konec zadej znak 'a'
 pro smazani prvniho vyskytu prvku zadej znak 'c'
 pro smazani vsech vyskytu prvku zadej znak 'd'
 pro zmenu prvku na indexu zadej znak 'z'
 pro ukonceni prace zadej jiny znak (napr. 'k')
Zadej operaci, kterou chces provest: a
Zadej cislo, ktere chces vlozit na konec pole: 7.8

Pole: 4.5 6.7 7.8
 pro pridani prvku na konec zadej znak 'a'
 pro smazani prvniho vyskytu prvku zadej znak 'c'
 pro smazani vsech vyskytu prvku zadej znak 'd'
 pro zmenu prvku na indexu zadej znak 'z'
 pro ukonceni prace zadej jiny znak (napr. 'k')
Zadej operaci, kterou chces provest: k
```

# Příklad: dynamicky alokovana matice

**Zadání: dynamicky alokovana matice**

- cílem je implementovat matici s proměnným počtem řádků i sloupců
- cvičení: implementujte nad maticemi operace sčítání a násobení

# Příklad: dynamicky alokovana matice

## Alokace dynamicke matice

```
int n = 5; // pocet radku
int m = 6; // pocet sloupcu
float **a; // matice (ukazatel na pole radku)

// alokace pole ukazatelu na radky
a = new float*[n];
for (int i = 0; i < n; i++)
{
    // alokace jednotlivych radku
    a[i] = new float[m];
}

// prace s matici
...

// dealokace
...
```

# Příklad: dynamicky alokovana matice

## Práce s dynamicky alokovanou maticí

```
int n = 5; // pocet radku
int m = 6; // pocet sloupcu
float **a; // matice (ukazatel na pole radku)

// alokace
...

// prace s matici
for (int i = 0; i < n; i++)
    for (int j = 0; j < m; j++)
        a[i][j] = 0; // vyplneni matice nulami
a[0][3] = 1;

// dealokace
...
```

# Příklad: dynamicky alokovana matice

## Dealokace dynamicke matice

```
int n = 5; // pocet radku
int m = 6; // pocet sloupcu
float **a; // matice (ukazatel na pole radku)

// alokace a prace s matici
...

// dealokace:
// 1. dealokace jednotlivych radku
for (int i = 0; i < n; i++)
{
    if(a[i] != nullptr);
        delete [] a[i];
}

// 2. dealokace pole ukazatelu na radky
delete[] a;
```

# Příklad: dynamicky alokovana matice

```
/* Funkce vytvori, inicializuje na 0 a vrati matici
 * o n radcich a m sloupcich */
float **vytvor(int n, int m)
{
    float **a;
    a = new float*[n];
    for (int i = 0; i < n; i++)
    {
        a[i] = new float[m];
    }
    for (int i = 0; i < n; i++) // inicializace
        for (int j = 0; j < m; j++)
            a[i][j] = 0;
    return a;
}
...
double **x = vytvor(n,m);
```

# Příklad: dynamicky alokovana matice

```
/* Funkce dealokuje matici o n radcich */
void zrus(float **a, int n)
{
        if (a == nullptr)
                return;
        for (int i = 0; i < n; i++)
        {
                if(a[i] != nullptr);
                        delete [] a[i];
        }
        delete[] a;
}
...
double **x = vytvor(n,m); // alokace
...
zrus(x,n); //dealokace
```

# Příklad: dynamicky alokovana matice

**Další oprace s maticemi (na procvičení):**

```
// secti matice a a b tvaru nxm, vrat vysledek
float** secti(float **a, float **b, int n, int m);

// vynasob matici a tvaru nxm a matici b tvaru mxo, vrat vysl
float** vynasob(float **a, float **b, int n, int m, int o);

// vypis matici a tvaru nxm na konzoli
void vypis(float **a, int n, int m);

// transponuj matici a tvaru nxm, vrat vysledek (mxn)
float** transponuj(float **a, int n, int m);
```

# Příklad: dynamicky alokovana matice

**Ještě lépe a přehledněji:**

- matici a její rozměry obalíme do struktury:

```
struct Matice
{
    int n = 0; // pocet radku
    int m = 0; // pocet sloupcu
    float **a = nullptr; // matice (ukazatel na pole radku)
}

int main()
{
    int n, m;
    ...
    Matice mat;
    vytvor(mat,n,m)
    ...
    vypis(mat);
    ...
    zrus(mat);
```