Základy programování v C++ 15. cvičení Dynamické proměnné

Zuzana Petříčková

19. listopadu 2019

Přehled

1. Ukazatele v C/C++ – pokračování
   - Dynamická alokace paměti
2. Výjimky - jemný úvod
3. Příklad: dynamicky alokovana matice

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 (ukazatelová aritmetika)
- **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. **statické**
   - v těle funkce: globální proměnné, které lze ovšem používat jen v dané funkci

```
void vypis(SvetovaStrana x)
{
    static const string nazvy[4]={"sever","jih","vychod","
    cout<<nazvy[x]<<endl;
}
```

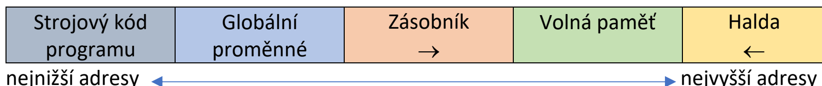
Proměnné v C/C++

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

Vytvoření a zrušení dynamické struktury

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

```
...
Zlomek *uz;
uz = new Zlomek;

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

delete uz;
uz = nullptr;
...
```

Uvolňování dynamicky alokované paměti

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

- pokud neuvolníme dynamickou proměnnou → napořád nám zabírat paměť
- pokud uvolníme již uvolněnou dynamickou proměnnou → nastane běhová chyba (někdy obtížně odhalitelná)

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 << "Chyba: Alokace se nezdarila.";
        return -1;
    }
     ... // veskera prace s promennou
    if (ui)
        delete [] ui;
    ...
}
```

Odbočka: jemný úvod do výjimek

Výjimky (exceptions)

- ošetření chyb za běhu programu
- 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 vyjimek
{
   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 (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 alokovana matice

Zadání: dynamicky alokovana matice

- cílem je implementovat matici s proměnným počtem řádků i sloupců
- cvičení: implementujte nad maticemi operace sčítání a transpozice (příp. 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

```
// 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

```
// alokace
...

// 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
x = nullptr;
```

Příklad: dynamicky alokovana matice

Další oprace s maticemi (na procvičení):

```
// 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);

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

// pokud byste s nudili:
// vynasob matici a tvaru nxm a matici b tvaru mxo,
// vrat vysledek (nxo)
float** vynasob(float **a, float **b, int n, int m, int o);
```

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);
```

Příklad: dynamicky alokovana matice

Oprace s maticemi (pro strukturu):

```
void zrus(Matice &mat);
void vytvor(Matice &mat, int n, int m);

// vypis matici a tvaru nxm na konzoli
void vypis(Matice &a);

// transponuj matici a tvaru nxm, b bude vysledek (mxn)
void transponuj(Matice &a, Matice &b);

// secti matice a a b tvaru nxm, c bude vysledek
void secti(Matice &a, Matice &b, Matice &c);

// pokud byste s nudili:
// vynasob matici a tvaru nxm a matici b tvaru mxo,
// c je vysledek (nxo)
void vynasob(Matice &a, Matice &b, Matice &c);
```