PAMĚŤOVÁ A ČASOVÁ SLOŽITOST ALGORITMU, DEKLARACE PROMĚNNÉ A DATOVÉ TYPY
CVIČENÍ Č. 1

Osnova

Opakování

- Datové typy, proměnná a deklarace proměnné
- Dynamická proměnná a ukazatel
- Dynamické alokace proměnné a její praktické využití, práce s dynamickou pamětí
- Statická alokace proměnné, paměť typu zásobník
- Principy OOP

Algoritmizace, algoritmus, vlastnosti algoritmu

Složitost algoritmu, druhy složitosti, odhad složitosti

Použité zdroje, doporučená studijní literatura

Hudec, B.: Přednáškové poznámky a příklady (https://moodle.vspj.cz/mod/resource/view.php?id=107676)

Hudec, B.: Algoritmy v C++, elektronický studijní text pro kombinovanou formu studia, Vysoká škola polytechnická Jihlava, 2011.

Přednáškové materiály

Ing. Marek Musil, marek.musil@vspj.cz

Proměnná, datový typ proměnné

Proměnná a datový typ, vytvoření proměnné (deklarace)

- Definice proměnné: **Proměnná** je místo v paměti, které slouží k uchování údaje určitého datového typu (celé číslo, číslo s desetinnou částí, znak, řetězec, …). Proměnná má datový typ, zabírá určitou velikost paměti, má pojmenování a je v paměti na určité adrese. Velikost alokovaného místa proměnné v paměti je dána datovým typem.

„deklarace proměnné“ = vytvoření proměnné příkazem.

„hodnota proměnné“ = hodnota zapsaná v proměnné.

„adresa proměnné“ = adresa paměti, na které leží proměnné.

```
int a, b;
int *p;

a = 5;
b = 3 + 2 * a;
p = &a;
*p = 4;
```

- Pojmenování proměnné je unikátní – jednoznačně určuje konkrétní proměnnou (v jednom kontextu nemůže existovat více proměnných stejného názvu).
- **Datový typ** určuje typ údaje a je charakteristikou typového systému programovacího jazyka. U číselných datových typů určuje datový typ rozsah čísla. Číselný datový typ může být dále znaménkové (signed) nebo (neznaménkový). Datové typy se obvykle rozdělují na jednoduché datové typy (primitivní) a složité datové typy.
- Datový typ udává typ údaje a určuje velikost proměnné v paměti. Velikost paměti pak určuje rozsah datového typu. Velikost paměti určené datovým typem je různých v programovacích jazycích a dále je závislý i na platformě (Unit, Windows, 32b x 64b, …)
  - V C a C++: char (2B, dle kódování), short (2B), int (4B), long (8B), float (4B), double (8B), viz *DatovéTypy.zip/DatoveTypy.cpp*
  - https://www.geeksforgeeks.org/c-data-types/
- Rozsah datového typu neznaménkového a znaménkového.

Deklarace a alokace proměnné

Proměnná a datový typ, vytvoření proměnné (deklarace)

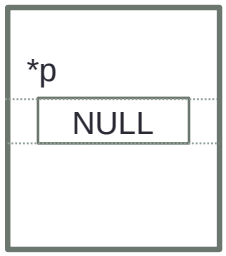
- Proměnná vzniká deklarací. Deklarace je příkaz, který vytváří proměnnou a udává datový typ a název proměnné. Deklarace s inicializací – dtto deklarace, ale navíc je proměnná inicializována. Deklarací proměnné je v paměti zabráno místo o velikosti odpovídajícímu datovému typu a paměť proměnné je přístupná programu, ve kterém byla proměnná deklarována. Přístup k paměti je pak záležitostí operačního systému: kontrola oprávnění, neoprávněný přístup do paměti.
- Zánik proměnné je na konci programového bloku, ve kterém byla proměnná vytvořena (deklarována). Programovací jazyk může umožnit deklarovat proměnnou pouze na začátku bloku nebo i uprostřed bloku. Jazyk C, C++ a Java umožní deklarovat proměnnou kdekoliv.
- Dle způsobu deklarace rozlišujeme
  - Statická deklarace proměnné – u proměnné není možné zajistit explicitní zrušení, zaniká na konci programového bloku, ve kterém byla deklarována.
  - Dynamická deklarace proměnné (deklarace dynamickou alokací) – proměnná vzniká voláním speciálního příkazu (v jazyce C++ příkaz *new*, v jazyce C funkce *malloc*), zaniká voláním příkazu (v jazyce C++ příkaz *delete*, v jazyce C funkce *free*).
- Dle místa a platnosti deklarace rozlišujeme
  - Lokální proměnnou – proměnná deklarována uvnitř programového bloku. Proměnné zaniká na konci daného programového bloku.
  - Globální proměnná – proměnná deklarována mimo programových blok. Proměnná je platná ve všech částech programu. Zaniká na konci programu. Může existovat lokální proměnná pojmenována stejně jako globální proměnná. Při vykonání bloku je globální proměnná zastíněna lokální proměnnou.

Dynamická proměnná, dynamická alokace proměnné

- Přístup k proměnné na základě adresy paměti. Uložena v proměnné typu ukazatel.
- Proměnná typu ukazatel obsahuje jako hodnotu adresu místa paměti a umožňuje přístup k dynamické paměti. Staticky deklarovaná proměnné je umístěna v paměti typu zásobník (stack), dynamicky alokovaná proměnné je umístěna v paměti typu halda.
- Proměnné typu ukazatel ukazuje na proměnné stejného datového typu jako je typ uvedený u ukazatele. Ukazatel všech typů zabírají v paměti stejnou velikost, která je závislá na platformě.

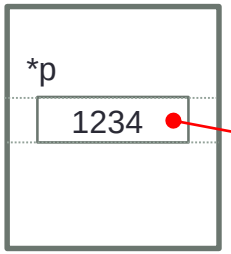
```
int *p = NULL;
```

stack (zásobník)

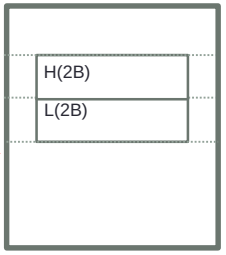



```
int *p = NULL;
p = new int();
*p = 65537;
delete *p;
```

stack (zásobník)

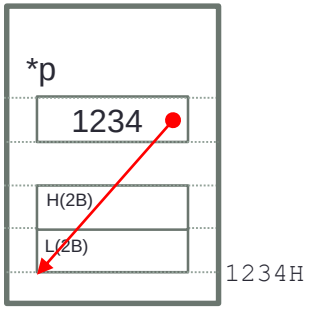


halda, dynamická paměť




```
int *p = NULL;
int c = 10;
p = &c;
*p = 65537;
```

stack (zásobník)



Demonstrační (vysvětlující) příklad:

DeklaraceAlokace-1.cpp a *DeklaraceAlokace-2.cpp*

```
int a;  // globalní proměnná

void f1()
{
    int a, b;
    // lokální proměnná, proměnná 'a' zastíní globální proměnnou 'a'
    a = 5;
    b = a * 2;
    printf("f1-1:%d;%d;", a, b);
}

void f2()
{
    int b, c;       // lokální proměnná
    a += 10;
    int d; // deklarace uvnitř bloku, pokud programovací jazyk podporuje
    b = a * 2;
    printf("f2-1:%d;", a);
}

int main()
{
    int b, c;   // lokální proměnná
    int d = 5;  // deklarace lokální proměnné s inicializací
    b = 4;
    c = 2;
    a = 1;
    printf("main-1:%d;", a);
    f1();
    f2();
    printf("main-2:%d;", a);
    f2();
}
```

```
// paměťová složitost vykonání programu:
// 4B + 3x4B + 2x4B + 2x(3x4B) = 48B

// Program vytiskne:
// main-1:1;f1-1:5;10;f2-1:11;main-2:11;f2-1:21;
```

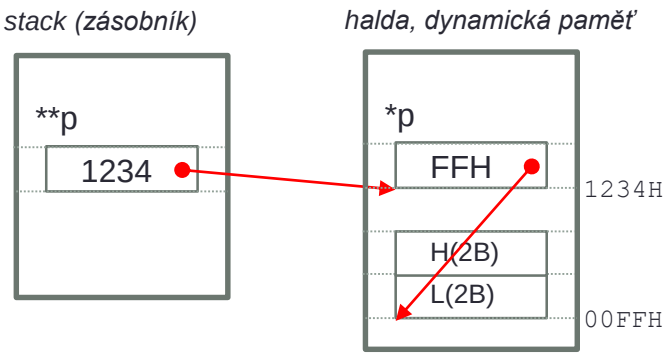
```
void f3()
{
    int a;
    // ...
    // ...
    int* p2;
    // ...
    p2 = new int();
    // ...
    // ...
    delete p2;
    // ...
    // ..
}

int main()
{
    int* p1;
    // ...
    p1 = new int();
    // ...
    f3();
    f3();
    // ...
    delete p1;
    // ...
}
```

```
// paměťová složitost vykonání programu:
// (8B + 4B) + 2x[4B + (8B + 4B)] = 48B
```

Alokace proměnné – shrnutí

- **Statická a dynamická alokace**
- Staticky alokovaná proměnná vzniká deklarací a je umístěna v paměti typu zásobník. Dynamicky alokovaná proměnná vzniká provedením příkazu alokace (v C funkce malloc, v C++ příkaz new) a je umístěna v paměti typu halda. Zatímco staticky alokovaná proměnná zaniká na konci programového bloku, ve kterém byla deklarována, dynamicky alokovaná proměnná zaniká až voláním příkazu dealokace (v C funkce free, v C++ příkaz delete).
- Paměť staticky alokované proměnné je programem „blokována“, i když už daná proměnná není v rámci bloku využívána.
- **Definice ukazatele**
- Ukazatel je datový typ, která slouží k uchování adresy v paměti počítače. Je to proměnná, která uchovává adresu paměti. Říkáme, že ukazatel ukazuje na místo paměti dané adresou.
  - https://cs.wikipedia.org/wiki/Ukazatel_(programov%C3%A1n%C3%AD)
  - https://www.klikzone.cz/cplusplus/ukazatele1.php
- Rozlišujeme pojem „hodnota ukazatele“ a „hodnota adresy ukazatele“. Pokud ukazatel neodkazuje na žádné místo paměti, je hodnotou ukazatele NULL. Proměnné typu ukazatel mají v paměti stejnou velikosti, bez ohledu na datový typ, na která ukazují. Velikost proměnné typu ukazatel je dána adresací operačního systému (16b -> 2B, 32b -> 4B, …).
- **Terminologie:**
  - Ukazatel - `int *p;`
  - Hodnota ukazatele - `*p`
  - Hodnota adresy ukazatele - `p`
  - Adresa ukazatele - `&p`
- Ukazatel se obvykle používá k odkazu na proměnnou v dynamické paměti (haldě), ale může odkazovat i na staticky alokovanou proměnnou (zásobník).
- Pokud ztratíme odkaz na dynamicky alokovanou proměnnou, je tato proměnná nedohledatelná. Neznáme její adresu v paměti.
- Ukazatel je v programování velmi efektivní a užitečný nástroj. V některých jazycích jsou pomocí ukazatele realizovány vstupně-výstupní a výstupní parametry funkce (metody).
- Proměnná typu ukazatel je alokována na zásobníku, ale může být alokována na haldě – „ukazatel na ukazatel“.

```
int **p = NULL;
*p = new *int();
p = new int();
**p = 65537;
delete *p; *p = NULL;
delete p; p = NULL;
```



Datový typ pole, alokace pole

Datová struktura pole

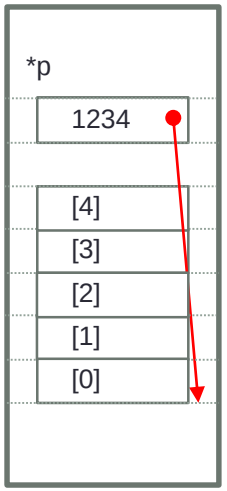
- Složitý datový typ. Více hodnot (položek) stejného datového typu. V paměti jsou prvky umístěny vedle sebe.
- Terminologie: pole, prvek pole, index prvku pole, velikost pole, datový typ

Alokace pole (Statická a dynamická alokace)

- Jazyk C a C++ podporují statickou i dynamickou alokaci pole, ale ostatní jazyky (např. Java, C#) nyní podporují přímo pouze dynamickou alokaci pole. Práce s polem je pak stejná bez ohledu na způsob vytvoření (typ alokace). V uvedených jazycích C a C++ nesmíme zapomenout na dealokaci.
- Pole je sekvence proměnných stejného datového typu, v paměti uloženy vedle sebe (pole pokrývá v paměti kontinuální blok). Pole chápeme jako ukazatel odkazující na začátek pole (na 0. prvek).
- **Alokace pole hodnot**

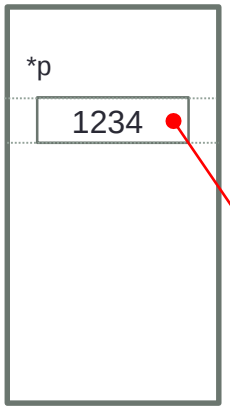
```
// Práce s polem, pointerová aritmetika
p[0] = 10;      *(p + 0) = 10;
p[4] = 8;       *(p + 4) = 8;
```

stack (zásobník)

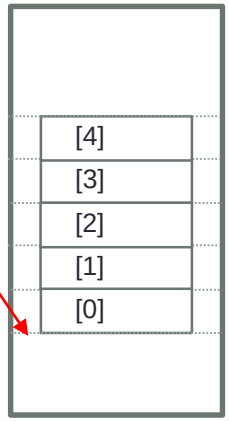



```
int p[5];

// Práce s polem
```

stack (zásobník)



halda, dynamická paměť




```
int *p = new int[5];

// Práce s polem

delete[] p;
```

- **Alokace pole ukazatelů na hodnoty**

```
int *p[5];

// Práce s polem
```

```
int **p = new int*[5];

// Práce s polem

delete[] p;
```

```
// Práce s polem, pointerová aritmetika
p[0] = new int(10);     p[0] = new int(); *p[0] = 10;      *(p + 0) = new int(10);
p[4] = new int(8);      p[4] = new int(); *p[4] = 8;       *(p + 4) = new int(8);
*p[4] = 7;              **(p + 4) = 7;
```

Alokace pole – demonstrační příklady (1)

```
/// <summary>
/// Ukázka alokace pole hodnot.
/// </summary>
void alokace_pole_1()
{
	// pole hodnot
	int pole1_1[5];

	pole1_1[0] = 5;  // *pole1_1 = 5;  *(pole1_1 + 0) = 5
	pole1_1[4] = 0;  // *(pole1_1 + 4) = 0;

	// ...

	int* pole1_2 = new int[5];

	pole1_2[0] = 5;  // *pole1_1 = 5;  *(pole1_1 + 0) = 5
	pole1_2[4] = 0;  // *(pole1_1 + 4) = 0;
	delete[] pole1_2;
}
```

Alokace pole – demonstrační příklady (2)

```
/// <summary>
/// Ukázka alokace pole ukazatelů na hodnotu.
/// </summary>
void alokace_pole_2()
{
	// pole ukazatelů na hodnotu
	int* pole2_1[5];

	pole2_1[0] = new int();
	*pole2_1[0] = 5;   // **(pole2_1 + 0) = 5;
	*(pole2_1 + 4) = 0;// **(pole2_1 + 4) = 0;

	for (int i = 0; i < 5; i++)
	{
		pole2_1[i] = new int(/*5 - i*/);
		*pole2_1[i] = 5 - i;  // **(pole2_1 + i) = 5 - i;
	}

	for (int i = 0; i < 5; i++)
		cout << &pole2_1[i] << ':' << pole2_1[i]
			<< ':' << *pole2_1[i] << endl;

	// --

	int** pole2_2;
	pole2_2 = new int* [5];

	pole2_2[0] = new int();
	*pole2_2[0] = 5;   // **(pole2_2 + 0) = 5;
	*(pole2_2 + 4) = 0;// **(pole2_2 + 4) = 0;

	for (int i = 0; i < 5; i++)
	{
		pole2_2[i] = new int(/*5 - i*/);
		*pole2_2[i] = 5 - i;  // **(pole2_2 + i) = 5 - i;
	}

	for (int i = 0; i < 5; i++)
		cout << &pole2_2[i] << ':' << pole2_2[i] << ':'
		   << *pole2_2[i] << endl;


	// ...

	for (int i = 0; i < 5; i++)
	{
		delete pole2_1[i];// .....
		pole2_1[i] = NULL;// .....
	}

	for (int i = 0; i < 5; i++)
	{
		delete pole2_2[i];  // .....
		pole2_2[i] = NULL;  // .....
	}
	delete[] pole2_2;
}
```

Alokace pole – demonstrační příklady (3)

```
/// <summary>
/// Ukázka alokace 2d-pole hodnot.
/// </summary>
void alokace_pole_3()
{
	// 2d-pole hodnot
	// ----
	int pole3_1[5][10];

	pole3_1[0][0] = 10;
	pole3_1[1][2] = 5;  // *(pole3_1[1] + 2) = 5;

	for (int i = 0; i < 5; i++)
	for (int j = 0; j < 10; j++)
		pole3_1[i][j] = i * j;  // *(pole3_1[i] + j) = i * j;

	// ----

	int* pole3_2[5];
	for (int i = 0; i < 5; i++)
		pole3_2[i] = new int[10];

	// dtto

	for (int i = 0; i < 5; i++)
		delete[] pole3_2[i];

	// ----

	int** pole3_3;
	pole3_3 = new int*[5];
	for (int i = 0; i < 5; i++)
		pole3_3[i] = new int[10];

	// dtto

	for (int i = 0; i < 5; i++)
		delete[] pole3_3[i];

	delete[] pole3_3; // todo
}
```

Alokace pole – demonstrační příklady (4)

```
/// <summary>
/// Ukázka alokace 2d-pole ukazatelů na hodnotu.
/// </summary>
void alokace_pole_4()
{
	// 2d-pole ukazatelů na hodnotu
	// ----
	int *pole3_1[5][10];

	pole3_1[0][0] = new int();
	*pole3_1[0][0] = 10;
	pole3_1[1][2] = new int();
	*pole3_1[1][2] = 5;  // **(pole3_1[1] + 2) = 5;

	for (int i = 0; i < 5; i++)
		for (int j = 0; j < 10; j++)
		{
			pole3_1[i][j] = new int();
			*pole3_1[i][j] = i * j;
			// **(pole3_1[i] + j) = i * j;
		}

	// ----

	int** pole3_2[5];
	for (int i = 0; i < 5; i++)
	pole3_2[i] = new int*[10];

	// dtto

	for (int i = 0; i < 5; i++)
	{
	for (int j = 0; j < 10; j++)
	delete pole3_2[i][j];
	delete[] pole3_2[i];
	}

	// ----
	... ...
```

```
	... ...


	int*** pole3_3;
	pole3_3 = new int**[5];
	for (int i = 0; i < 5; i++)
	pole3_3[i] = new int*[10];

	// dtto

	for (int i = 0; i < 5; i++)
	{
		for (int j = 0; j < 10; j++)
		delete pole3_3[i][j];
		delete[] pole3_3[i];
	}

	delete[] pole3_3; // todo
}
```

- Konec opakování.
- Na následujících slajdech je látka cvičení.

Složitost algoritmu

Algoritmizace a algoritmus – úvod do problému

- Algoritmus:
  - Návod na provedení činnosti, posloupnost kroků vedoucí k vyřešení problému (tzv. výpočetní problém). Má vstupní data a poskytuje výstupní data. Provedení algoritmu znamená transformace vstupních hodnot na výstupní hodnoty.
  - Vlastnosti algoritmu: deterministický, konečný, rezultativní, správný, obecný (hromadný)
- Algoritmizace: proces vytvoření algoritmu.
- Výpočet problém má minimálně jeden algoritmus. Složitost algoritmu je nástroj na porovnání algoritmů.

Složitost algoritmu

- Časová složitost: jak závisí „doba“ vykonání programu (algoritmu) na velikosti vstupních dat a na počtu vstupních dat.
- Paměťová složitost: jak závisí využití paměti na velikosti vstupních dat a na počtu vstupních dat.

Časová složitost

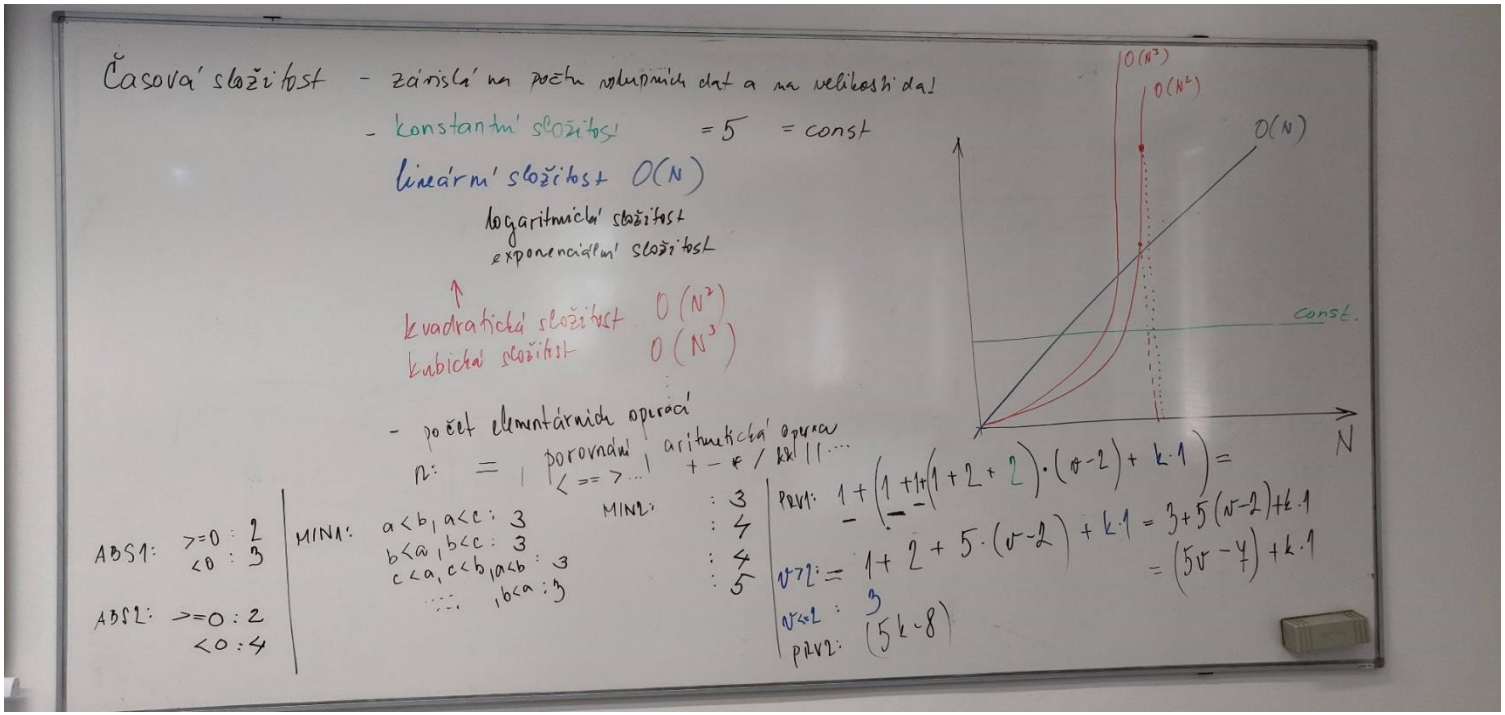
- Způsob vyhodnocení časové složitosti – metrika vyhodnocení
  1. Vyjádřeno jako počet provedených elementárních operací
     - Přiřazení: =
     - Porovnání: ==, <, >, >=, <=, != , …
     - Aritmetická operace: > +, -, *, /, &&, ||, !, …
  2. Vyjádřeno jako počet provedených elementárních operací na daty
  3. Vyjádřeno jako počet porovnání.

- Vyjádření složitosti ve všech případech (nekonečně mnoho): krajní případy, nejlepší případ, nejhorší případ, obecný (průměrný) případ. Uvažovat lze složitost pouze očekávané pravděpodobnosti výskytu případu.
- Odhad složitosti O, asymptotická složitost (nejhorší případ)
- Složitost:
  - Konstantní K, Lineární $O(N)$, Logaritmická $O(\log N)$, $O(N*\log N)$ a exponenciální
  - Kvadratická $O(N^2)$, Kubická $O(N^3)$, …

- Závislost na velikost vstupních dat a počtu vstupních dat.
- Slouží k porovnání algoritmů. Lze rozhodnout, který algoritmus je lepší, případně za jakých podmínek. Zohlednit frekvenci výskytu případu.

Časová složitost – typy složitostí, případy

- Definice složitost, druhy složitosti, vyjádření složitosti.
- Vyhodnocení složitosti
  - Větvení, rozhodování
  - Konstrukce cyklu
- Porovnání složitostí: porovnání identifikovaných případů
  - Optimální varianta: beze změny složitosti nebo zhoršení složitosti.
  - Pokud se uvažuje častý výskyt jistého testovacího případu, lze podle toho zvolit návrh algoritmu.



Řešení cvičení (1)

- U1: Algoritmus pro určení opačného čísla.
  - Pro všechny hodnoty konstantní a stejná složitost. **1, const**
  - Volání metody má složitost **3**, potom je složitost 1 + [3] = 4.

```
float CislaOperace::cisloOpacne(float v)
{
     return v * -1.0F;
}
```

- U2: Algoritmus pro určení absolutní hodnoty čísla.

```
float CislaOperace::cisloAbsolutniHodnota_1(float v)
{
    float abs;

    if (v >= 0)
        abs = v;
    else
        abs = v * -1.0F;

    return abs;
}

float CislaOperace::cisloAbsolutniHodnota_2(float v)
{
    float abs = v;

    if (v < 0)
        abs *= -1.0;

    return abs;
}
```

Případ	Složitost verze 1	Složitost verze 2
Číslo je kladné nebo nulové.	2, const	2, const
Číslo je záporné.	3, const	4, const

Řešení cvičení (2)

- U3: Algoritmu pro určení minimální (maximální) hodnoty ze tří čísel.

```
float CislaOperace::urciMin_1(float a, float b, float c)
{
    float m;
    if (a < b)
        if (a < c)
            m = a;
        else
            m = c;
    else
        if (b < c)
            m = b;
        else
            m = c;
    return m;
}
```

Případ	Složitost verze 1	Složitost verze 2
Hodnota a je minimální min(a, b, c) = a	3, const	3, const
Hodnota b je minimální min(a, b, c) = b	3, const	4, const
Hodnota c je minimální min(a, b, c) = c, a < b	3, const	4, const
Hodnota c je minimální min(a, b, c) = c, a > b	3, const	5, const

```
float CislaOperace::urciMin_2(float a, float b, float c)
{
    float m = a;
    if (m > b)
        m = b;
    if (m > c)
        m = c;
    return m;
}
```

- Záměna pořadí testování proměnných

```
float CislaOperace::urciMin_3(float a, float b, float c)
{
    float m = a;
    if (m > c)
        m = c;
    if (m > b)
        m = b;
    return m;
}
```

Případ	Složitost verze 3
Hodnota a je minimální min(a, b, c) = a	3, const
Hodnota b je minimální min(a, b, c) = b, c > a	4, const
Hodnota b je minimální min(a, b, c) = b, c < a	5, const
Hodnota c je minimální min(a, b, c) = c	3, const

- Co když budou hodnoty předávány v jiném pořadí? Při implementaci algoritmu zohlednit očekávané pořadí hodnot. Např. budeme vědět, že třetí hodnota je nejpravděpodobněji nejmenší.

Řešení cvičení (3)

- U4: Algoritmu pro určení minimální a maximální hodnoty ze tří čísel.

```
void CislaOperace::urciMinMax_1(float a, float b, float c, float *min, float *max)
{
    if (min == NULL || max == NULL)
        return;
    // min
    if (a < b)
        if (a < c)
            *min = a;
        else
            *min = c;
    else
        if (b < c)
            *min = b;
        else
            *min = c;

    // max
    if (a > b)
        if (a > c)
            *max = a;
        else
            *max = c;
    else
        if (b < c)
            *max = b;
        else
            *max = c;
}

void CislaOperace::urciMinMax_2(float a, float b, float c, float *min, float *max)
{
    if (min == NULL || max == NULL)
        return;
    *min = urciMin_1(a, b, c);  // / urciMin_2(a, b, c);
    *max = urciMax_1(a, b, c);  // / urciMax_2(a, b, c)
}
```

Případ	Složitost verze 1	Složitost verze 2
Parametr ukazatele *min* má hodnotu NULL	1	1
Parametr ukazatele *max* má hodnotu NULL	3	3
Hodnota *a* je minimální *min(a, b, c) = a* Hodnota *b* je minimální *min(a, b, c) = b* Hodnota *c* je minimální *min(a, b, c) = c*	3 + 3 + 3 = 9, const	3 + (3 + *4*) + (3 + *4)* = 15, const

Řešení cvičení (4)

- U5: Zajištění načtení kladné hodnoty čísla opakovaným načítáním čísel.

```
int RetezceOperace::nactiCisloKladne()
{
    int c;
    do
    {
        printf("Zadej cele cislo: ");
        scanf("%d", c);
    } while (c < 0);
    return p;
}

int* RetezceOperace::nactiCisloKladneDyn()
{
    int *p = new int();
    do
    {
        printf("Zadej cele cislo: ");
        scanf("%d", p);
    } while (*p < 0);
    return p;
}
```

Případ	Složitost verze 1	Složitost verze 2
Načtení hodnoty s n pokusy	2*n+1, lineární	1 + (2*n + 1), lineární

- U6: Test prvočísla.
  - Složitost cyklu for : 1 + 1 + n * (složitost-těla-cyklu + 1 + 2), n je počet opakování těla cyklu (počet iterací)

```
bool CislaOperace::testPrvocislo_1(int v)
{
    bool jePrvocislo = true;
    for (int i = 2; i < v; i++)
        if (v % i == 0)
            jePrvocislo = false;
    return jePrvocislo;
}

bool CislaOperace::testPrvocislo_2(int v)
{
    for (int i = 2; i < v; i++)
        if (v % i == 0)
            return false;
    return true;
}
```

Případ	Složitost verze 1	Složitost verze 2
v <= 2	3 + 1, const	2 + 1, const
v > *2*, číslo *v* není prvočíslem	[1 + 2 + 5 * (v – 2) + 1 * m] + 1 = **5 * v – 7** + m + 1, **O(*2*v), lineární**	[2 + 5 * (k – 1) + 2] + 1 = 5 * k – 1, **O(k), lineární**
v > *2*, číslo *v* je prvočíslem	[5 * v – 7] + 1, **O(v), lineární**	[2 + 5 * (v – 2)] + 1 = [5 * v - 8] + 1, **O(v), lineární**

Řešení cvičení (5)

- Rekurze
  - Podstata rekurze, pravidla pro realizaci rekurze
  - Ukončovací podmínka = plyne z podstatu problému řešeného rekurzivním přístupem (rekurzivní metodou).

- U7: Výpočet faktoriálu čísla pomocí rekurze

```
int CislaOperace::faktorialR(int n)
{
    if (n < 0)
       return -1;
    if (n == 0)
       return 1;
    return n * faktorialR(n - 1);
}
```

Případ	Složitost verze 1
n < 0	[1] + 1, const
n = 0	[2] + 1, const
n > 0	[2 + 4 * n + 2] + 1 = [4 + 4 * n] + 1, **O(n), lineární**

  - Jak se změní složitost, pokud prohodíme první dvě konstrukce if?

- U8: Zjištění délky řetězce

```
int RetezceOperace::lengthStr_1(char *str)
{
    if (str == NULL)
        return 0;
    int d = 0;
    while (*str != '\0')
    {
        str++;
        d++;
    }
    return d;
}

int RetezceOperace::lengthStr_2(char *str)
{
    if (str == NULL)
        return 0;
    char *p = str;
    while (*p != '\0')
        p++;
    return p - str;
}
```

Případ	Složitost verze 1	Složitost verze 2
Délka řetězce N >= 0	2 + 1 + 5 * N = 5 * N + 3, lineární, N	2 + 1 + 3 * n = 3 * N + 3, lineární, N

Řešení cvičení (6)

- U9: Nalezení znaku (resp. pozice znaku) v textu

```
int RetezceOperace::indexOf_1(char *str, char z)
{
    if (str == NULL)
        return -1;

    int d = lengthStr_1(str);

    for (int i = 0; i < d; i++)
        if (str[i] == z)
            return i;
    return -1;
}

int RetezceOperace::indexOf_2(char *str, char z)
{
    if (str == NULL)
        return -1;

    char *p = str;
    while (*p != '\0')
    {
        if (*p == z)
            return (p - str) / sizeof(char);
        p++;
    }
    return -1;
}
```

Případ	Složitost verze 1	Složitost verze 2
Nulová délka řetězce (prázdný řetězec)	1	1
Znak na pozici k	1 + ((5N + 3) + 2) + (2 + 4K) = 5N + 4K + 8, lineární, O(2N)	2 + 1 + 4K + 3 = 4K + 6, lineární, O(N)

Řešení cvičení (7)

- U10: Kopírování řetězce

```
void RetezceOperace::copyStr_1(char *str1, char *str2)
{
    if (str1 == NULL || str2 == NULL)
        return;
    int d = lengthStr_1(str2);
    for (int i = 0; i < d; i++)// ;i <= d;
        str1[i] = str2[i];
    str1[d] = '\0';
}

void RetezceOperace::copyStr_2(char *str1, char *str2)
{
    if (str1 == NULL || str2 == NULL)
        return;
    char *p1 = str1, *p2 = str2;
    while (*p2 != '\0')
    {
        *p1 = *p2;
        p1++;
        p2++;
    }
    *p1 = '\0';
}

void RetezceOperace::copyStr_3(char *str1, char *str2)
{
    if (str1 == NULL || str2 == NULL)
        return;
    char *p1 = str1, *p2 = str2;
    do
    {
        *p1 = *p2;
        if (*p2 == '\0')
            break;
        p1++;
        p2++;
    } while (true);
    *p1 = '\0';
}
```

Testovací případy:

- Není odkaz na řetězec 1
- Není odkaz na řetězec 2
- Není odkaz na oba řetězce
- Kopírovaný řetězec má nulovou délku (prázdný řetězec)
- Kopírovaný řetězec má délku 1 (jeden znak)
- Kopírovaný řetězec má délku > 1

Případ	Složitost verze 1	Složitost verze 2	Složitost verze 3
Není odkaz na řetězec 1	1	1	1
Není odkaz na řetězec 2	3	3	3
Délka řetězce N, N >= 0	3 + ((5N + 3) + 2) + (2 + 4N) + 1 = 9N + 9 + 2, lineární, O(9N)	3 + 2 + 1 + 6N + 1 = 6N + 7, lineární, O(6N)	3 + (2 + 6N + 2) + 1 = 6N + 8, lineární, O(6N)

Řešení cvičení (8)

- U11: Porovnání řetězců

```
bool RetezceOperace::cmpStr_1(char *str1, char *str2)
{
    if (str1 == NULL || str2 == NULL)
        return false;
    int d1 = lengthStr_1(str1); // lengthStr_2(str1);
    int d2 = lengthStr_1(str2); // lengthStr_1(str2);
    if (d1 != d2)
        return false;
    bool stejne = true;
    for (int i = 0; i < d1; i++)
        if (str1[i] != str2[i])
        {
            stejne = false;
            break;//
        }
    return stejne;
}

bool RetezceOperace::cmpStr_2(char *str1, char *str2)
{
    if (str1 == NULL || str2 == NULL)
        return false;
    int d1 = lengthStr_1(str1); // lengthStr_2(str1);
    int d2 = lengthStr_1(str2); // lengthStr_1(str2);
    if (d1 != d2)
        return false;
    int i = 0;
    while (*str2 != '\0')
    {
        if (str1[i] != str2[i])
            return false;
        str1++;
        str2++;
    }
    return true;
}
```

Testovací případy:

- Není odkaz na řetězec 1
- Není odkaz na řetězec 2
- Není odkaz na oba řetězce
- Porovnávané řetězce mají různou délku
- Porovnávané řetězce mají stejnou délku, ale jsou různé
  - Mají délku 1
  - Mají délku > 1
- Porovnávané řetězce mají stejnou délku, ale jsou stejné
  - Mají délku 1
  - Mají délku > 1

Případ	Složitost verze 1	Složitost verze 2
Není odkaz na řetězec 1	1	1
Není odkaz na řetězec 2	3	3
Délka řetězce M a N, M <> N	3 + ((5M + 3) + *2*) + ((5N + 3) + *2*) + 1 = 5M + 5N + 14, lineární, O(M+N)	3 + ((5M + 3) + *2*) + ((5N + 3) + *2*) + 1 = 5M + 5N + 14, lineární, O(M+N)
Délka řetězce M a N, M = N, rozdíl znaků na pozici K	3 + ((5M + 3) + *2*) + ((5N + 3) + *2*) + 1 + (1 + 2 + 4K + 2) = 5M + 5N + 4K + 20 = 10N + 4K + 20 , lineární, O(2N+K)	3 + ((5M + 3) + *2*) + ((5N + 3) + *2*) + 1 + (2 + 6K + 2) = 10N + 6K + 18, lineární, O(2N+K)
Délka řetězce M a N, řetězce jsou stejné (M = N)	3 + ((5M + 3) + *2*) + ((5N + 3) + *2*) + 1 + (1 + 2 + 4N) = 5M + 5N + 4N + 17 = 14N + 17, lineární, O(3N)	3 + ((5M + 3) + *2*) + ((5N + 3) + *2*) + 1 + (2 + 6N) = 16N + 16, lineární, O(3N)

Řešení cvičení (9)

- Úkol pro cvičení:
  - U12: Doplňte metodu CislaOperace::swapInt(int &a, int &b) a metodu *CislaOperace::swapIntP(int *a, int *b)* pro záměnu hodnot proměnných. Jaká je složitost algoritmu? Přidejte deklaraci hlavičky a definici metody do projektu.
  - U13: Doplňte metodu RetezceOperace::*reverseStr(char *str)*, která provede reverzi řetězce (záměnu znaků v řetězci – první znak s posledním znakem, druhý znak s předposledním znakem, atp.) Jaká je odhad složitosti algoritmu? Jaká je složitost algoritmu?
  - U14: Doplňte metodu *RetezceOperace::intToStr(int c) : *char* pro převod hodnoty celého čísla na řetězec (text). Jaký je odhad složitosti algoritmu? Jaká je složitost algoritmu?