Vývojové diagramy – 1. díl

Autor: Jiří Chytil, korektura: Zdeněk Lehocký, 24. 07. 2005
http://programujte.com/clanek/2005080105-vyvojove-diagramy-1-dil/

Povíme si, k čemu vývojové diagramy slouží a jak se používají.

Vývojové diagramy znázorňují průběh či stavbu programu. Používají se jako část dokumentace projektu. Většina projektů začíná tvorbou takového vývojového diagramu. Většinou si ale nebudete vytvářet vývojový diagram, pokud si budete něco zkoušet. Pokud se však rozhodnete programovat nějakou aplikaci, která má být funkční, není nikdy na škodu vytvořit si vývojový diagram. Celkem dobře se v něm hledají chyby, můžete sledovat postup programu a případně vše poopravit. Vhodný program pro návrh či následnou úpravu vývojového diagramu je třeba Microsoft Visio, o kterém jsem psal nedávno, aneb jak se říká – pokrok je pokrok, ale papír je papír. Zde je ovšem problém při editaci. Vývojový diagram není nutností, ale určitě bych ho doporučil, obzvláště pokud pracujete jako skupina nebo pokud tvoříte rozsáhlejší projekt. Vývojový diagram se také hodí, pokud přecházíte z jiného jazyka a chcete předělat nějaký program na jiný jazyk, tehdy je vývojový diagram velkým pomocníkem.

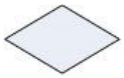
Vývojový diagram je grafické znázornění algoritmu.

Algoritmus je přesný postup, který vede k vyřešení určitého výsledku. Vysvětlili jsme si pojem algoritmus a přidáme ještě jeden pojem – **deterministický**. Znamená to, že pokud programu dáme určitá data, vrátí nám výsledek, a pokud mu tatáž data zadáme znovu, výsledek bude totožný s předchozím.

Vývojové diagramy se skládají z grafických značek. Značky jsou různé a různě se kombinují, tím se simulují různé situace a různé příkazy, do těchto značek se pak vypisují upřesňující údaje. Nyní se podíváme na to, jak vypadají jednotlivé části vývojového diagramu:


Konec a začátek algoritmu


Běžný příkaz


Podmíněný výraz


Cyklus s určeným počtem opakování


Cyklus s podmínkou na konci


Cyklus s podmínkou na začátku


Ruční vstup


Zobrazení výstupu


Zpracování souboru


Uložení dat do souboru


Podprogram


Spojovací značka

Spojovací čára

Kdybych začal od spojovací čáry, tak bych k ní podotkl, že pokud směřuje doprava nebo dolů, není k ní potřeba dělat šipku, pokud ovšem směřuje do leva či nahoru, šipka se dělat musí.

Příště se podíváme na to, jak se vývojové diagramy tvoří.

Vývojové diagramy – 2. díl

Autor: Jiří Chytil, korektura: Zdeněk Lehocký, 11. 10. 2005
http://programujte.com/clanek/1970010171-vyvojove-diagramy-2-dil/

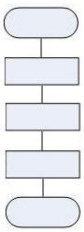
Při tvorbě vývojových diagramů se vychází z několika základních struktur a zvyklostí. My si je teď ukážeme a budeme se jich držet, protože se standardně používají, tak ať si nevymýšlíme zbytečně vlastní.

Sekvence

Sekvence je první z možností. Jsou to příkazy jdoucí po sobě bez oklik, skoků a větvení, jednoduše řada příkazů, které se jeden po druhém provedou. Například:

- zajdi do obchodu
- udělej snídani
- ukliď si pokoj

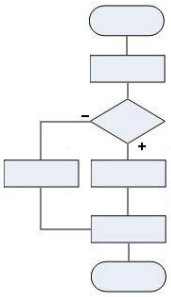
Pokud bychom se chtěli více přiblížit výpočetní technice, mám tu jiný příklad:



- načti data do proměnných
- sečti proměnné
- vytiskni výsledek
- ukonči program

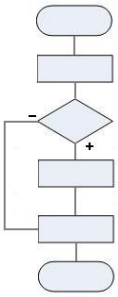
Větvení

Program se větví na několik částí. Která z těchto částí se vykoná, závisí často na podmínce. Pokud je tedy podmínka splněna, provede se něco jiného, než když podmínka splněna není.



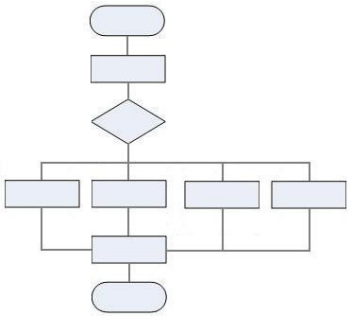
Úplná alternativa

Program se větví do dvou částí: pokud je podmínka splněna, provede se jedna část kódu, pokud ne, provede se druhá část. Příklad: Pokud je součet zadaných čísel vyšší nebo roven nule, vytiskni, že je výsledek kladný, pokud je ale výsledek menší než nula, vytiskni, že je výsledek záporný.



Neúplná alternativa

Jak je vidět na obrázku, neliší se tento případ příliš od předchozího, jen je jedna jeho větev prázdná. Znamená to, že pokud se podmínka splní, provede se zadaný kód a pak program pokračuje dalším příkazem. Pokud podmínka pravdivá není, pokračuje se za kódem podmínky v provádění dalšího kódu. Příklad: Pokud proměnná *x* není prázdná, program ji vyprázdní, pokud je prázdná, program pokračuje dál.

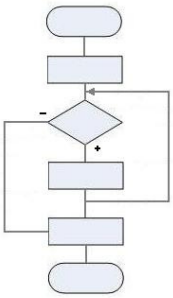


Několikanásobná alternativa

Poslední způsob větvení, kterým se budeme zabývat, je několikanásobné větvení. Podle toho, čemu se rovná porovnávaná proměnná, se provede určitý kód. Např.: Dejme tomu, že po uživateli vyžadujeme, ať zadá nějaké písmeno, a podle toho, jaké písmeno zadal, provedeme to, co uživatel chce. Právě v tomto případě se nám bude hodit tato několikanásobná alternativa.

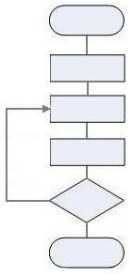
Uvedené způsoby se dají různě kombinovat, vnořovat a proplétat a tak dál, ale teď se pustíme do cyklů.

Cykly



Se vstupní podmínkou

Tento cyklus se bude provádět, dokud si nebude výraz v podmínce cyklu roven. Pokud si výraz bude roven ještě předtím, než se začne cyklus provádět, cyklus se neprovede vůbec. Záleží také na typu cyklu.



S výstupní podmínkou

Tento cyklus je velmi podobný tomu předchozímu s tím rozdílem, že podmínka je až na konci. Zaručuje nám to tedy, že tento cyklus se provede vždy alespoň jednou.

Pokud byste neměli možnost využít cyklů a větvení, nebyly byste prakticky schopni tvořit plnohodnotné programy.

Vývojové diagramy – 3. díl

Autor: Libor Beneš, korektura: Zdeněk Lehocký, 08. 07. 2011
http://programujte.com/clanek/2010041500-vyvojove-diagramy-3-dil/

Než se začínající programátor vrhne na učení nějakého jazyka, měl by si osvojit techniku „myslet jako počítač“. Rozumí se tím umět problém rozdělit na jednotlivé krátké části, dílčí úseky, které ve výsledku budou tento problém řešit, neboli vytvořit algoritmus. Proces tvoření algoritmu se nazývá algoritmizace a vývojové diagramy jsou jedním z nástrojů pro jejich zápis.

Předchozí dva díly (první, druhý) byly věnovány vývojovým diagramům z obecného hlediska. Víme tedy, z čeho se takový vývojový diagram může skládat a jak mohou vypadat různé varianty větvení a cyklení. Ale je rozdíl v tom znát cestu a jít po ní… [*Morfeus, film Matrix*]. Následujících několik dílů bude věnováno řešení konkrétních problémům od jednoduchých po složitější.

Tento díl je věnován úplně těm nejjednodušším algoritmům, které obsahují pouze vstup od uživatele, zpracování (výpočet) a vypsání výsledku. Začneme používat tyto 4 značky:

- označení začátku/konce
- spojení
- vstup/výstup
- příkaz

Značka pro vstup je stejná jako značka pro výstup (společná značka pro vstupně-výstupní operace). Rozlišovat je budeme zápisem textu do značky.

Dost bylo úvodu a přejděme k praktickým ukázkám. Začneme tím nejjednodušším.

Ahoj světe

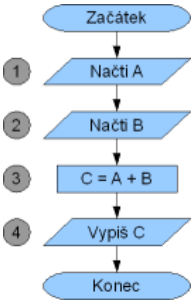
Téměř každá kniha nebo seriál o programování začíná aplikací typu „Hello World“. Tak ani my neuděláme výjimku a pro takovou aplikaci si vytvoříme vývojový diagram. Program je to velice jednoduchý, chceme pouze vypsat text na výstup, aby si ho uživatel mohl přečíst. K výpisu použijeme značku pro vstupně-výstupní operace s popisem činnosti.

„Ahoj světe“ je sice velmi hezký program, ale takových, které by pouze něco vypisovaly, je jako šafránu (pokud vůbec). Ostatní programy zpracovávají nějaká data a je v podstatě úplně jedno, jestli data zadává uživatel, čtou se ze souboru, jsou výsledkem komunikace, nebo jsou uložena v databázi. Všechny (s největší pravděpodobností) vývojové diagramy, které budou v rámci tohoto seriálu rozebrány, budou mít na vstupu načtení dat od uživatele, ale klidně by to mohlo být i načtení ze souboru nebo z databáze.

Sčítání dvou čísel

Zadáním úlohy je vytvořit algoritmus na sčítání dvou čísel. Každého, kdo začíná, tak určitě napadne otázka: jakých dvou čísel? Tím se dostaneme k jedné z vlastností, které musí každý navržený algoritmus splňovat, a to **hromadnost**. Hromadnost znamená, že algoritmus neslouží k řešení jedné úlohy (konkrétních dat), ale slouží k řešení celé skupiny úloh, které se od sebe liší jen vstupními daty.

Hromadnost – algoritmus neslouží k řešení jedné úlohy, ale k řešení celé skupiny úloh, které se od sebe liší jen vstupními daty.



Odpověď na otázku „součet jakých dvou čísel“ je tedy jednoduchá: libovolných dvou čísel. Tato libovolnost má samozřejmě svoje omezení daná požadavky na vstupní data (např. všechna celá čísla). Nebudeme tedy sčítat dvě konkrétní čísla (např. 5 + 8), ale dvě libovolná čísla, která reprezentujeme zástupným označením (jménem, např. číslo1 a číslo2). Hodnota takto označených čísel se může měnit, bude proměnná – odtud tedy název pro toto označení — **proměnná**.

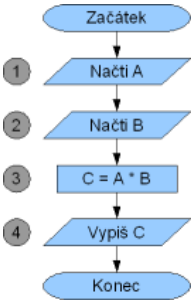
Sčítat budeme tedy dvě čísla, která načteme ze vstupu (zadá je uživatel) a budou zastoupena proměnnými. Následně je sečteme, výsledek opět uložíme do proměnné a vypíšeme. Tím je celý algoritmus hotov a můžeme ho zakreslit. Výsledný vývojový diagram obsahuje zadání prvního čísla (načte se a uloží do proměnné *A*) a druhého čísla (*B*). Následuje výpočet součtu, uložení výsledku do proměnné *C* a jejího následného vypsání.

Čísla vedle vývojového diagramu označují číslo kroku (označeny tak budou pouze vývojové diagramy v prvních několika dílech). Toto číslo použijeme v tabulce hodnot, která obsahuje hodnoty pro zvolenou sadu vstupních dat, a to z důvodu lepší názornosti. Tabulka hodnot se používá pro kontrolu správnosti algoritmu. Na obrázku si můžete prohlédnout tabulku pro zadané hodnoty 10 a 5 (včetně popisu).

	A	B	C	popis činnosti
1	10			zadání prvního čísla, ukládáme do proměnné A
2	10	5		zadání druhé čísla (B), proměnná A se nemění
3	10	5	15	výpočet součtu, výsledek se ukládá do C (A, B se nemění)
4	10	5	15	výpis výsledku (A, B, C se nemění)

A narážíme na další otázku začátečníků: a kde se ta čísla vezmou? Tato čísla simulují vstup od uživatele, jsou vybrána zcela náhodně a vybírá je ten, kdo bude tvořit algoritmus resp. vývojový diagram. Pokud se vám líbí třeba číslo 5, tak můžete do omrzení zadávat na vstupu číslo 5. Pro vyzkoušení algoritmu je ovšem velmi vhodné zadávat taková čísla (data), která budou testovat i tzv. mezní případy. V příštím díle si ukážeme, co takový mezní případ je a samozřejmě si pro něj vytvoříme i tabulku.

Násobení dvou čísel

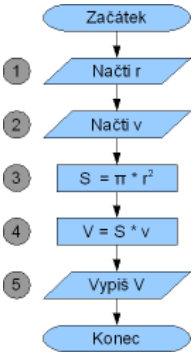


Zadáním úlohy je vytvořit algoritmus pro násobení dvou čísel. Opět tedy potřebujeme na vstupu dvě čísla, která vynásobíme a výsledek vypíšeme. Vstupem algoritmu budou dvě čísla a výstupem výsledek (jejich součin). Úloha je tedy v podstatě stejná jako v předchozím případě, a proto nebudeme na nic čekat a vývojový diagram pro tento algoritmus zakreslíme. Výsledný vývojový diagram se liší pouze v početním operátoru.

	A	B	C	popis činnosti
1	3			zadání prvního čísla, ukládáme do proměnné A
2	3	7		zadání druhé čísla (B), proměnná A se nemění
3	3	7	21	výpočet součinu, výsledek se ukládá do C (A, B se nemění)
4	3	7	21	výpis výsledku (A, B, C se nemění)

Tabulka hodnot zobrazuje obsah proměnných při zadání čísel 3 a 7. Čísla byla zvolena opět náhodně.

Výpočet objemu válcové nádoby



Zadáním úlohy je vytvořit algoritmus na výpočet objemu válcové nádoby. K výpočtu objemu válcové nádoby použijeme vzorec $V = \pi r^2 v$, kde r je poloměr podstavy a v je výška válce. K výpočtu tedy potřebujeme poloměr podstavy a výšku válce, které budou vstupem algoritmu. Výstupem algoritmu bude objem válce. Potřebujeme tedy zadat obě hodnoty, vypočítat objem a vypsat ho. Úloha je tedy opět velmi podobná těm předchozím. My si v ní z cvičných důvodů ovšem uděláme jednu změnu, a to takovou, že výpočet provedeme s mezivýsledkem. Nejprve si spočítáme obsah podstavy a následně celkový objem.

	r	v	S	V	popis činnosti
1	1				zadání poloměru, ukládáme do proměnné r
2	1	2			zadání výšky válce (v)
3	1	2	π		výpočet obsahu podstavy (pomocná proměnná S)
4	1	2	π	2π	výpočet objemu V s využitím mezivýsledku S
5	1	2	π	2π	vypsání výsledku (objemu V)

Tabulka hodnot zobrazuje obsah proměnných v jednotlivých krocích při zadání čísel 1 a 2. Čísla byla zvolena opět náhodně.

V příštím díle si přidáme podmínku, která nám umožní větvit program a tvořit složitější konstrukce.

Vývojové diagramy – 4. díl

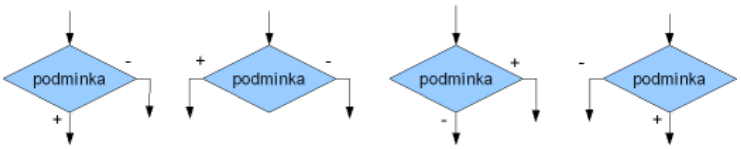
Autor: Libor Beneš, korektura: Zdeněk Lehocký, 13. 07. 2011
http://programujte.com/clanek/2010042600-vyvojove-diagramy-4-dil/

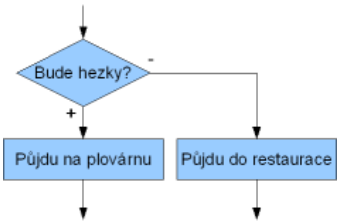
V dnešním díle začneme s podmínkami, které nám umožní dělat složitější a komplexnější programy.

Minule jsme si ukázali, jak načíst data, zpracovat je a zobrazit výsledek. Programů, které by měly takto jednoduchou strukturu, je velmi málo. V každém větším programu potřebujeme například zkontrolovat vstupní data, rozhodnout se, jak budeme výpočet provádět apod. Potřebujeme umět **větvit program**, tj. umět na základě něčeho rozhodnout, jakou cestou se program vydá dál. K tomuto účelu slouží podmínky.

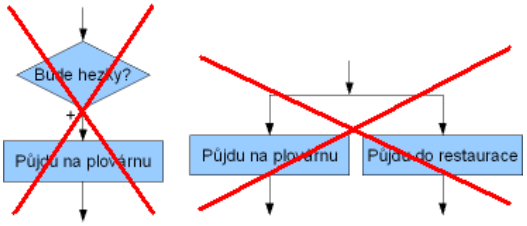
Podmínky mají tvar kosočtverce s jedním vstupem a dvěma výstupy. Rozhodování, kterým výstupem se má program dále ubírat, je dán právě tou podmínkou (zapsanou uvnitř). Jedná se o rozhodování platí/neplatí – označení ANO/NE nebo 1/0 nebo +/–. Jedním výstupem program pokračuje v případě, že je podmínka splněna. Takový výstup budeme označovat **+**. Druhým výstupem pokračuje program v případě, že podmínka není splněná. Takový výstup budeme označovat **–**.

Do podmínky se nejčastěji vstupuje shora (vstup). Výstupy mohou být různě kombinované, jak je vidět na následujícím obrázku. Proto je nutné označovat výstupy, protože v zájmu přehlednosti diagramu můžeme použít kterýkoliv z těchto zápisů. Nejčastěji používaný zápis je první uvedený.





Jak tedy s podmínkou zacházet? Řekněme, že si děláme plán na odpoledne a máme připravené dvě varianty. Když bude hezky, půjdeme se osvěžit na plovárnu, a když nebude hezky, tak se půjdeme osvěžit do restaurace. Jaké bude odpoledne počasí, to se teprve uvidí, ale jednu věc můžeme říci již teď: provedeme pouze jednu z těchto činností. Buď budeme tekutinou polévat sebe, nebo ji budeme lít do sebe. Uděláme pouze jednu z činností – vždy projdeme pouze jednu větev za podmínkou. Na otázku, která to bude, můžeme odpověď až ve chvíli, kdy budeme testovat podmínku. Tak jako v našem příkladu musíme čekat až do odpoledne.

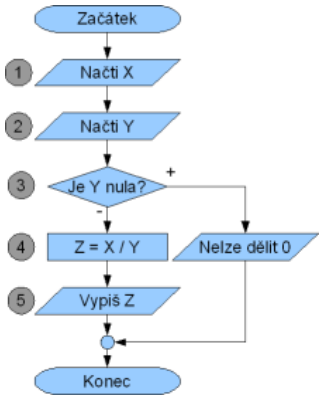


Důležité je ve vývojovém diagramu zakreslit obě větve. Vývojový diagram s podmínkou, která má ošetřenu pouze jednu z možných variant, je neúplný. Takto zakreslený algoritmus není jednoznačný, protože v každém kroku nemůžeme přesně určit, jaký bude následovat. Stejně tak není možné rozvětvit vývojový diagram (algoritmus) bez podmínky. Jde

o další důležitou vlastnost, kterou musí algoritmus splňovat. V každém kroku algoritmu musí být jasné, který krok bude následovat.

Jednoznačnost – v každém kroku algoritmu musí být jasné, který krok bude následovat.

Dělení dvou čísel



Teorie už máme dost, takže se pustíme do jednoduchého příkladu. V minulém díle jsme si udělali sčítání a násobení dvou čísel. Nyní si tato dvě čísla vydělíme. Takže opět jen změníme znaménko, nebo ne? V čem se liší dělení od násobení nebo sčítání? Je možné udělat 10 : 5? Určitě ano a výsledek je 2. A je možné udělat 3 : 0? Není, nulou nelze dělit a tuto skutečnost musíme v našem algoritmu brát v potaz, neboli musíme rozlišit stav, kdy je jmenovatel roven nule a kdy není. Toto rozlišení uděláme právě prostřednictvím podmínky.

Čísla, která budeme dělit, opět načteme ze vstupu od uživatele. V případě, že jmenovatel (Y) není nulový, tak výsledek můžeme spočítat a vypsat. V opačném případě (jmenovatel je nulový) výpočet provést nelze. Musíme proto dát vědět uživateli, že není něco v pořádku, tj. vypíšeme hlášení o nemožnosti dělení nulou. A ještě jedno připomenutí: větve **+** a **–** za podmínkou neznamenají, že by to bylo pro kladná a záporná čísla. Větev **+** je určena pro případ, že byla podmínka splněna a větev **–** je určena pro případ, že podmínka splněna není.

Následují dvě tabulky pro různé hodnoty na vstupu tak, aby byl ukázán průchod vývojovým diagramem pro obě možnosti. V prvním případě není podmínka splněna a v druhém případě podmínka splněna je.

	X	Y	Z	popis činnosti
1	10			zadání prvního čísla, ukládáme do proměnné X
2	10	5		zadání druhého čísla (Y), proměnná X se nemění
3	10	5		test Y na hodnotu 0 → **není (- větev)** (X, Y se nemění)
4	10	5	2	výpočet dělení (X, Y se nemění)
5	10	5	2	vypsání výsledku (X, Y, Z se nemění)

	X	Y	Z	popis činnosti
1	3			zadání prvního čísla, ukládáme do proměnné X
2	3	0		zadání druhé čísla (Y), proměnná X se nemění
3	3	0		test Y na hodnotu 0 → **je (+ větev)** (X, Y se nemění)
4				Nelze dělit 0

Tímto jednoduchým příkladem dnešní díl zakončíme. S podmínkami budeme pokračovat i příště, kde si přidáme relační a spojovací operátory.

Nejdůležitější na celém programování je ale to, co nedělám ani v profesionálním ani v soukromém životě – ROZLOŽIT PROBLÉM NA MENŠÍ DÍLY, TY ROZDĚLIT NA MENŠÍ DÍLY ..... AŽ DOSTANU DÍLY, KTERÉ JSEM SCHOPEN JEDNODUŠE VYŘEŠIT.

Vývojové diagramy – 5. díl

Autor: Libor Beneš, korektura: Zdeněk Lehocký, 20. 07. 2011
http://programujte.com/clanek/2010051900-vyvojove-diagramy-5-dil/

V dnešním díle budeme pokračovat s podmínkami. Ukážeme si relační operátory a operátory pro spojení více podmínek.

V minulém díle jsme si ukázali, jak se pracuje s podmínkou. Již víme, že větev označená **+** se prochází v případě splnění podmínky. Větev označená **–** se naopak prochází v případě nesplnění podmínky. Dnes si předvedeme, jak k zápisu využít relační operátory. A dále vysvětlíme, jak spojovat více podmínek pomocí spojovacích operátorů.

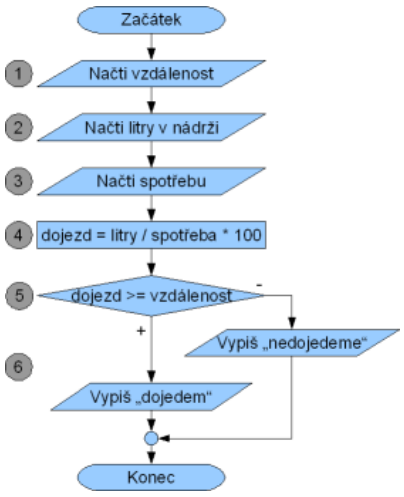
Při prvním seznámení s podmínkou jsme použili textový zápis. Tento způsob je samozřejmě možné použít vždy. Používání relačních operátorů není nezbytně nutné, ale vede k rychlejšímu a často i přesnějšímu zápisu a hlavně je tento zápis velmi blízký programovacímu jazyku.

Relační operátory jsou operátory, které známe z matematiky. Jejich funkcí je zjistit (porovnat) vztah mezi dvěma operandy. Minule jsme zapsali podmínku: *Je Y nula?* S pomocí relačního operátoru bychom napsali: **Y = 0**. Nepoužívá se otazník jako za větou, protože to se rozumí samo sebou – zapisujeme přece podmínku, ve které se na něco ptáme.

Nyní si uvedeme všechny relační operátory:

- **=** …… rovno - X = Y: Je X rovno Y?
- **==** …… rovno (ekvivalentní zápis) - X == Y: Je X rovno Y?
- **>** …… větší než - X > Y: Je X větší než Y?
- **<** …… menší než - X < Y: Je X menší než Y?
- **<>** …… nerovno - X <> Y: Je X různé od Y? nebo Není X rovno Y?
- **!=** …… nerovno (ekvivalentní zápis) - X != Y: Je X různé od Y? nebo Není X rovno Y?
- **>=** …… větší nebo rovno - X >= Y: Je X větší nebo rovno Y?
- **<=** …… menší nebo rovno - X <= Y: Je X menší nebo rovno Y?

Výpočet dojezdu automobilu



Použití relačních operátorů si ukážeme na jednoduchém příkladu výpočtu dojezdu automobilu. Příklad je na principu, kdy známe vzdálenost, kterou chceme urazit, a dále známe aktuální obsah nádrže a spotřebu. Chceme spočítat, jestli vzdálenost máme šanci urazit, nebo ne. Na vstupu od uživatele tedy budou tři hodnoty: vzdálenost, kterou chceme urazit, aktuální obsah nádrže v litrech a spotřeba automobilu. Výsledkem algoritmu má být oznámení, zda zadanou vzdálenost můžeme urazit, nebo ne (spotřeba se uvádí v litrech na 100 ujetých kilometrů, proto je ve vzorci násobení hodnotou 100).

K porovnání dojezdu a zadané vzdálenosti je použit relační operátor větší nebo rovno. Pokud je tato podmínka splněna, pak je dojezdová vzdálenost větší nebo rovna zadané vzdálenosti, a proto víme, že ji s obsahem nádrže urazíme. V opačném případě, tj. vypočítaná dojezdová vzdálenost je menší než zadaná, ji neurazíme.

Následující dvě tabulky obsahují jednotlivé kroky pro obě možnosti. První obsahuje hodnoty proměnných pro případ, kdy na obsah nádrže zadanou vzdálenost urazíme. V druhé tabulce projdeme druhou větev, tj. obsah nádrže na zadanou vzdálenost nestačí.

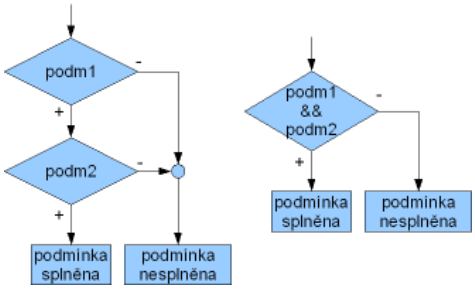
	v	l	s	d	popis činnosti
1	100				zadání vzdálenosti (v)
2	100	21			zadání aktuálního obsahu nádrže (l)
3	100	21	7		zadání spotřeby (s)
4	100	21	7	300	výpočet dojezdové vzálenosti ze zadaných parametrů
5	100	21	7	300	test: je daná vzdálenost menši než vypočítaný dojezd?
6	100	21	7	300	ANO je (+) → Vypiš „dojedem"

	v	l	s	d	popis činnosti
1	200				zadání vzdálenosti (v)
2	200	7			zadání aktuálního obsahu nádrže (l)
3	200	7	7		zadání spotřeby (s)
4	200	7	7	100	výpočet dojezdové vzálenosti ze zadaných parametrů
5	200	7	7	100	test: je daná vzdálenost menši než vypočítaný dojezd?
6	200	7	7	100	NE není (-) → Vypiš „nedojedem"

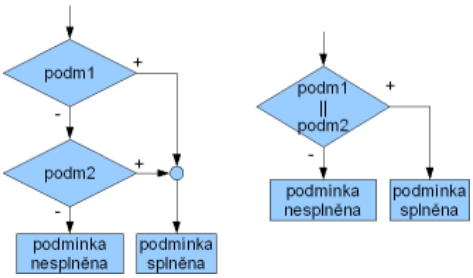
Často se stává, že rozhodnutí, kterou větví se budeme ubírat, závisí na více okolnostech. Musíme proto provádět více testů, tj. více podmínek. První možností je zapsat každou jednu podmínku do samostatného objektu a postupně ošetřovat jednotlivé větve. Nebo použijeme tzv. spojovací operátor. Spojovací operátory jsou dva:

- **a** – tento spojovací operátor říká, že musí být splněny obě podmínky, které spojuje, aby byla podmínka vyhodnocena jako splněná; operátor se může značit slovem **a**, anglickým překladem **and**, případně značkou **&&**,
- **nebo** – tímto operátorem se říká, že musí být splněna alespoň jedna ze dvou spojovaných podmínek, aby byl celek vyhodnocen jako splněný; operátor se může značit slovem **nebo**, anglickým překladem **or**, případně značkou **||**.

Opět je možné použít více zápisů. My budeme používat zápis pomocí znaků, tj. && a ||.



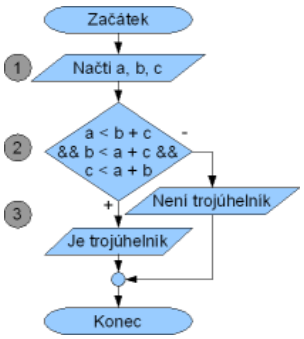
Spojovací operátor && má ekvivalentní zápis dle obrázku. Jedná se o podmínky, které jsou řazeny za sebou a všechny musí být splněny, aby výsledkem byla splněná podmínka. Spojovací operátor && má tedy pouze jeden úspěšný scénář, ale více neúspěšných. Jinak řečeno, musí být splněny všechny části, aby byla celá podmínka splněná. V programovacích jazycích je to většinou tak, že při první nesplněné části je podmínka označena za nesplněnou.



Spojovací operátor || má ekvivalentní zápis dle obrázku. Jedná se o podmínky, které jsou řazeny za sebou a musí být splněna pouze jedna, aby výsledkem byla splněná podmínka. Spojovací operátor || má tedy více úspěšných scénářů a jeden neúspěšný. Jinak řečeno, musí být splněna alespoň jedna část, aby byla celá podmínka splněná. V programovacích jazycích je to většinou tak, že při první splněné části je podmínka označena za splněnou.

Spojovací operátory se mohou libovolně kombinovat. V jedné podmínce může být několik spojovacích operátorů && a/nebo ||. Stejně jako v matematice lze ovlivnit „pořadí" uzavřením podmínek do závorek. Například chceme provést nějakou akci pro body z I. nebo III. kvadrantu. Bod v I. kvadrantu má *x* a *y* souřadnici kladnou a ve III. kvadrantu má obě souřadnice záporné. Pro I. kvadrant by podmínka byla: `x > 0 && y > 0`, pro III. kvadrant by podmínka byla `x < 0 && y < 0`. Celá podmínka by byla spojením obou předchozích s tím, že musí být splněna alespoň jedna z nich (bod může ležet pouze v jednom kvadrantu): `(x > 0 && y > 0) || (x < 0 && y < 0)` Stejně bychom podmínku mohli zapsat i pomocí slovního zápisu operátorů: `(x > 0 a y > 0) nebo (x < 0 a y < 0)` nebo `(x > 0 and y > 0) or (x < 0 and y < 0)`.

Trojúhelník



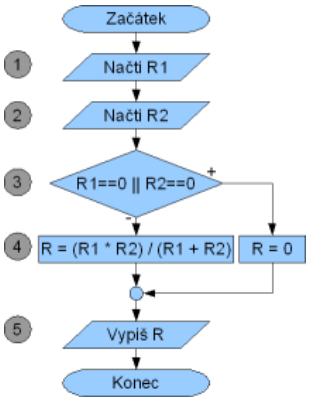
Spojovací operátor uplatníme v jednoduchém příkladu na zjištění, jestli tři úsečky mohou tvořit trojúhelník. Zadání úlohy je následující: vytvořte algoritmus pro zjištění, zda zadané tři strany (a, b, c) mohou tvořit trojúhelník. K řešení použijeme trojúhelníkovou nerovnost. Od uživatele dostaneme zadané tři strany a musí platit, že součet kterýchkoliv dvou z nich musí být větší než třetí strana. Máme tedy tři podmínky: `a < b + c`, `b < a + c`, `c < a + b` a všechny tyto podmínky musí platit najednou, tj. musíme použít spojovací operátor && (musí platit první podmínka **a** druhá podmínka **a** třetí podmínka). Výsledná podmínka tedy bude: `a < b + c && b < a + c && c < a + b`.

Následují dvě tabulky pro různé hodnoty na vstupu tak, aby byl ukázán průchod vývojovým diagramem pro obě možnosti. V prvním případě je podmínka splněna – zadané strany mohou tvořit trojúhelník. V druhém případě není podmínka splněna – zadané strany trojúhelník tvořit nemohou.

	a	b	c	popis činnosti
1	10	10	10	zadání délek stran trojúhelníka a, b, c
2	10	10	10	test, mohou-li strany tvořit trojúhelník
3	10	10	10	10 < 10 + 10 → podmínka splněna (+ větev) – **Je Δ**

	a	b	c	popis činnosti
1	1	5	10	zadání délek stran trojúhelníka a, b, c
2	1	5	10	test, mohou-li strany tvořit trojúhelník
3	1	5	10	10 < 5 + 1 → podmínka nesplněna (- větev) – **Není Δ**

Paralelní zapojení dvou rezistorů



Dnešní díl zakončíme příkladem algoritmu na výpočet celkové hodnoty odporu dvou paralelně zapojených rezistorů, na kterém si ukážeme použití spojovacích operátorů. A také si na něm demonstrujeme, že lze podmínku zapsat různými způsoby, a také to, že obě větve mohou být platné, tj. že jedna z větví neoznačuje chybový stav.

Zadání příkladu je tedy následující: vytvořte algoritmus (a pro něj vývojový diagram) na výpočet celkové hodnoty odporu dvou paralelně zapojených rezistorů. Vstupem od uživatele tedy budou dvě hodnoty odporů. Výstupem algoritmu bude výsledná hodnota paralelního zapojení. Vzorec pro výpočet je následující: `R = (R1 × R2) / (R1 + R2)`. Obě hodnoty paralelně zapojených rezistorů opět načteme od uživatele. Ze vzorce je vidět, že je zde dělení, tj. musíme testovat, jestli součet R1 + R2 není roven 0. Zapsaná podmínka by mohla vypadat takto: `(R1 + R2) == 0` nebo bychom mohli testovat pouze jednotlivé hodnoty (neboť odpor nemůže být záporný): `R1 == 0 && R2 == 0`.

Ke vzorci jednu poznámku. Nejpodstatnější na tomto algoritmu je použitý vzorec. I kdyby byl algoritmus vymyšlený maximálně optimálně, ale byl by v něm použitý špatný vzorec, tj. dával by chybné výsledky, tak takový algoritmus není správný. Každý algoritmus musí splňovat podmínku správnosti – musí dávat správné výsledky. Použité vzorce, postupy a metody výpočtu musí být správné.

Správnost – algoritmus musí dávat správné výsledky.

Než podmínku zapíšeme, tak se zamyslíme nad tím, co se vlastně stane, když jeden nebo oba rezistory budou nulové. Výsledný odpor v takovém případě bude také 0 (v čitateli by bylo násobení 0), neboli je možné podmínku zapsat i ve tvaru, který je použit ve vývojovém diagramu: `R1 == 0 || R2 == 0`, kde testujeme, jestli hodnota rezistoru R1 je 0 nebo hodnota rezistoru R2 je 0. Pokud alespoň jeden z nich (nebo oba) jsou nulové, tak procházíme + větev a výsledný odpor bude 0. V opačném případě (oba rezistory jsou nenulové) se provede výpočet dle vzorce.

Následují dvě tabulky pro různé hodnoty na vstupu tak, aby byl ukázán průchod vývojovým diagramem pro obě možnosti. V prvním případě není podmínka splněna (provádí se výpočet přes vzorec) a v druhém případě je podmínka splněna (výsledek je 0).

	R1	R2	R	popis činnosti
1	10			zadání prvního odporu, ukládáme do proměnné R1
2	10	10		zadání druhého odporu (R2)
3	10	10		test R1 nebo R2 na hodnotu 0 → **není (- větev)**
4	10	10	5	výpočet celkového odporu
5	10	10	5	vypsání výsledku (R1, R2, R se nemění)

	R1	R2	R	popis činnosti
1	0			zadání prvního odporu, ukládáme do proměnné R1
2	0	20		zadání druhého odporu (R2)
3	0	20		test R1 nebo R2 na hodnotu 0 → **je (+ větev)**
4	0	20	0	výsledný odpor je 0
5	0	20	0	vypsání výsledku (R1, R2, R se nemění)

Příště si ukážeme složitější příklady, ve kterých bude více podmínek.

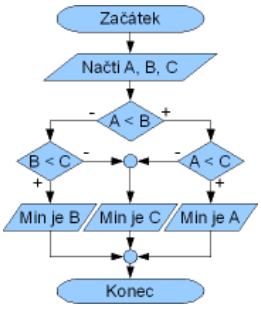
Vývojové diagramy – 6. díl

Autor: Libor Beneš, korektura: Zdeněk Lehocký, 29. 07. 2011
http://programujte.com/clanek/2010060400-vyvojove-diagramy-6-dil/

V tomto dílu zakončíme téma podmínky složitějšími příklady, na kterých si ukážeme, že podmínek může být ve vývojovém diagramu více.

Podmínky se používají na řízení toku programu, určujeme jimi, kudy se bude program ubírat dále. Ukázky, které jsou v předchozím dílu, by snadno mohly vést k mylnému dojmu, že ve vývojovém diagramu může být pouze jedna podmínka. Ve vývojovém diagramu není žádné omezení na počet a kombinace podmínek. Složitější programy obsahují desítky, stovky, tisíce,… podmínek. A stejně tak složitější vývojové diagramy mohou obsahovat více podmínek. V tomto dílu už nebudou u vývojových diagramů tabulky na průchody – můžete si je zkusit vytvořit sami.

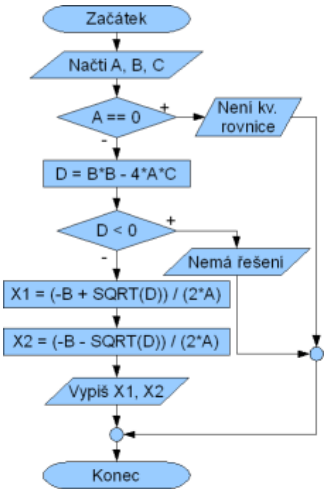
Hledání minima



Prvním příkladem je hledání minima ze tří zadaných čísel. Zadání tohoto příkladu zní: najděte minimální číslo ze tří zadaných. Na vstupu tedy budou tři čísla, která zadá uživatel. Výstupem bude vypsané číslo, které je nejmenší (zadá-li uživatel 1, 2, 3, tak výsledkem algoritmu musí být vypsání čísla 1). Je více způsobů, jak zadanou úlohu řešit. Vzhledem k tomu, že nehledáme optimální nebo nejrychlejší řešení, ale pouze si demonstrujeme algoritmy s použitím podmínek, tak použijeme metodu „porovnání každý s každým“. Z těchto porovnání je samozřejmě možné některá vynechat. Pokud bude platit, že A < B, tak již víme, že A je menší, než B. Dále tedy stačí A porovnat s C a B již s C porovnávat nemusíme, protože víme, že A je menší (jenom připomínám, že hledáme nejmenší číslo).

Výsledný vývojový diagram je na obrázku. Nejprve se tedy porovná A s B. Podle výsledku tohoto porovnání se buď C porovnává s A (pokud předchozí podmínka byla splněna, neboli A je opravdu menší než B), nebo se C porovnává s B (v opačném případě). Pokud bychom stejnou metodu použili i pro více vstupních hodnot, tak by nám strom vývojového diagramu utěšeně rostl. Proto se častěji používá metoda s uloženým lokálním minimem (maximem), kterou si ukážeme v některém z následujících dílů.

Kvadratická funkce



Druhý příklad je na výpočet reálných kořenů kvadratické rovnice. Zadání úlohy je následující: vytvořte algoritmus pro výpočet reálných kořenů kvadratické rovnice, která je zadaná koeficienty A, B, C (dle vzorce $Ax^2 + Bx + C = 0$). Na vstupu od uživatele dostaneme tři hodnoty (A, B, C) a výsledkem budou vypočtené kořeny.

Toto je první úloha, ve které se zmíním o kontrole vstupu od uživatele. **V reálných programech je nutné vstup od uživatele kontrolovat vždy.** Tam, kde váš program bude očekávat číslo, uživatel jistě zadá slovo „pokus" apod. Kvadratická rovnice je kvadratická pouze v případě, že A je nenulové. V případě, že A je nulové, pak se jedná o lineární rovnici a tu neřešíme. Při zadání A = 0 musí být výsledkem algoritmu hlášení, že zadané koeficienty netvoří kvadratickou rovnici.

Dále postupujeme přes výpočet diskriminantu. Vzorec pro výpočet diskriminantu je: $D = B^2 - 4AC$. Řešení kvadratické rovnice v oboru reálných čísel je pouze v případě, že vypočtený diskriminant není záporný. To bude druhá podmínka v algoritmu. V případě záporného algoritmu nemá kvadratická rovnice řešení (v oboru reálných čísel). Pokud není diskriminant záporný, pak je možné vypočítat kořeny a vypsat je.

Jednoduchá kalkulačka

Posledním příkladem je jednoduchá kalkulačka. Zadání je následující: vytvořte algoritmus pro jednoduchou kalkulačku, která bude umět načíst 2 operandy a operátor (+, -, ×, ÷), vypočítat výsledek a vypsat ho. Na vstupu od uživatele tedy budou 2 čísla a operátor (znak), který bude signalizovat operaci, která se má provést. Musíme tedy vytvořit rozhodovací strom, kde budeme kontrolovat, jestli se jedná o jednu z podporovaných operací, a pokud ano, tak provedeme výpočet. V případě, že zadaný operátor nebude mezi podporovanými, vypíšeme hlášení, že se nejedná o podporovanou operaci (kontrola vstupu od uživatele).

Vzhledem k tomu, že mezi vybranými operátory je i dělení, tak musíme (stejně jako ve 4. dílu) kontrolovat, jestli by nedošlo k dělení nulou. Před samotným dělením tak musí být podmínka, která této variantě zabrání, a místo chyby programu (tak by dělení nulou v programu končilo) zobrazíme uživateli hlášení, že nulou nelze dělit. Nakonec vypíšeme výsledek, který jsme vypočítali.

Krátká poznámka na konec. V této úloze by se dala s výhodou použít podmínka s více větvemi (switch), kde se každá větev prochází pro přesně danou hodnotu proměnné v podmínce. V našem případě by v podmínce bylo pouze *oper* a jednotlivé větve by byly označené +, -, ×, ÷. V tomto seriálu takto zapsanou podmínku používat nebudeme.

Tím tento díl zakončíme a příště na nás čekají cykly.

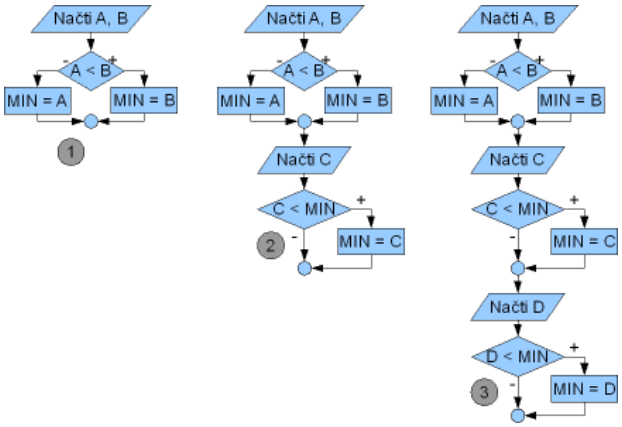
Vývojové diagramy – 7. díl

Autor: Libor Beneš, korektura: Martin Šimeček, 08. 08. 2011
http://programujte.com/clanek/2010061000-vyvojove-diagramy-7-dil/

Dnešním dílem začneme cykly. Nejprve si ukážeme, k čemu jsou dobré, a následně si vysvětlíme první typ, tj. cyklus s daným počtem opakování.

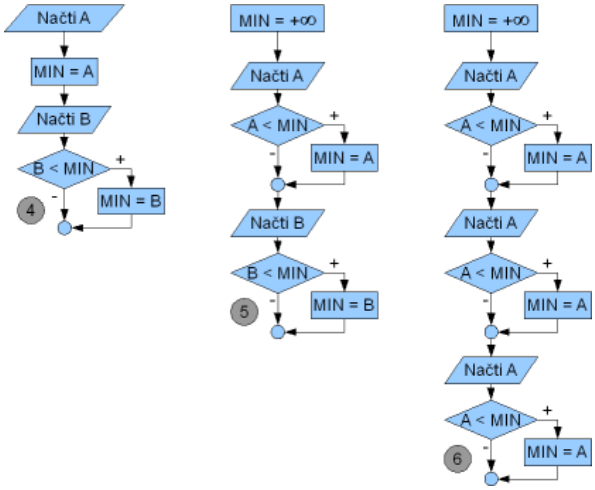
Co to vlastně takový cyklus je a k čemu je dobrý? Cyklus je určitá část algoritmu, která se provádí opakovaně (nejlépe tolikrát, kolikrát jsme chtěli :)). Sebelepší vysvětlování nenahradí příklad, takže si jeden postupný uděláme. V minulém díle jsme hledali minimum ze tří čísel. Kdybychom chtěli najít minimum z pěti čísel, z deseti čísel, z tisíce čísel atd., tak by se nám vývojový diagram neúměrně rozkošatil. A smyslem vývojových diagramů je zápis programů, nikoliv popsání všech dostupných papírů.

Když začínáme „ty cykly", tak vás asi napadne, že k řešení použijeme cyklus. Ale kde je tam opakující se kód? Jsou tam sice podmínky, ale každá je jiná, resp. v každé se porovnávají jiné dvě proměnné. Řešením problému opravdu bude cyklus, ale také trochu jiný pohled na daný problém.



Začneme od zjištění minima ze dvou zadaných hodnot (obrázek 7.1 (1)). Výsledkem bude menší z obou hodnot. Kdybychom chtěli tento algoritmus rozšířit na tři, tak můžeme postupovat stejně, jako v předchozím díle nalezené menší číslo porovnáme se třetím zadaným. Můžeme to ovšem udělat i tak, že si nalezené minimum uložíme a další porovnání provedeme pouze s tímto minimem (obrázek 7.1 (2)).

Pokud bychom přidali další porovnání, tak bychom ho opět mohli dělat pouze s uloženým minimem (obrázek 7.1 (3)). Tím dostáváme opakující se kód. Jsou zde ovšem ještě dvě věci, které by bylo potřeba změnit. Načtení provádím do proměnné, která se pokaždé jmenuje jinak, a za druhé by bylo vhodné (pokud to půjde) upravit počátek tak, aby jeho zápis vypadal stejně jako následně přidávané sekvence.



Nejprve upravíme počáteční sekvenci, tj. vývojový diagram z obrázku 7.1 (1). Je několik možností a my si tu ukážeme dvě hlavní. V první jde o to, že první načtené číslo se stane počáteční inicializací hledaného minima, které si průběžně ukládáme do proměnné *MIN* (obrázek 7.2 (4)). Druhou možností, která je mému srdci bližší (a kterou budu v seriálu používat), je počáteční inicializaci proměnné *MIN* udělat na nějakou strašně velkou hodnotu. Nejlépe na tu maximální, kterou si označíme *+nekonečno*. Kdybychom hledali maximum, tak by počáteční inicializace byla na-nekonečno.

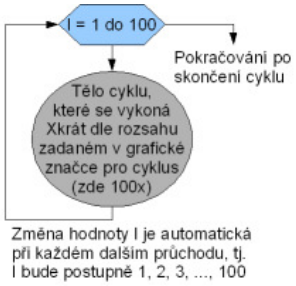
V principu jde o to, aby z prvního porovnání vyšlo menší (resp. větší při hledání maxima) námi zadané číslo, ať je jakékoliv. To nám zajistí pouze a jenom nějaká maximální hodnota při hledání minima (resp. minimální hodnota při hledání maxima), protože cokoliv je menší než *+nekonečno*. Výsledkem je vývojový diagram na obrázku 7.2 (5).

Inicializace je důležitá i z jiného hlediska. Každý algoritmus musí totiž splňovat podmínku opakovatelnosti, tj. pro stejná vstupní data musí dávat stejný výsledek. Kdybychom proměnnou *MIN* neinicializovali, tak by se nedalo minimálně při prvním porovnání určit, jakou má hodnotu, a výsledek takového porovnání by byl nejistý. Každá proměnná musí mít známou hodnotu před svým prvním použitím.

Opakovatelnost – algoritmus musí pro stejná vstupní data dávat stejný výsledek.

A jako poslední se zbavíme měnících se jmen proměnných. Když se podíváme, kde se načtená proměnná (třeba *A*) používá, tak zjistíme, že po načtení se s ní provede porovnání a případně se její hodnota uloží do proměnné *MIN*. Dále se již nepoužívá. Z toho vyplývá, že při dalším načítání nemusíme používat jinou proměnnou, ale můžeme využít tuto proměnnou opakovaně.

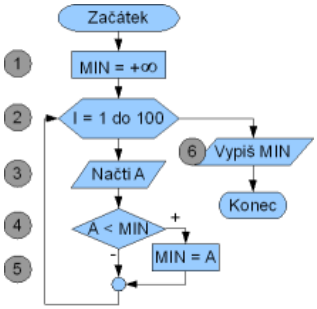
Výsledkem tedy bude vývojový diagram, který je na obrázku 7.2 (6) – výsledek diagramu pro nalezení maxima ze tří hodnot. Na tomto obrázku je vidět, že zde máme část, která se stále opakuje, a to včetně jmen proměnných. Nyní můžeme přistoupit k zápisu takového algoritmu s pomocí cyklu.



Nejprve si ukážeme, jak se takový cyklus zapisuje do vývojového diagramu. Pro cyklus s daným počtem opakování se uvádí mnoho zápisů. Mým favoritem se stal zápis z obrázku 7.3, neboť v něm je nejlépe graficky znázorněný cyklus a následný kód po jeho ukončení je oddělen od těla cyklu. Tělo je část kódu, která se má cyklicky opakovat. Počet opakování je dán rozsahem, který je zapsán ve značce pro cyklus. V něm se zadává rozmezí od-do, např. rozmezí od 1 do 100 znamená, že se cyklus provede stokrát.

Hodnota cyklu se mění každým průchodem automaticky a ukládáme si ji do nějaké pomocné proměnné (nejčastěji *i* od slova index). Stává se, že tuto hodnotu k ničemu nepotřebujeme, ale i tak ji ukládáme do proměnné. Asi častější případ je, že hodnotu použijeme. Jednoduché použití si ukážeme v příštím díle a až se dostaneme k polím a indexaci (odtud označení index), tak si ukážeme další velmi časté použití cyklů s daným počtem opakování, kde se nám uložená hodnota indexu bude hodit.

Hledání minima



Dnešní díl uzavřeme dokončením toho, co jsme na začátku rozpracovali, tj. vytvoření vývojového diagramu pro zjištění minima ze zadaných čísel pomocí cyklu. Zadání úlohy je následující: najděte minimum ze 100 zadaných čísel. Úlohu jsme si podrobně rozebrali, takže už jenom ve zkratce. Hodnotu hledaného minima uložíme do proměnné *MIN*, kterou si musíme inicializovat na nějaké velmi vysoké číslo (v našem případě je to *+nekonečno*). Inicializace se provádí na začátku, tj. před cyklem. Od uživatele musíme získat 100 čísel, takže načítání a následné porovnání bude tělem cyklu (tak, jak jsme si to ukázali výše). Po skončení cyklu nalezené minimum už jen vypíšeme.

Jak vidíte, tak zápis je přehledný a velmi nám zjednodušil úlohu, která by bez cyklu zabrala několik stránek. Další velkou výhodou zápisu v cyklu je snadná změna počtu opakování, tj. rozsahu, pro který se má cyklus vykonávat. Pokud bychom chtěli místo hledání minima ze 100 udělat příklad pro hledání minima z 389 450 čísel, tak ve vývojovém diagramu (a potažmo i programu) změníme pouze jedno číslo (100 na 389 450).

Následující tabulka obsahuje jednotlivé kroky pro průchod vývojovým diagramem. Vzhledem k opakování v cyklu se zde některé kroky (2, 3, 4, 5) cyklicky opakují. Pro všech 100 průchodů by tabulka byla velmi dlouhá, proto je její výpis zkrácen na několik průchodů. Podstatné je v ní uvedeno, tj. první průchod a hlavně poslední.

	MIN	I	A	popis činnosti
1	∞			inicializace minima (MIN) na +nekonečno
2	∞	1		vstup do cyklu, začínáme od zadané hodnoty (zde 1)
3	∞	1	10	zadání hodnoty uživatelem (např. 10)
4	∞	1	10	porovnání (10 < ∞) ➔ podmínka splněna (+ větev)
5	10	1	10	uložení průběžné hodnoty minima (MIN = 10)
2	10	2		další průchod cyklem, hodnota I se automaticky zvedá
3	10	2	15	zadání hodnoty uživatelem (např. 15)
4	10	2	15	porovnání (15 < 10) ➔ podmínka nesplněna (- větev)
5	10	2	15	žádná operace
2	10	3		další průchod cyklem, hodnota I se automaticky zvedá
3	10	3	-5	zadání hodnoty uživatelem (např. -5)
4	10	3	-5	porovnání (-5 < 10) ➔ podmínka splněna (+ větev)
5	-5	3	-5	uložení průběžné hodnoty minima (MIN = -5)
...	...	...	...	další a další průchody cyklem (nalezené MIN je -49)
2	-49	100		poslední průchod cyklem
3	-49	100	11	zadání hodnoty uživatelem (např. 11)
4	-49	100	11	porovnání (11 < -49) ➔ podmínka nesplněna (- větev)
5	-49	100	11	žádná operace
2	-49	101		hodnota I překročila zadané rozmezí ➔ konec cyklu
6	-49			vypsání nalezeného minima (MIN)

V příštím díle si ukážeme další příklady, kde použijeme cykly.

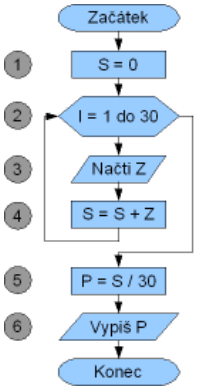
Vývojové diagramy – 8. díl

Autor: Libor Beneš, korektura: Martin Šimeček, 17. 08. 2011
http://programujte.com/clanek/2010061400-vyvojove-diagramy-8-dil/

V minulém díle jsme začali s cykly. V tom dnešním si ukážeme několik příkladů, kde využijeme jejich sílu.

V dnešním díle si projdeme několik příkladů, na nichž si vysvětlíme fungování cyklů a také si ukážeme, že rozmezí cyklu nemusí být dané napevno, ale může být ovlivněno vstupem uživatele, tj. že rozmezí nemusí být jen od 1 do 100. A vedle cyklů si předvedeme i operace, které se běžně používají v programování.

Průměr známek



Začneme příkladem na výpočet průměru zadaných známek. Zadání takového příkladu by mohlo být: vytvořte algoritmus pro výpočet průměru z 30 zadaných známek. Průměr známek se vypočítá jako jejich součet lomeno jejich počet. Ve výsledném algoritmu tedy potřebujeme zadat 30 známek, které sečteme a následně vypočteme jejich průměr.

Úlohu bychom mohli řešit tak, že bychom nechali uživatele najednou zadat 30 známek, následně bychom je všechny sečetli a vypočítali průměr. To by bylo řešení bez cyklu, jehož nevýhodou je nesnadné rozšíření třeba na jiný (větší, ale i menší) počet známek, ale i případná kontrola zadaného vstupu atd. Posledně jsme si ukázali, že s cyklem je možné takové úlohy řešit efektivněji.

Uživatel má zadat 30 známek, které může vkládat postupně. Udělejme si cyklus od 1 do 30, kde dostaneme možnost nechat si nějaký kousek algoritmu – tělo cyklu – 30krát zopakovat. Co by v těle takového cyklu mělo být? Když se nám bude tělo cyklu 30× opakovat, tak stačí vyřešit úlohu, kde se zadává jedna známka, která se přičte do nějakého součtu.

Úloha se tedy redukuje na zadání jedné známky a součet – vše 30× zopakujeme. Zadání známky je normální vstup od uživatele. Samozřejmě bychom měli kontrolovat, jestli zadal platnou školní známku (1 až 5), ale to v tuto chvíli řešit nebudeme, protože si chceme pouze ukázat cyklus.

Zbývá nám tedy vyřešit součet. Výsledný součet bude v proměnné *S*, zadaná známka je v proměnné *Z*. Nesmíme zapomenout na to, že sčítání se také bude provádět postupně v cyklu, stejně jako zadávání, takže v proměnné *S* budeme mít průběžný součet, ke kterému musíme hodnotu známky přičítat.

Provedeme tedy součet, který si uložíme: X = S + A a následně vrátíme do proměnné *S*: S = X. Toto lze v programování zapsat zkráceně, a to: S = S + A. Matematicky by tento zápis nedával smysl (pouze pro nulové A). Nesmíme ovšem zapomenout, že rovnítko v příkazu znamená přiřazení, nikoliv porovnání (není to podmínka). Zápis by se dal slovně popsat: *proveď součet proměnných S a A a výsledek ulož do proměnné S*. Stejný zápis lze použít pro všechny operandy (viz násobení u příkladu Faktoriál) a nemusí se vždy jednat o operaci s dvěma proměnnými (viz příklad Prvočíslo).

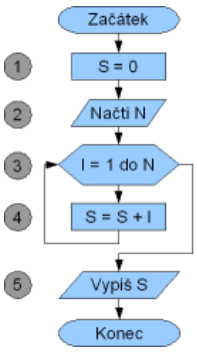
Někdy se používá příměr se sklenicemi a vodou nebo mincemi v dlani. Máme dvě sklenice (X, Y), u kterých chceme provést „součet obsahu". To můžeme provést tak, že obsah obou sklenic přelijeme do třetí (Z): Z = X + Y. Nic nám ovšem nebrání v tom, abychom obsah jedné sklenice (X) přelili do druhé (Y), takže výsledný součet bude ve sklenici Y: Y = X + Y nebo Y = Y + X (zápisy jsou ekvivalentní).

Výsledný vývojový diagram můžete vidět na obrázku. Tělem cyklu je zadávání a„sčítání. Po skončení cyklu se provede výpočet průměru, který nakonec vypíšeme. A„samozřejmě nesmíme zapomenout inicializovat proměnnou pro součet *S* – provádíme sčítání, takže inicializace bude na 0.

Následující tabulka obsahuje jednotlivé kroky pro průchod vývojovým diagramem. Vzhledem k opakování v cyklu se zde některé kroky (2, 3, 4) cyklicky opakují. Pro všech 30 průchodů by tabulka byla velmi dlouhá, proto je její výpis zkrácen na několik průchodů. Podstatné je v ní uvedeno, tj. první průchody a hlavně poslední. Tabulka končí výpočtem průměru ze zadaných známek.

	S	I	Z	P	popis činnosti
1	0				inicializace proměnné S
2	0	1			vstup do cyklu, začínáme od 1
3	0	1	2		načtení první známky
4	2	1	2		přičtení načtené známky Z (2) do součtu S
2	2	2			další průchod cyklem, hodnota I se zvedá
3	2	2	3		načtení další známky
4	5	2	3		přičtení načtené známky Z (3) do součtu S
...					další a další průchody cyklem
2	65	30			poslední průchod cyklem
3	65	30	1		načtení další známky
4	66	30	1		přičtení načtené známky Z (1) do součtu S
2	66	31			hodnota I překročila zadané rozmezí → konec cyklu
5	66			2.2	vypočtení průměru známek
6	66			2.2	vypsání vypočteného průměru známek

Součet číselné řady



Druhým dnešním příkladem bude součet (suma) číselné řady. Zadání je následující: vytvořte algoritmus pro součet číselné řady do zadaného čísla. Číselná řada začíná 1 a končí zadanou hodnotou. Například pro konečné číslo 8 se jedná o řadu 1, 2, 3, 4, 5, 6, 7, 8, jejímž součtem je 36 (= 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8).

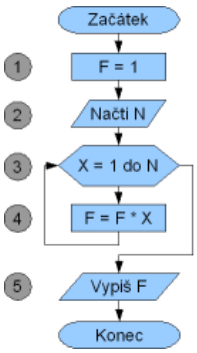
Řešení pomocí cyklu s daným počtem opakování se přímo nabízí. Rozmezí cyklu bude od 1 do nějakého zadaného N, které načteme od uživatele. Index cyklu se tedy bude měnit od 1 do N, každým průchodem se zvedne o 1. V těle cyklu bude stačit posčítat postupně měnící se hodnoty indexu. Na tomto příkladu vidíme, že index je možné používat jako jakoukoliv jinou proměnnou.

Výsledný součet uložíme do proměnné *S*. Opět použijeme zápis S = S + I, který znamená, že součet hodnot *S* a *i* uložíme do proměnné *S*. Protože provádíme součet, tak počáteční hodnota (inicializace) bude stejně jako v předchozím příkladu 0. Počáteční hodnota nemusí být vždy nulová, o tom se přesvědčíme v dalším příkladu.

Následující tabulka obsahuje jednotlivé kroky pro průchod vývojovým diagramem. Vzhledem k opakování v cyklu se zde některé kroky (3, 4) cyklicky opakují. Pro všech 10 průchodů (uživatel do N v tabulce zadal 10) by tabulka byla velmi dlouhá, proto je její výpis zkrácen na několik průchodů. Podstatné je v ní uvedeno, tj. první průchody a hlavně poslední.

	S	N	I	popis činnosti
1	0			inicializace proměnné S
2	0	10		načtení N – délka sčítané řady čísel
3	0	10	1	vstup do cyklu, začínáme od 1
4	1	10	1	přičtení hodnoty indexu I (1) do součtu S
3	1	10	2	další průchod cyklem, hodnota I se automaticky zvedá
4	3	10	2	přičtení hodnoty indexu I (2) do součtu S
...	...	...	...	další a další průchody cyklem
3	45	10	10	poslední průchod cyklem
4	55	10	10	přičtení hodnoty indexu I (10) do součtu S
3	55	10	11	hodnota I překročila zadané N → konec cyklu
5	55	10		vypsání hodnoty součtu řady čísel S

Faktoriál



Dalším dnešním příkladem bude výpočet faktoriálu. Zadání by mohlo znít: vytvořte algoritmus pro výpočet faktoriálu ze zadaného čísla. Faktoriál je definován jako: n! = 1 × 2 × 3 × 4 × ... × n. Na vstupu od uživatele tedy bude jedno číslo. Na výstupu bude výpis vypočteného faktoriálu.

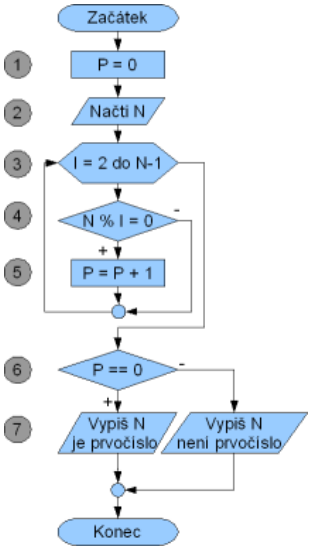
V podstatě jde o stejný příklad jako v předchozím případě. Jediný rozdíl je v tom, že jednotlivé hodnoty číselné řady mezi sebou nesčítáme, ale násobíme. Místo sčítání tak v těle cyklu použijeme násobení a opět využijeme zkrácený zápis: F = F × I (vynásob F a I a výsledek ulož do proměnné *F*).

Druhý podstatný rozdíl je v inicializaci proměnné *F*. U součtu jsme nastavili výchozí nulu – před součtem musí být celková hodnota 0. U součinu by 0 v inicializaci měla za následek to, že výsledek by byl opět 0 pro jakékoliv číslo. Proto musíme zvolit hodnotu 1. Můžeme obecně volit jiné výchozí hodnoty, vždy záleží na konkrétním příkladu.

Následující tabulka obsahuje jednotlivé kroky pro průchod vývojovým diagramem. Vzhledem k opakování v cyklu se zde některé kroky (3, 4) cyklicky opakují. Pro všech 5 průchodů (uživatel do N v tabulce zadal 5) by tabulka byla dlouhá, proto je její výpis zkrácen na několik průchodů. Podstatné je v ní uvedeno, tj. první průchody a hlavně poslední.

	F	N	X	popis činnosti
1	1			inicializace proměnné F
2	1	5		načtení N – hodnota, ze které chceme počítat N!
3	1	5	1	vstup do cyklu, začínáme od 1
4	1	5	1	vynásobení hodnoty v F hodnotou indexu X (1)
3	1	5	2	další průchod cyklem, hodnota X se zvedá
4	2	5	2	vynásobení hodnoty v F hodnotou indexu X (2)
...	...	...	...	další a další průchody cyklem
3	24	5	5	poslední průchod cyklem
4	120	5	5	vynásobení hodnoty v F hodnotou indexu X (5)
3	120	5	6	hodnota X překročila zadané N → konec cyklu
5	120	5		vypsání hodnoty vypočteného faktoriálu N!

Prvočíslo



Poslední dnešní příklad bude zjištění, jestli je zadané číslo prvočíslo. Zadání takového příkladu by mohlo být: sestavte algoritmus pro zjištění, jestli zadané číslo je prvočíslo. Prvočíslo je celé číslo, které je dělitelné pouze 1 a samo sebou. Vstupem algoritmu tedy bude zadané číslo a výstupem algoritmu bude vypsání, jestli zadané číslo je, nebo není prvočíslo.

Metod na zjišťování je více. My si vystačíme s tou nejjednodušší, a to takovou, že projdeme všechna čísla od 1 do zadaného N a vyzkoušíme, jestli některé z nich nedělí zadané N. Všechna čísla od 1 do N projdeme v cyklu. Abychom se po skončení cyklu mohli rozhodnout, jestli je dané číslo prvočíslo, nebo nikoliv, potřebujeme vědět, zda ho nějaké číslo dělí. Opět nebudeme vymýšlet nic složitého – počet čísel, která zadané číslo dělí, si spočítáme.

Abychom si ukázali, že cyklus nemusí být jen od 1 do N, tak si provedeme malou optimalizaci. Každé číslo je dělitelné 1 a samo sebou, takže je jasné, že v rozmezí od 1 do N bude zadané číslo N dělit číslo 1 a číslo N. Z tohoto důvodu je nemusíme vůbec zkoušet a rozmezí cyklu může být od 2 do N-1 (pro zvídavější: větší optimalizaci dosáhneme tím, že by cyklus byl od 2 do odmocniny z N, což není ani zdaleka poslední možnost). Pokud nebudeme do počtu dělitelů započítávat 1 a sebe sama, tak prvočíslo bude takové číslo, které bude mít v rozmezí 2 až N-1 celkem nula dělitelů.

Test dělitelnosti provádíme výpočtem, tzv. zbytku po dělení (operace modulo). Pro zápis operace zbytku po dělení se používá příkaz **%** nebo **mod**, např. výpočet zbytku po dělení 3: X = Y % 3 nebo X = Y mod 3. My budeme používat první zápis se znakem procent, který je používanější a úspornější v zápisu (například: 7 % 3 = 1).

Ve výsledném vývojovém diagramu nejprve inicializujeme celkový počet dělitelů (proměnnou *P*) na 0 a načteme testované číslo od uživatele. Dále v cyklu projedeme všechna čísla od 2 do čísla, které je o jedno menší než zadané uživatelem. Těmito čísly z cyklu (index) zkusíme vydělit zadané číslo – zjistíme, jaký je zbytek po dělení. Pokud je zbytek nulový, pak je zadané číslo od uživatele dělitelné indexem a jediné, co provedeme, je přičtení 1 do proměnné *P*, která obsahuje celkový počet dělitelů. Na konci cyklu vyhodnotíme proměnnou *P*. Pokud je nulová, tak nebylo nalezeno žádné číslo, které by zadané dělilo, a proto je výsledkem vypsaný text, že zadané číslo N je prvočíslo. V opačném případě, kdy hodnota P není nulová, existovalo alespoň jedno číslo, které zadané číslo dělilo, a proto se nemůže jednat o prvočíslo.

Následující tabulka obsahuje jednotlivé kroky pro průchod vývojovým diagramem. Vzhledem k opakování v cyklu se zde některé kroky (3, 4, 5) cyklicky opakují. Tabulka obsahuje všechny kroky pro zadané číslo 5. Výsledkem je zjištění, že číslo 5 je prvočíslo.

	P	N	I	popis činnosti
1	0			inicializace proměnné P
2	0	5		načtení testovaného čísla od uživatele
3	0	5	2	vstup do cyklu, začínáme od 2
4	0	5	2	test dělitelnosti 5 % 2 → zbytek po dělení je 1 (-)
5	0	5	2	- větev (žádná operace)
3	0	5	3	další průchod cyklem, hodnota I se zvedá
4	0	5	3	test dělitelnosti 5 % 3 → zbytek po dělení je 2 (-)
5	0	5	3	- větev (žádná operace)
3	0	5	4	další průchod cyklem, hodnota I se zvedá
4	0	5	4	test dělitelnosti 5 % 4 → zbytek po dělení je 1 (-)
5	0	5	4	- větev (žádná operace)
3	0	5	5	hodnota I překročila zadané N-1 → konec cyklu
6	0	5		test hodnoty P==0 → + větev
7	0	5		vypiš: N (5) je prvočíslo

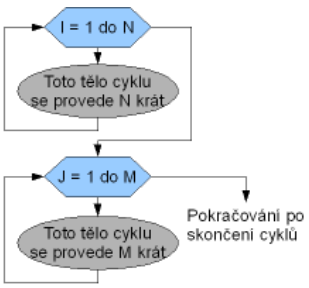
Příště si ukážeme složitější algoritmy s využitím cyklů.

Vývojové diagramy – 9. díl

Autor: Libor Beneš, korektura: Martin Šimeček, 26. 08. 2011
http://programujte.com/clanek/2010030102-vyvojove-diagramy-9-dil/

V tomto díle zakončíme téma cyklů s daným počtem opakování.

V 6. díle jsme si ukázali, že podmínek je možné použít v programu více. Stejné je to samozřejmě i s cykly. Doposud jsme ve vývojovém diagramu použili pouze jeden, ale to jenom z toho důvodu, že jsme jich více nepotřebovali. Kolik cyklů ve vývojovém diagramu (programu) použijeme, záleží pouze na nás, resp. na problému, který řešíme. V dnešním díle si ukážeme několik příkladů, kde využijeme více jak jeden cyklus.

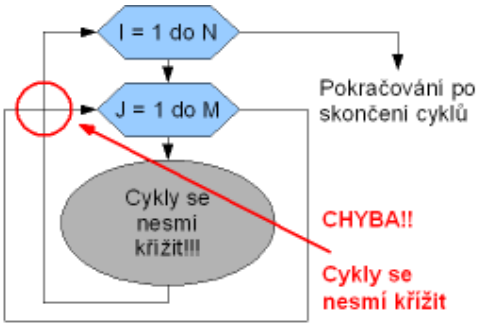


Než se vrhneme na příklady, tak se podíváme na kombinování cyklů. V principu jsou dvě možnosti: buď jsou cykly na sobě nezávislé, nebo je jeden součástí druhého. První možnost je vidět na obrázku vlevo. Cykly jsou na sobě nezávislé – až jeden skončí, tak začne druhý. Druhý je tedy spuštěn až po skončení prvního. Těla cyklů se vykonají nezávisle na sobě podle daného rozmezí – v případě výřezu vývojového diagramu jde o N a M opakování. V obou se mohou zpracovávat různá data (ale i stejná), protože cykly jsou na sobě nezávislé.

Druhou možností je **vnořený cyklus**. Jde o situaci, kdy jeden cyklus je v těle jiného. Tělo vnořeného cyklu se provede N × M-krát, jak je vidět na obrázku. Při vstupu do prvního cyklu je I=1. Následně se vykonává jeho tělo, kde je další cyklus. Při vstupu do něj je J=1 – tento cyklus se provede M-krát, tj. **J** se změní postupně od 1 do M,

ale **I zůstává rovno 1**. Po skončení vnořeného (vnitřního) cyklu se vracíme do vnějšího cyklu, kde se **I mění na 2**. Následně se vykoná jeho tělo, kde máme vnořený cyklus, který se znovu provede, tj. **J** se bude opět měnit od 1 do M (**I je stále 2**) atd. Nakonec se tělo vnořeného cyklu provede N × M-krát. Průchod si můžeme demonstrovat výpisem (odsazení značí průchod vnořeným cyklem):

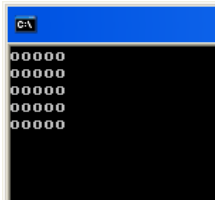
```
(kód před cyklem)
I=1
        J=1  {zde se vykoná tělo vnitřního cyklu}
        J=2  {a znovu}
        ...
        J=M  {celkem M-krát}
I=2
        J=1  {opět se vykonává tělo vnitřního cyklu}
        J=2
        ...
        J=M  {zase celkem M-krát, neboli zatím 2 x M krát}
I=3
        J=1
        J=2
        ...
        J=M
...
        ...
I=N
        J=1
        J=2
        ...
        J=M  {nakonec se tělo vnitřního cyklu provede N x M krát}
(pokračování po skončení cyklu)
```

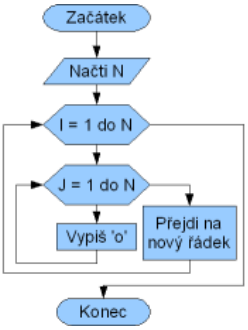


U vnořeného cyklu jsou tedy zpracovávaná data společná. Počet vnoření není nijak omezen. Typický případ pro vnořený cyklus z obrázku je procházení tabulky (matice), kde I označuje řádek a J sloupec. V těle vnořeného cyklu se postupně můžeme dostat k jednotlivým buňkám tabulky.

Zakreslení cyklu tak, jak ho používáme, má i jednu výhodu – jednoduše se na něm pozná chyba, kdy se vnější a vnořený kříží. Vnořený a vnější cyklus se nesmí křížit, protože to znamená chybu v návrhu nebo v jeho zakreslení. Chybně zakreslený vnořený cyklus je vidět na obrázku.

Vykreslení plného čtverce



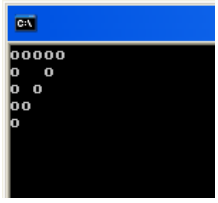


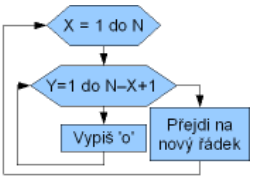
V prvním dnešním příkladu si ukážeme použití vnořeného cyklu při „vykreslení" plného čtverce. Zadání úlohy by bylo následující: vytvořte algoritmus, který na obrazovku vypisováním znaků „o" nakreslí čtverec o zadané délce strany. Vstupem od uživatele tedy bude zadaná délka strany čtverce. Výstupem algoritmu bude „vykreslený" čtverec.

Vykreslený čtverec bude vypadat podobně, jako bychom ho „nakreslili" například v Notepadu. Dejme tomu pro zadanou délku strany 5, tj. 5 řádek, kde by na každém řádku bylo 5 znaků „o" vedle sebe. A jak by vlastně vypadal algoritmus nakreslení takového čtverce 5×5 v Notepadu? Začali bychom na prvním řádku a napsali bychom postupně pět těchto znaků. Následně bychom odřádkovali a napsali dalších pět znaků. Pak ještě tři takové řádky a měli bychom nakresleno.

Když vypisujeme znaky do řádku, tak odpočítáme daný počet. V programu takový přístup vede na cyklus s daným počtem opakování. Stejně tak odpočítáváme řádky, tj. opět se jedná o cyklus s daným počtem opakování. Výsledný algoritmus budou tedy tvořit dva (vnořené) cykly. Vnitřní cyklus bude vypisovat znaky do řádku a vnější cyklus se bude starat o přechod na nový řádek (stejně jako jsme to dělali v Notepadu). Výsledný vývojový diagram je vidět na obrázku.

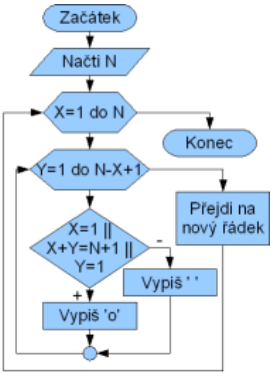
Vykreslení rovnostranného trojúhelníku





Druhý dnešní příklad bude velice podobný. Jeho zadání zní následovně: vytvořte algoritmus, který na obrazovku vypisováním znaků „o" a mezera nakreslí rovnostranný trojúhelník o zadané délce strany. Od uživatele dostaneme zadán jeden rozměr – délku strany trojúhelníku. Algoritmus jako svůj výstup bude mít vykreslený trojúhelník.

Opět si zkusíme takový trojúhelník nakreslit v Notepadu. Řešení úlohy si rozdělíme, nejprve si takto vykreslíme plný trojúhelník. První řádek bude velice podobný jako při kreslení čtverce – vypíšeme všechny znaky „o". Druhý řádek plného trojúhelníku bude mít o jeden znak „o" méně. Třetí řádek bude mít o 2 znaky méně, čtvrtý řádek o 3 znaky atd. Výsledný trojúhelník v textovém zápisu sice vypadá spíše jako rovnoramenný, ale to nám nevadí, důležitý je princip.

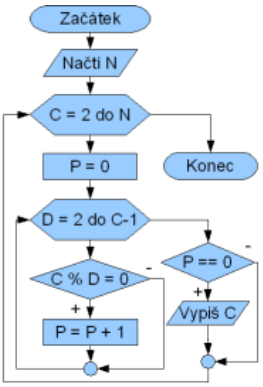


V druhém vnořeném cyklu je vidět závislost na prvním cyklu. Počet vnořených opakování klesá s narůstajícím počtem vnějšího cyklu. Dostáváme tedy další možnost omezení, která je také poměrně častá – počet opakování jednoho cyklu ovlivňuje počet opakování druhého cyklu. Výsledný rozsah opakování je tedy dán rozdílem maximálního počtu, který zadal uživatel (N), a aktuální hodnotou vnějšího cyklu. Jednička se do rozmezí přičítá, protože začínáme od 1, takže tím ji kompenzujeme. Výsledný rozsah je tedy od 1 do N-X+1.

Nyní je potřeba vyřešit vykreslení pouze hran trojúhelníku. Když si trojúhelník v Notepadu upravíme podle zadání, tak vidíme, že znaky jsou pouze v prvním řádku, prvním sloupci a na diagonále. V podmínce to musíme zohlednit. Znaky na prvním řádku znamená, že se budou vypisovat v případě, že X je rovno 1 (X=1). Podobně je to i se sloupcem, tj. Y je rovno 1 (Y=1). Poslední částí podmínky je diagonála. Můžeme vysledovat, že na diagonále je součet pozice řádku a sloupce vždy stejný, a to o jednu větší, než je maximum (N), neboli podmínka bude X+Y=N+1.

Všechny části podmínky zohledňují část vykresleného trojúhelníka. Stačí, aby byla splněná jedna z podmínek (1. řádek nebo 1. sloupec atd.). Spojovací operátor mezi jednotlivými částmi podmínek bude tedy || (nebo). A vůbec nevadí, že pro některé pozice (např. pro pozici [1, 1]) budou splněny dvě najednou. Výsledný vývojový diagram je vidět na obrázku.

Všechna prvočísla do zadaného čísla



Od kreslení se opět vrhneme na matematiku. Minule jsme si vytvořili algoritmus pro zjištění, jestli dané číslo je prvočíslo. Tento příklad si nyní rozšíříme o zjištění všech prvočísel v daném rozsahu. Zadání takového příkladu by znělo: vytvořte algoritmus pro zjištění všech prvočísel od 2 do zadaného čísla. Vstupem od uživatele tedy bude horní mez rozsahu. Výstupem budou vypsaná prvočísla v daném rozmezí.

Funkce bude stejná, jako kdybychom program dle vývojového diagramu z minulého dílu spustili pro čísla 2, 3, 4, 5, …, N. V podstatě celý vývojový diagram z minula bude tělem nově přidaného cyklu. Hledáme prvočísla v rozsahu 2 až zadané N, takže rozmezí vnějšího cyklu bude od 2 do N.

Tělo vnitřního cyklu musí obsahovat nulování počítadla dělitelů (*P*), aby se počítání provádělo pro každé testované číslo zvlášť. Kdyby byla inicializace *P* před vnějším cyklem, tak by po prvním zjištěném čísle, které má více dělitelů (tj. od čísla 4), již žádné další číslo nebylo označeno jako prvočíslo. Proto se musí hodnota *P* nulovat před každým novým průchodem vnitřním cyklem.

Rozsah vnitřního cyklu je stejný jako minule, jen je ovlivněn vnějším cyklem. Horní mez rozsahu je dána aktuální hodnotou vnitřního cyklu. Pro hodnotu 2 se vnitřní cyklus vůbec neprovede (jeho rozmezí je od 2 do 1), pro hodnotu 3 se provede jednou (od 2 do 2), pro hodnotu 4 je rozmezí od 2 do 3 (neboli se provede výpočet zbytku po dělení číslem 2 a číslem 3) atd.

Výsledný vývojový diagram je vidět na obrázku. Složitost vývojových diagramů nám utěšeně roste a jak je vidět na tomto příkladu (a částečně na předchozím), tak často lze úlohu rozdělit na menší části, které se dají řešit samostatně. Více o této problematice se dozvíme v některém z dalších dílů, ale nejprve nás čekají další dva typy cyklů: s podmínkou na začátku a s podmínkou na konci.

Vývojové diagramy – 10. díl

Autor: Libor Beneš, korektura: Zdeněk Lehocký, 02. 09. 2011
http://programujte.com/clanek/2010030103-vyvojove-diagramy-10-dil/

Doposud jsme se zabývali cykly, u nichž byl přesně dán počet opakování. Dnešním dílem začneme cykly, u kterých tento počet předem neznáme.

Cyklus, který má zadaný počet opakování, je samozřejmě velmi často používaný. Rozměry nebo rozsahy pro ně bývají dobře známy (velikost obrázku, velikost tabulky atd.). Existují ale případy, kdy nevíme, kolik opakování budeme muset projít, abychom se dobrali cíle.

Řekněme, že děláme program pro pračku. V jedné fázi programu budeme čekat na to, až se ohřeje voda na určitou teplotu, a toto čekání budeme provádět v cyklu. Možná někoho napadne, že bychom mohli čekat daný počet vteřin (daný čas). To bohužel nejde, protože jednak nevíme, jak studená voda bude na začátku, a také nevíme, jak rychle se bude voda ohřívat, takže se nedá určit, jak dlouhou dobu to bude trvat. Jediný opravdový konec čekacího cyklu může nastat ve chvíli, kdy teplota vody dosáhne požadované nebo vyšší hodnoty.

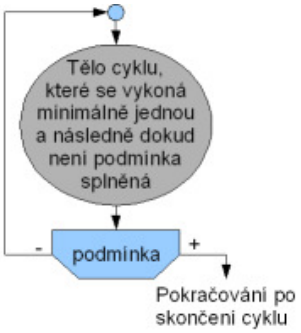
Možná vám to připadá jako podmínka, protože ona to podmínka je. Cykly, které si dnes ukážeme, se řídí takovou podmínkou a platí pro ni stejná pravidla jako pro „normální" podmínku.

Cykly, které se řídí podmínkou, jsou dva, a to s podmínkou na začátku a s podmínkou na konci. Obě varianty mají společné, ale i odlišné vlastnosti, které je předurčují k různým oblastem použití. Oproti cyklu s daným počtem opakování nemají index, který by automaticky měnil svou hodnotu.

Fragment vývojového diagramu pro první typ vidíte na obrázku. Jak je již patrné z názvu „cyklus s podmínku na začátku", tak vyhodnocovací podmínka je před samotným tělem. Cyklus má tyto vlastnosti:

- nejprve se vyhodnocuje podmínka a až pak se provádí tělo,
- tělo se vykonává, dokud je podmínka splněna,
- tělo se nemusí vykonat ani jednou (pokud není podmínka splněna hned při prvním vstupu),
- počet opakování je omezen pouze podmínkou a může jich být 0 až nekonečno.

Druhý typ (s podmínkou na konci) má dvě varianty. V některých programovacích jazycích se tělo vykonává, dokud není podmínka splněna (např. Pascal), a v jiných, dokud splněna je (např. C). Vzhledem k tomu, že pro výuku programování je vhodný více Pascal než C, tak budeme používat první variantu.



Fragment vývojového diagramu pro tento cyklus vidíte na obrázku, a protože existují dvě varianty, tak budeme důsledně označovat větve, tj. kdy cyklus končí a kdy pokračuje (a to i pro cyklus s podmínkou na začátku). Z názvu „cyklus s podmínkou na konci" jasně plyne, že vyhodnocovací podmínka je až za tělem. Cyklus má tyto vlastnosti:

- nejprve se provede tělo a až pak se vyhodnocuje podmínka,
- tělo se vykonává, dokud není podmínka splněna,
- tělo se vykoná minimálně jednou,
- počet opakování je omezen pouze podmínkou a může jich být v rozmezí 1 až nekonečno.

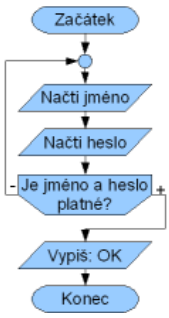
Možná někoho napadne: a co v případě, že podmínka je neustále splněná (pro cyklus s podmínkou na začátku) nebo nebude nikdy splněná (pro cyklus s podmínkou na konci)? Odpověď je jednoduchá – jedná se o nekonečný cyklus. Jedná se o speciální typ algoritmu, který se používá pouze ve výjimečných případech (např. program v jednočipovém procesoru pro nějaké řízení, který běží neustále). Jinak musí platit, že výsledek algoritmu musí být znám v konečném počtu kroků.

Konečnost – výsledek algoritmu musí být znám v konečném počtu kroků.

Než přejdeme na praktické ukázky, tak je potřeba zmínit, že jakákoliv úloha na cykly je řešitelná jakýmkoliv z těchto cyklů. Vždy je možné úlohu řešit jak s podmínkou na začátku, tak s podmínkou na konci. Dokonce mohou tyto cykly nahradit i předchozí typ, tj. s daným počtem opakování – jeden takový příklad si ukážeme. A i další příklady budeme nejprve řešit pro oba typy cyklů.

Možná se ptáte, proč jsou dva typy, když jsou v podstatě zaměnitelné. Odpověď už tu byla naznačena – každý cyklus je vhodný na jiný typ úloh. Někdy je výhodnější použít podmínku na začátku, někdy na konci a v některých případech je to jedno. Uvedené příklady by to měly demonstrovat.

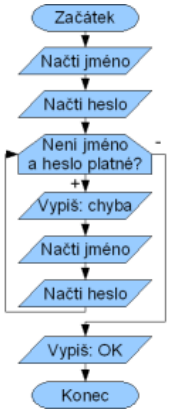
Login



Jediný dnešní příklad bude všem důvěrně známý, protože přihlašování (login) do nějakého systému (e-mail, Facebook, IM a mnoho dalších) je na denním pořádku (ač se často děje již automaticky). Zadání úlohy by mohlo být následující: vytvořte algoritmus pro přihlášení uživatele do systému. Přihlašování se děje zadáním jména a hesla. Vstupem algoritmu tedy bude jméno a heslo, které zadá uživatel. Výstupem bude hlášení, že jsou nebo nejsou zadané údaje správně a že již případně došlo k úspěšnému přihlášení.

Jak jsem slíbil, úlohu budeme řešit oběma typy cyklů. Nejprve je nutné zadat přihlašovací údaje a pak je teprve možné rozhodnout, jestli jsou správně a povolit další přístup. V tomto případě má výhodu cyklus s podmínkou na konci, neboť se

nejprve vykoná tělo cyklu, kde uživatel zadá jméno a heslo, a pak dojde k vyhodnocení. Výsledný algoritmus je na obrázku. Cyklus s podmínkou na konci je pro tento typ úloh ten vhodnější.



Naproti tomu cyklus s podmínkou na začátku je na tento typ úloh méně vhodný. Před prvním vyhodnocením je nutné zadat přihlašovací údaje, ale následně v těle cyklu také (pro případ, že by je uživatel nezadal správně). V algoritmu tak dochází k opakování části diagramu (programu) a výsledný vývojový diagram není již tak přehledný. Volbou nevhodného typu cyklu si zbytečně přidělváme práci a algoritmus se stává složitějším.

Všimněte si také rozdílnosti podmínek. Cyklus s podmínkou na začátku končí, pokud není podmínka splněna, takže se musíme ptát, jestli není jméno a heslo platné. V případě, že je podmínka splněna (tedy nejsou platné), tak je potřeba, aby je uživatel zadal znovu. V případě, že není splněna (neboli údaje jsou platné), tak se uživatel úspěšně přihlásil. U druhého cyklu je to vyhodnocení podmínky obráceně, takže se ptáme tak, jak bychom očekávali.

Jedinou výhodou cyklu s podmínkou na začátku je to, že se u něj přirozeně rozlišuje první zadání a následná zadání. To umožňuje vypsat hlášení o tom, že se přihlášení nezdařilo. U cyklu s podmínkou na konci bychom tohoto dosáhli přidáním podmínky na počet pokusů, které bychom si museli počítat. V případě, že by se nejednalo o první pokus, tak by se vypsalo hlášení o chybném přihlášení.

Možná to teď vypadá jako velká výhoda, ale pokud bychom dělali algoritmus pro login nejenom pro ukázkové účely, ale pro praktické použití, tak bychom v něm počítadlo pokusů stejně měli. Při nějakém počtu neúspěšných přihlášení se další přihlášení znemožní nebo se zařízení zablokuje (např. mobil po třech špatně zadaných PINech). Komplexnější vývojový diagram pro zadávání PINu s maximálním počtem opakování (i s podmínkou na hlášení) si ukážeme příště.

To je pro tentokráte vše. V příštím díle budeme s cykly pokračovat. Ukážeme si na jednom příkladu řešení pomocí všech tří cyklů a také si ukážeme příklad, ve kterém bude výhodnější použít cyklus s podmínkou na začátku.

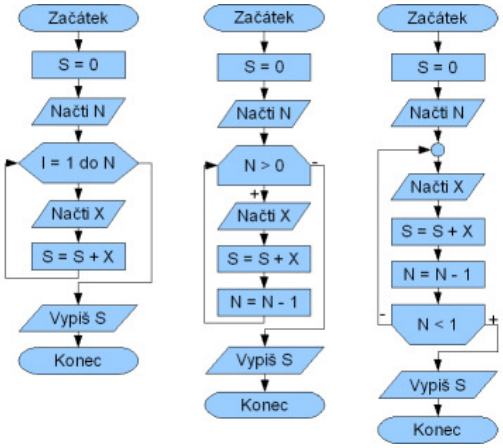
Vývojové diagramy – 11. díl

Autor: Libor Beneš, korektura: Zdeněk Lehocký, 09. 09. 2011
http://programujte.com/clanek/2010030104-vyvojove-diagramy-11-dil/

V dnešním díle si ukážeme příklad, který vyřešíme pomocí všech tří typů cyklů. Dále si ukážeme typ úlohy, pro kterou je vhodnější použití cyklu s podmínkou na začátku.
A nezapomene na příklad se zadáním PINu do mobilu.

Součet N čísel

V prvním dnešním příkladu si ukážeme, že úlohu lze řešit všemi typy cyklů. Podobnou úlohu už jsme řešili. Zadání té aktuální by mohlo znít: vytvořte algoritmus pro součet N zadaných čísel, kdy nejprve zadáme kolik čísel budeme sčítat a následně tato čísla postupně načteme a provedeme jejich součet. Na vstupu od uživatele dostaneme nejprve počet čísel (*N*) a následně uživatel zadá N různých čísel, která budeme sčítat. Výsledkem algoritmu bude výpis tohoto součtu.



Řešení pomocí cyklu s daným počtem opakování se nabízí a stále je pro tento typ úloh nejvhodnější. Rozmezí cyklu bude od 1 do zadaného N. V těle se načítají čísla od uživatele (*X*) a provádí se jejich součet do *S*. Proměnnou *S* nesmíme zapomenout před vstupem do cyklu inicializovat na 0. Po jeho skončení máme v *S* výsledný součet, který vypíšeme. Vývojový diagram je vidět na obrázku v jeho levé části.

S	N	I	X	popis činnosti
0				inicializace proměnné S
0	2			načtení N
0	2	1		vstup do cyklu, začínáme od 1
0	2	1	4	načtení první hodnoty do X
4	2	1	4	přičtení X do celkového součtu S
4	2	2		další průchod cyklem (I se inkrementuje)
4	2	2	5	načtení hodnoty do X
9	2	2	5	přičtení X do celkového součtu S
9	2	3		ukončení cyklu, hodnota I přesáhla rozmezí dané N
9	2			vypsání výsledného součtu S

Použití cyklu s podmínkou na začátku nebo na konci není sice to nejlepší řešení, ale je možné. Základ bude stejný, jen potřebujeme počítat počet zadaných čísel, abychom jich opět zadali pouze N. V proměnné *N* máme celkový počet, takže můžeme jít podobnou cestou jako u cyklu s daným počtem opakování, tj. do další proměnné bychom si ukládali počet již zadaných čísel. Podmínka by kontrolovala, jestli počet načtených čísel se rovná *N*.

Vzhledem k tomu, že hodnotu proměnné *N* zadanou uživatelem již nikde nevyužíváme, tak ji můžeme „znehodnotit". Použijeme princip stříhání metru, kde se za každý den čekání odstřihne jeden dílek. My budeme za každé zadané číslo uživatelem snižovat hodnotu *N* o 1.

Vývojový diagram opět začíná inicializací *S* na 0 a zadáním hodnoty N. Následuje vstup do cyklu přes jeho podmínku. Tělo se opakuje, dokud je podmínka splněna, tj. dokud je co „stříhat". V našem případě dokud jsme nezadali všech N čísel, neboli dokud je hodnota v *N* větší než 0.

Tělo cyklu je téměř stejné. K zadávání a sčítání jsme přidali zmenšování hodnoty *N* o 1 (stříhání metru). Cyklus skončí ve chvíli, kdy hodnota *N* bude 0 (nebo menší) – již není co stříhat. Algoritmus bude funkční i v případě, že uživatel zadá do *N* nulové nebo záporné číslo.

S	N	X	popis činnosti
0			inicializace proměnné S
0	2		načtení N
0	2		vstup do cyklu, testování podmínky N > 0 - splněno
0	2	4	načtení první hodnoty do X
4	2	4	přičtení X do celkového součtu S
4	1	4	zmenšení hodnoty N o 1
4	1		další průchod cyklem – test N > 0 - splněno
4	1	5	načtení hodnoty do X
9	1	5	přičtení X do celkového součtu S
9	0	5	zmenšení hodnoty N o 1
9	0		další průchod cyklem – test N > 0 - nesplněno
9	0		vypsání výsledného součtu S

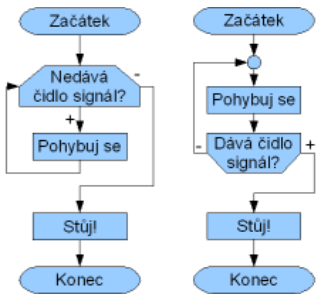
Řešení pomocí cyklu s podmínkou na konci je nejméně vhodné. Rozdíl proti řešení s podmínkou na začátku je hlavně v podmínce – cyklus pokračuje v případě, že není podmínka splněná (dělej, dokud není *N* menší než 1). Jinak je algoritmus v podstatě stejný. Problém tohoto řešení je v tom, že nebude dobře fungovat, pokud uživatel zadá *N* nulové nebo záporné. I v takovém případě se provede tělo cyklu, takže uživatel bude nucen zadat jedno číslo a výsledný součet tak nemusí být 0.

Vstup od uživatele (N) bychom v tomto případě museli extra ošetřit. Zde v podstatě o nic nejde, ale už na něm je vidět to, co bylo řečeno na začátku – pro určité typy úloh jsou některé typy cyklů vhodnější a některé vhodné méně nebo zcela nevhodné. Jak uvidíme na dalším příkladu, tak existují i úlohy, kde se můžeme volbou špatného typu cyklu dostat do problémů.

S	N	X	popis činnosti
0			inicializace proměnné S
0	2		načtení N
0	2	4	vstup do cyklu, načtení první hodnoty do X
4	2	4	přičtení X do celkového součtu S
4	1	4	zmenšení hodnoty N o 1
4	1	4	test N < 1- nesplněno – cyklus pokračuje
4	1	5	načtení hodnoty do X
9	1	5	přičtení X do celkového součtu S
9	0	5	zmenšení hodnoty N o 1
9	0	5	test N < 1- splněno – cyklus končí
9	0		vypsání výsledného součtu S

Pohyb vozíku

Na této úloze si ukážeme, že je nutné zamyslet se nad výběrem cyklu a jeho podmínky. Máme vozík, který se pohybuje po koleji (má jeden stupeň volnosti). Na koncích koleje jsou senzory, které nám řeknou, že vozík dosáhl konce a dál nemůže. Naším úkolem je vytvořit algoritmus pro automatický pohyb po koleji v jednom směru. Úloha je velmi zjednodušená a půjde pouze o demonstraci vhodnosti volby typu cyklu a podmínky.



Úlohu vyřešíme oběma typy cyklů s podmínkou. Začneme tím vhodnějším, a to s podmínkou na začátku. Tělo tohoto cyklu se vykonává, dokud je podmínka splněná, takže se bude vykonávat (pohybovat vozíkem), než nám čidlo dá vědět, že jsme na konci. Jak vidíte, tak vývojový diagram je velice jednoduchý. Testujeme hodnotu z čidla, a pokud ještě nejsme na konci, tak se pohybujeme. Ve chvíli, kdy senzor „zabere", tak zastavíme pohyb a ukončíme program.

Stejně jako v předchozím případě (se záporným číslem) je toto řešení odolné i na mezní případ, a to ten, že vozík stojí na senzoru. Na verzi s cyklem na konci si ukážeme slibované problémy.

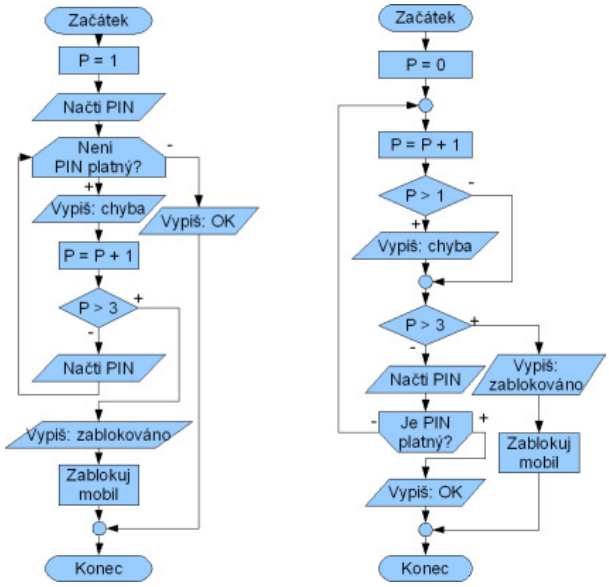
Vývojový diagram je opět jednoduchý. Podmínka je samozřejmě opačná než v předchozím případě. Když jsme např. uprostřed kolejnice, tak i s tímto typem cyklu dosáhneme konce. V čem je tedy problém?

Problém je v krajním případě, tj. pokud vozík stojí na senzoru. V takovém případě by se s tímto typem cyklu dal vozík i tak do pohybu (nejprve se provádí tělo cyklu – pohyb). A pokud by čidlo dávalo stop signál pouze na té jedné pozici, tak by se již vozík nezastavil a vykolejil by. Drobná „chyba", která se nemusí za celou dobu používání projevit, může mít v případě jejího „objevení" fatální následky.

Na obrázku vidíte, že cykly jsou opět velmi podobné. Jediné, v čem se liší, je podmínka, kterou se řídí.

PIN

Poslední dnešní příklad je slibované zadávání PINu do mobilu. Tuto činnost dělá valná většina z vás několikrát do měsíce, takže je všeobecně známá. Máme tedy za úkol vytvořit algoritmus pro zadávání PINu do mobilu s tím, že je možné maximálně 3× po sobě zadat špatný PIN a pak se mobil zablokuje. V případě správného zadání PINu se mobil aktivuje.



Na vstupu od uživatele tedy bude PIN, který následně otestujeme. Výsledkem algoritmu bude buď aktivní mobil, nebo zablokovaný mobil. Úlohu si vyřešíme opět pomocí obou typů cyklů s podmínkou, a to s úplným ošetřením vstupu od uživatele i možných chybových stavů (v rámci zadávání).

Začneme cyklem s podmínkou na začátku. Abychom měli co vyhodnocovat v podmínce, musíme PIN zadat před ní. Jde vlastně o první pokus, takže počáteční hodnotu počítadla pokusů (*P*) můžeme nastavit na 1. Po vyhodnocení jsou 2 možnosti: buď je PIN neplatný a cyklus pokračuje dále, nebo je PIN platný a je možné aktivovat mobil a skončit.

V případě, že PIN není platný (a taková musí být i podmínka), je o této skutečnosti nutno informovat uživatele. Vypíšeme, že PIN není platný a přejdeme na další pokus zadávání PINu. Ještě před zadáním si musíme ověřit, že se nejedná o čtvrtý a další pokus, protože v takovém případě je nutné zadávání ukončit a zablokovat mobil. Na obrázku vývojového diagramu vidíte jedno z možných uspořádání. Mohli bychom bloky uspořádat i jinak, např. počítání pokusů dát až za zadání PINu, ale v takovém případě by podmínka musela být P > 2. Ve výsledku dostaneme tu samou funkčnost.

To je v podstatě vše. Ve výsledném algoritmu je jedna nová věc, cyklus má vlastně dvě možnosti ukončení. Jedna možnost je standardní, a to při nesplnění podmínky cyklu. Druhá možnost ukončení je od vložené podmínky, při jejím splnění se cyklus také ukončí. Toto je poměrně běžná praxe, kdy cyklus končí za nějakých (optimálních) podmínek a přes podmínky se vkládá ukončení cyklu například při nějak chybě nebo „neočekávané" situaci. A také se takového ukončení využívá u „nekonečných" cyklů, tj. cyklů, které by bez další dodatečné možnosti ukončení běžely neustále.

Řešení pomocí cyklu s podmínkou na konci je pro tuto úlohu lepší varianta, ale když porovnáte vývojové diagramy, tak zjistíte, že jsou v podstatě stejné. Velký rozdíl je, a to už jsme vyzdvihli v některém z minulých dílů, že se PIN zadává na jednom místě. Naproti tomu musíme rozlišit v těle cyklu, kdy se jedná o první zadávání a kdy o následné, abychom případně mohli vypsat hlášení uživateli, že PIN není správně.

Stejně jako v řešení s podmínkou na začátku lze bloky v těle cyklu poskládat více způsoby. Důležité je před zadáváním zjistit, jestli se má vypsat hlášení nebo jestli už by se nejednalo o 4. pokus (P > 3). Pokud bychom umístili počítání za zadávání PINu, pak bychom opět museli upravit obě podmínky (na P > 0 a P > 2).

Vývojový diagram také obsahuje 2 možnosti ukončení (stejnou možnost lze využít i v cyklu s daným počtem opakování). Výsledný vývojový diagram si můžete prohlédnout na obrázku a případně zkontrolovat jeho funkci tabulkami hodnot pro různé možnosti (vlastně jsou čtyři: správně zadaný PIN na první, druhou a na třetí možnost a 3× nesprávně žádný PIN).

Tím dnešní díl zakončíme, příště se podíváme na složitější příklady, kde využijeme nejenom více cyklů, ale hlavně jejich kombinace.