

Algoritmizace

Jiří Rybička

Výstup projektu IP MENDELU č. 8.1.17/2020
Studijní materiály pro inovované předměty – Vytvoření studijních materiálů pro předmět
Algoritmizace

Obsah

1	Úvod	7
1.1	Jak studovat z tohoto textu	8
2	Algoritmus	9
2.1	Pojem a vlastnosti algoritmu	9
2.2	Vlastnosti algoritmu	9
2.3	Způsoby vyjádření algoritmu	10
2.4	Vývojové diagramy	11
2.5	Programovací jazyk	14
2.6	Programovací paradigmata	17
3	Programovací jazyk C++	21
3.1	Charakter jazyka	21
3.2	Lexikální jednotky	21
3.3	První program	27
3.4	Hlavičkové soubory	28
3.5	Deklarace proměnných a datové typy	28
3.6	Výraz	30
3.7	Příkaz přiřazení	31
3.8	Konverze a přetypování	32
3.9	Vstupy a výstupy	34
4	Základní algoritmické obraty	37
4.1	Větvění	37
4.2	Cyklus	42
4.3	Převody cyklů	46
4.4	Poškození průběhu cyklu	47
4.5	Nepodmíněný skok	47
5	Elementární algoritmy	50

5.1	Způsoby získání posloupnosti dat	50
5.2	Extrémy	53
5.3	Součty, součiny	55
5.4	Počty, výběry	56
5.5	Metoda skládání elementárních algoritmů	58
5.6	Princip shora dolů	60
5.7	Trasování	61
6	Časová a prostorová složitost algoritmů	64
6.1	Složitost algoritmů	64
6.2	Způsob zápisu	64
6.3	Varianty složitosti	64
6.4	Stanovení složitosti	65
6.5	Porovnání složitostí programů	65
6.6	Přehled typických složitostí	66
7	Operace s jednotlivými bity	69
7.1	Operace s bity	69
7.2	Posuvy (rotace)	70
7.3	Masky	71
7.4	Zjišťování hodnot význačných bitů	72
8	Uživatelské datové typy	73
8.1	Definice identifikátoru datového typu	73
8.2	Uživatelský výčtový typ	74
8.3	Definice identifikátoru typu výčet	75
8.4	Datový typ void	75
8.5	Definice konstant	76
9	Podprogramy	77
9.1	Druhy a definice podprogramů	77
9.2	Volání podprogramu	79
9.3	Skok tam, skok sem	80

9.4	Globální/lokální	82
9.5	Funkce <code>main</code> a komunikace s OS	83
9.6	Další možnosti komunikace s operačním systémem	84
9.7	Rekurze	85
10	Strukturované datové typy – pole	88
10.1	Koncept pole	88
10.2	Algoritmy využívající pole	90
10.3	Znaková pole	94
10.4	Pole jako parametr podprogramu	96
10.5	Vícerozměrná pole	99
11	Záznamy	104
11.1	Datový typ záznam	104
11.2	Datový typ variantní záznam	107
12	Dynamické datové struktury	109
12.1	Statické, nebo dynamické?	109
12.2	Princip dynamické proměnné	110
12.3	Ukazatele v C++	111
12.4	Pole v C++	113
12.5	Dynamické datové struktury	114
13	Soubory	123
13.1	Standardní soubory	123
13.2	Uživatelské soubory	124
14	Algoritmy vyhledávání	130
14.1	Sekvenční vyhledávání	130
14.2	Sekvenční vyhledávání se zarážkou	131
14.3	Sekvenční vyhledávání v uspořádaném poli	131
14.4	Vyhledávání půlením intervalu	132
14.5	Stromové vyhledávání	132

14.6 Vyhledávání indexací	132
14.7 Vyhledávání hašováním	135
15 Algoritmy řazení	140
15.1 Vlastnosti řadicích metod	140
15.2 Přípravné definice	141
15.3 Kvadratické metody	142
15.4 Lineárně logaritmické metody	144
15.5 Lineární metody	148
16 Preprocesor, moduly	149
16.1 Preprocesor	149
16.2 Programové moduly	151
17 Struktury s obecnými daty	155
17.1 Datový typ podprogram	155
17.2 Modifikace operací vyžadujících znalost dat	156
17.3 Použití obecné datové struktury	160
18 Dodatky	161
18.1 Tabulka priorit operátorů jazyka C++	161
18.2 Vybraná klíčová slova jazyka C++	162
18.3 Možnosti formátování vstupu a výstupu	162
18.4 Zkrácené vyhodnocování logických výrazů	164
19 Literatura	167

1 Úvod

Učební text Algoritmizace je určen pro stejnojmenný předmět vyučovaný typicky v počátečních semestrech oborů spojených s informatikou. Samotný pojem **algoritmizace** představuje jeden ze základních kamenů informatiky a prakticky vždy je nezbytné jej právě v počátcích výuky informatiky projít. Jak budeme pojem algoritmizace chápat? Přikloníme se k pravděpodobně nejfrekventovanějšímu pojetí, tj. jde o *vytváření postupu řešení zadané úlohy*.

V této souvislosti je vhodné poznamenat, že takto stanovený pojem algoritmizace má zcela abstraktní obsah – zmíněnou „zadanou úlohou“ může být cokoliv a o způsobu zadání se v uvedené definici nic neříká.

Často se algoritmizace spojuje s počítači a informatikou – tam algoritmizaci můžeme upřesnit jako vytváření postupů řešení zadaných úloh *pomocí počítače*. V tomto případě se ovšem často algoritmizace zaměňuje s jiným pojmem, a to **programování**. V některých textech je programování chápáno dokonce jako synonymum pojmu algoritmizace. Vzhledem k tomu, že v případě tohoto textu půjde o specializaci postupů pro řešení úloh pomocí počítače, budeme s pojmem programování také často pracovat, upřesněme si tedy jeho význam:

Programování je vyjádření (zakódování) algoritmu v podobě zpracovatelné počítačem.

Nyní je jasnější, proč pojem algoritmizace a programování může splývat v jedno: i když algoritmizace je tvůrčí proces hledání řešení nějaké úlohy, zatímco programování je (mnohdy už jen mechanické) zapsání nalezeného řešení v určité podobě, můžeme obě tyto činnosti provozovat najednou, neodděleně. Pak při psaní programu (tedy programování) zároveň hledáme řešení (tedy algoritmizujeme úlohu). Setkáváme se tudíž s učebními texty, jejichž název už uvedené splynutí přímo deklaruje, například „Algoritmizace a programování v jazyce Pascal“ (Krček a Kreml, 1993).

Tento učební text si také klade za cíl oba zmíněné aspekty – přestože primárním výstupem je vždy nalezený algoritmus, budeme zároveň hledat i optimální způsob jeho realizace, tedy budeme programovat.

Snahy zásadním způsobem oddělit algoritmizaci od programování byly vždy provázeny poněkud obtížným hledáním vhodného nástroje k vyjádření algoritmů nezávislým na jakémkoliv operačním prostředí, programovacím jazyce či používané technologii. Je samozřejmé, že pak lze v takovém „čistém“ prostředí realizovat „čistý“ proces algoritmizace a získané algoritmy mají zcela univerzální charakter. Na druhé straně je však třeba mít na mysli, že následné programování také vyžaduje do určité míry tvůrčí zapojení, nejde jen o čistě mechanické zakódování obecně zapsaného algoritmu. Uvažujeme o optimalizaci datových i řídicích struktur a pohlížíme na konkrétní nástroj tak, abychom úlohu dobře vyřešili nejen v abstraktní, ale také ve zcela konkrétní rovině. Různé realizační alternativy jednoho a téhož algoritmu mohou vést ke zcela rozdílnému výslednému chování a ukazuje se tím, že čistě abstraktní algoritmus a jeho konkrétní realizace mohou být dvě dost rozdílné věci.

V této chvíli ovšem narážíme na bezbřehý problém – jaké implementační prostředí zvolit pro optimální realizaci algoritmů. Na toto téma bylo již publikováno obrovské množství prací a určitě by bylo možné dále polemizovat, která varianta je „nejlepší“. To zmíněné obrovské množství prací ale spíše prokazuje, že *není nejlepší řešení*, že vždy se při jakékoliv volbě něčeho vzdáváme a současně něco

jiného získáváme. V této souvislosti bychom tedy mohli (možná poněkud unáhleně) konstatovat, že je jedno, jakou cestou se vydáme. Jedno to samozřejmě není. Zvolili jsme ale implementační prostředí podle zcela jiného kritéria (vědomi si současně i mnoha nevýhod), a to podle rozšířenosti použití a pokud možno bez ovlivnění různými módními trendy. Dlouhodobě je v žebříčcích využití programovacích jazyků na prvních místech skupina jazyků odvozených od jazyka C. Proto volba padla na jazyk C++, a to i s ohledem na kontext jiných předmětů vyučovaných ve stejném oboru.

Učební text bude jednak prezentovat hledání algoritmů, ale také bude řešit jejich realizaci. Jde v něm tedy o algoritmizaci i programování v jazyce C++. V některých kapitolách půjde především o algoritmy, v jiných naopak zase spíše o jazykové a implementační záležitosti.

1.1 Jak studovat z tohoto textu

Charakter textu vzhledem ke stanoveným cílům rozhodně není beletristický. Vyžaduje nejen prosté čtení, ale také uvažování nad různými souvislostmi a přemýšlení o principech a technikách. Je silně doporučeno při prvotním studiu nepřeskakovat žádné pasáže v textu, neboť výklad na sebe poměrně těsně navazuje.

2 Algoritmus

2.1 Pojem a vlastnosti algoritmu

Na úvod si vymezíme tento základní pojem:

Algoritmus je přesný návod nebo postup řešení zadané úlohy.

Slovo algoritmus bylo postupně různými zkomoleními získáno ze jména významného perského matematika žijícího v první polovině 9. století (cca 780–840), kterým byl Abū Abd Allāh Muhammad ibn Mūsā al-Chwārizmī.

V dnešní době se algoritmus často spojuje s řešením úloh pomocí počítačů. Jde však o úplně obecný pojem, s jehož projevy se setkáváme v běžné denní praxi. Jakékoliv řešení každého problému lze chápat jako aplikaci algoritmu, ať se například jedná o jednoduché cestování městskou hromadnou dopravou z jednoho místa na druhé nebo o několikaleté komplexní řešení výstavby rodinného domu. Ještě než se plně ponoříme do realizace algoritmů pomocí počítače, budeme se zabývat i dalšími možnostmi.

2.2 Vlastnosti algoritmu

Algoritmus má čtyři základní vlastnosti (v různých zdrojích bývají uvedeny vlastnosti s různými názvy, někdy je vlastností více apod.):

1. Jednoznačnost (determinističnost) – v každém místě algoritmu je jednoznačně určeno, co se bude dít dál, je vyloučen prvek náhody.¹
2. Konečnost – algoritmus je zapsán konečným počtem prvků a pro jakákoliv vstupní data skončí v konečném počtu kroků; vždy dospěje k nějakému výsledku (i když je to třeba jen zpráva o chybných vstupních datech).
3. Opakovatelnost – pro stejné vstupní hodnoty dospěje algoritmus ke stejným výsledkům.
4. Hromadnost (obecnost) – algoritmus popisuje celou třídu podobných úloh (nepopisuje pouze jednu nebo několik málo variant s konkrétními vstupy a výstupy.)

Kromě těchto vlastností můžeme hovořit například o srozumitelnosti (algoritmus musí být zapsán tak, aby mu rozuměl vykonávající subjekt), dále o kvalitativních vlastnostech (přehlednost, časová a prostorová složitost), úrovni podrobnosti atd.

¹ To platí i o algoritmech pro generování tzv. náhodných čísel. Prvek náhody je zde zastoupen vstupními daty, která mohou vzejít z hodnoty nějaké fyzikální veličiny, například okamžitého času, teploty v místnosti atd. Bude-li tato vstupní hodnota stejná, bude algoritmus generovat naprosto stejnou řadu „náhodných“ čísel. Tato čísla jsou tedy přesněji označována jako pseudonáhodná.

2.3 Způsoby vyjádření algoritmu

Chceme-li algoritmus, tedy nalezený způsob řešení úlohy, někomu sdělit, musíme jej nějak zapsat, vyjádřit. Základní čtyři způsoby vyjádření algoritmu jsou:

1. *Slovně* – řešení úlohy je popsáno volnou slovní formou. Jedním z nejrozšířenějších příkladů slovního vyjádření algoritmu je recept v kuchařce. Základní výhodou slovního vyjádření je naprostá univerzalita. Slovní vyjádření však může vést k nejednoznačnosti, protože velká většina slov nebo slovních spojení v přirozeném jazyce nemá naprosto přesný význam.

Vzpomeňme si například na velmi známou scénku Felixe Holzmann, kde se svěřuje svému známému (idnes.cz, 2002): „Já nerad vařím podle kuchařky, psali tam – krájejte tři dny starý housky – a já je nekrájel ještě ani den a už jsem jich měl plnou vanu ...“

2. *Matematicky* – řešení úlohy je nejčastěji popsáno matematickým vztahem mezi vstupními hodnotami a požadovanými výsledky. Hodí se pro určité typy úloh, kde mezi vstupy a výstupy existuje relativně snadno vyjádřitelný vztah, často se jedná o numerické algoritmy nebo algoritmy s logickými funkcemi. Výhodou matematického zápisu je jeho jednoznačnost a možnost vyjádřit i nejrůznější význačné situace.

Příklad: Vstupní hodnotou je číslo x , výstupní hodnotou číslo y a má platit vztah:

$$y = \sqrt{x}, \quad x \in \mathbb{R}, x \geq 0.$$

Vidíme zde přesné vymezení, o jaká vstupní čísla se může jednat, v případě hodnot nesplňujících vstupní podmínky není výsledek definován.

3. *Graficky* – jednotlivé kroky řešení jsou znázorněny pomocí grafických prvků doplněných potřebnými popisy (obvykle v přirozeném jazyce, ale mohou být použity různé přesnější způsoby vyjádření). Jedním z nejpoužívanějších způsobů grafického vyjádření algoritmu je **vývojový diagram**. Budeme se mu podrobněji věnovat v následující sekci.

Grafické vyjádření algoritmu je obvykle velmi názorné, zejména pro jednodušší případy. Nepřesnost vyjádření je dána především nepřesností doplňujících popisů v přirozeném jazyce.

Kromě zmíněných vývojových diagramů se také používá **kopenogram**, jehož zásadní výhodou je strukturovanost záznamu a snadnější převoditelnost na konstrukce programovacího jazyka (podrobnější informace lze například nalézt na stránce Kopenogramy, 2020). Dalším nástrojem může být **struktoqram** neboli Nassiův-Shneidermanův diagram určený zejména pro vyjádření strukturovaného přístupu k řešení úlohy.

Všechny grafické způsoby vyjádření algoritmu jsou určeny pro člověka – slouží proto především jako dorozumívací nástroj mezi lidmi. A to se všemi vlastnostmi jiných lidských dorozumívacích nástrojů.

4. *Počítačově (například programovacím jazykem)* – jde o jedinou formu přímo zpracovatelnou počítačem, a tedy prakticky realizovatelnou. Bývá konečným cílem tvorby algoritmu, jak už bylo řečeno dříve. Základní vlastností tohoto vyjádření algoritmu je naprostá jednoznačnost –

všechny použité výrazové prostředky mají zcela přesně stanovený význam, nemůže tedy dojít k žádnému nečekanému efektu. Programovacímu jazyku bude věnována značná část tohoto textu, nebudeme proto v tomto místě uvádět další podrobnosti.

V této souvislosti je vhodné připomenout, že často vídaným postupem je vyjádření algoritmu například vývojovým diagramem, jehož přepisem se pak získává zdrojový program. Z vlastností obou forem vyjádření algoritmu ovšem vyplývá, že jde o cestu neefektivní – vyjadřovací schopnosti jsou rozdílné a je-li jako první v řadě vývojový diagram, pak se všechny jeho nevýhody nutně projevují v cílovém zápisu pomocí programovacího jazyka. Zhruba v 70. letech 20. století, kdy se vývojové diagramy masivně používaly, tento problém prakticky neexistoval – programovací jazyky tehdejší praxe (zejména FORTRAN a BASIC) se ve svých vyjadřovacích schopnostech od možností vývojových diagramů příliš nelišily. To ovšem v dnešní době, kdy se používají jazyky disponující komplexními řídicími i datovými konstrukcemi, vůbec neplatí.

Postup programování, kdy úlohu vyjádříme pomocí vývojového diagramu a ten následně přepisujeme do programovacího jazyka, tedy používat nebudeme, i když pro ukázkou uvedeme nějaké příklady.

Nyní se budeme podrobněji věnovat dvěma zmíněným způsobům vyjádření, a to vývojovým diagramům a programovacím jazykům.

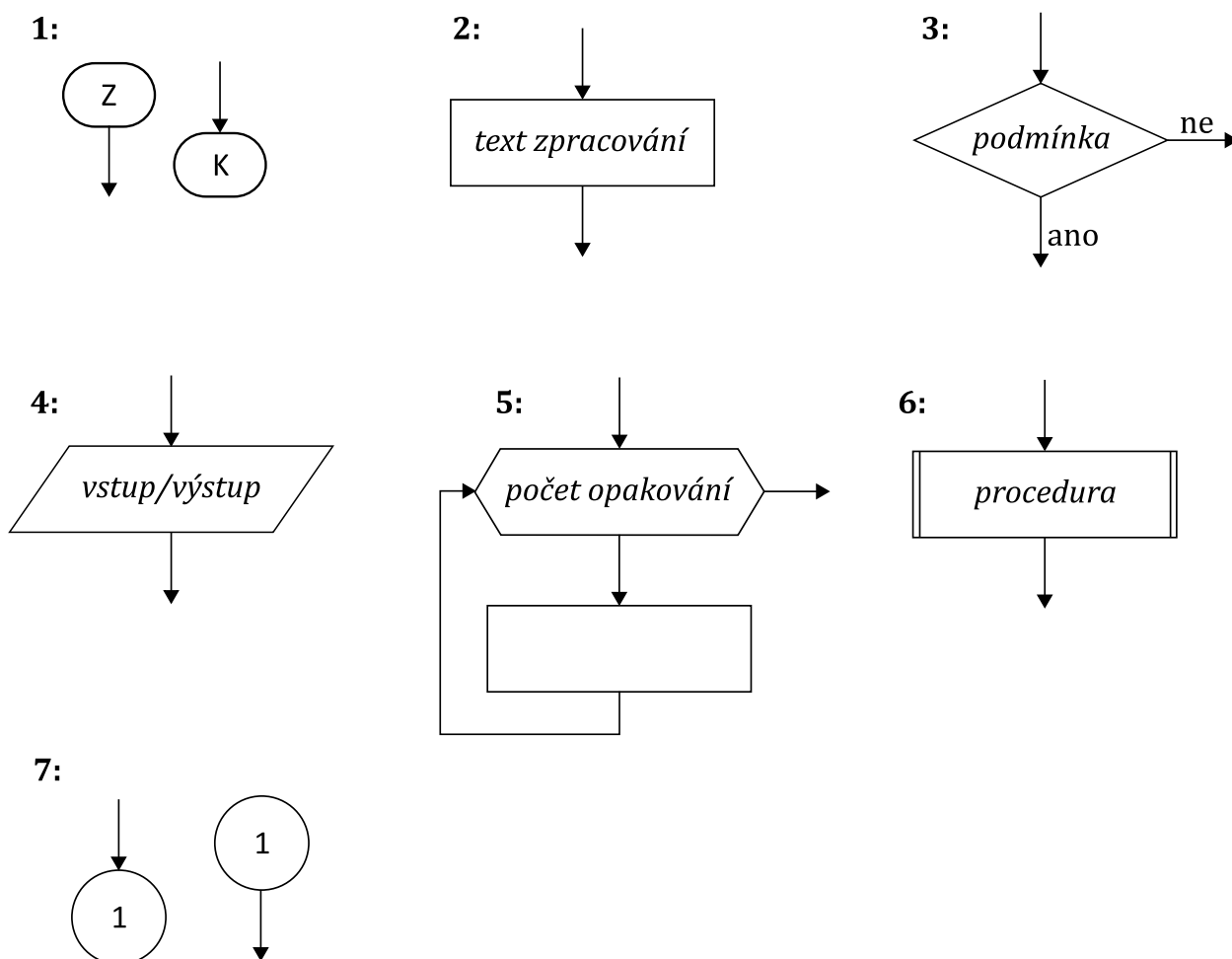
2.4 Vývojové diagramy

Vývojový diagram znázorňuje jednotlivé kroky algoritmu pomocí smluvených značek, které mají jednoznačně určený význam. K těmto značkám lze doplnit slovní popisy v běžném (živém) jazyce, v jazyce s přesněji definovanými výrazovými prostředky nebo v libovolném programovacím jazyce.

Značky jsou propojeny orientovanými spojnicemi. Orientaci spojnic lze vyznačit šipkou, pokud však šipka chybí, předpokládá se, že svislá spojnice má orientaci shora dolů a vodorovná zleva doprava.

Značky používané ve vývojových diagramech zahrnují (obr. 1):

- *Mezní značky* – určují začátek a konec algoritmu.
- *Zpracování* – prvek algoritmu, v němž dochází k činnosti s daty (k výpočtu nebo jiné manipulaci).
- *Rozhodování* – prvek algoritmu, v němž na základě hodnoty nějaké podmínky dochází k větvení postupu.
- *Vstup a výstup* – prvek, v němž se získávají vstupní hodnoty nebo vypisují výsledky.
- *Opakování* – prvek naznačující opakování určité větve diagramu.



Obrázek 1: Značky vývojových diagramů: 1 – mezní značky, 2 – zpracování, 3 – rozhodování, 4 – vstup a výstup, 5 – opakování, 6 – procedura, 7 – spojky

- *Procedura* – prvek představující ucelenou část algoritmu znázorněnou jiným diagramem nebo řešenou jinými prostředky.
- *Spojky* – značky umožňující navázání spojnice do jiné části diagramu.

Na příkladu jednoho algoritmu ukážeme popsané vlastnosti a způsoby vyjádření: Chceme řešit úlohu, jak se lze v Brně dopravit městskou hromadnou dopravou z hlavního nádraží do kampusu Mendelovy univerzity.

Slovní vyjádření můžeme získat například dotazem na nějakého rodilého Brňana v prostoru před hlavním nádražím. Můžeme dostat odpověď:

No, šalinó devitkó a jed' na Zemědělskó.

Jako zpracovávající procesor musíme vědět:

- „šalina“ znamená tramvaj.

- „devítka“ je číslo tramvajové linky.
- „Zemědělská“ je název zastávky, na které máme vystoupit.
- Tramvaj musí jet ve směru Lesná.
- Musíme mít platnou jízdenku.
- Tramvaj jede z prvního nástupiště.
- Tramvaje nejezdí v nočních hodinách od 23.20 do 4.50 – v tom případě musíme použít noční autobusovou linku jedoucí z jiného místa.

Z příkladu je vidět, že schopnosti zpracovávajícího procesoru mají na tvar algoritmu zcela zásadní vliv. Lze konstatovat, že čím složitější operace je daný procesor schopen provádět, tím kratší může vyjádření algoritmu být.

Mezi schopnostmi člověka (jako zpracovávajícího procesoru) a schopností počítače je propastný rozdíl – a to lze chápat jako celý problém algoritmizace: postup vyjádřený pro člověka se musí značně obohatit, aby jej „pochopil“ i stroj pracující pouze s relativně elementárními prvky.

Předpokládejme nyní, že přepravu bude potřebovat člověk, který nikdy nebyl ve větším městě. Slovní vyjádření zdaleka nebude tak stručné jako v předchozím případě. Můžeme sepsat pokyny v následujícím tvaru:

1. Opatří si jízdenku. Je potřeba jízdenka na 45 minut. Zvol si jednu z následujících možností: automat na mince mobilní telefon u řidiče po nástupu do vozidla
2. Jaká je denní doba? Pokud je časové rozmezí od 5 do 22.30 hodin, pokračuj následujícím krokem. Pokud je noční doba od 22.30 do 5, pokračuj krokem 11.
3. Přesun na nástupiště: U hlavního nádraží lze nastupovat na 5 místech, je potřebné vyhledat nástupiště 1 (odpovídající směr tramvaje).
4. Vyčkej příjezdu tramvaje linky 9.
5. Zkontroluj, zda cílová stanice tramvaje je „Lesná, Čertova rokle“. Pokud ne, může jít o vůz jedoucí do vozovny, čekej tedy na další.
6. Nastup do tramvaje.
7. Pokud ještě nemáš jízdenku, kup si ji u řidiče.
8. Pokud máš papírovou jízdenku, označ ji v odpovídajícím strojku ve vozidle, aby byla platná. Jízdenka v mobilu se neoznačuje, je platná od okamžiku potvrzovací SMS.
9. Po dobu jízdy tramvaje sleduj hlášení o zastávkách. Dokud není ohlášena zastávka Zemědělská, čekej.

10. Po zastavení na zastávce Zemědělská vystup z vozidla. Jdi na krok 18.
11. Vyčkej příjezdu autobusů do prostoru nástupišť před hlavním nádražím. (To se děje vždy před časem rozjezdu, což je v celou hodinu a někdy i v půlhodinu.)
12. Vyhledej autobus linky 92 s cílovou stanicí Soběšice.
13. Nastup do autobusu.
14. Pokud ještě nemáš jízdenku, kup si ji u řidiče.
15. Pokud máš papírovou jízdenku, označ ji v odpovídajícím strojku ve vozidle.
16. Po dobu jízdy autobusu sleduj hlášení o zastávkách. Dokud není ohlášena zastávka Zemědělská, čekej.
17. Po zastavení na zastávce Zemědělská vystup z vozidla.
18. Jsi u cíle.

Zjednodušené grafické vyjádření slovně popsaného algoritmu znázorňuje obr. 2.

Další příklady:

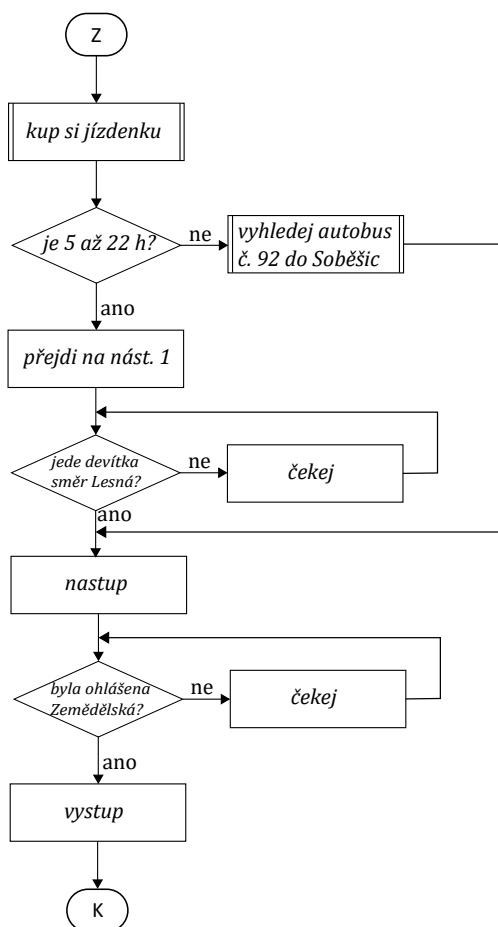
Zkuste slovním vyjádřením a vývojovým diagramem zapsat následující algoritmy:

- Uvaření N vajíček natvrdo.
- Nákup potravin v internetovém obchodě.
- Postavení rovné zdi z porobetonových bloků.
- Zjištění stupně kontaminace vody chemickými látkami použitými pro hnojení.
- Oprava vadné žárovky levé zadní směrovky v automobilu značky Škoda Fabia.

2.5 Programovací jazyk

Programovací jazyk přesně definuje schopnosti zpracovávajícího procesoru – vše, co v tomto jazyce zapíšeme, je určité daný procesor schopen provést. To je zásadní rozdíl oproti slovnímu nebo grafickému vyjádření. Například slovní pokyn „kup si jízdenku u řidiče“ vůbec nepočítá s nejrůznějšími komplikacemi – nemáme drobné, řidič již jízdenky vyprodal, nastoupíme do přívěsného vozu, kde není řidič atd. To se v žádném případě nemůže stát při zápisu jakéhokoliv příkazu programovacího jazyka. Je-li předepsáno třeba volání funkce pro výpočet hodnoty logaritmu zadaného čísla, máme jistotu, že výsledná hodnota bude vždy získána.

Algoritmus zapsaný programovacím jazykem, budeme nazývat **program**.



Obrázek 2: Příklad vývojového diagramu – přeprava pomocí MHD

2.5.1 Vlastnosti programovacích jazyků

Programovacích jazyků je nepřeberné množství a mají nejrůznější vlastnosti. Některé jsou obecně použitelné, jiné mohou být specializovány na úlohy určitého typu. Vždy však platí, že zápis je zcela jednoznačný a lze zcela objektivně prokázat, jakou akci a jaký efekt tímto zápisem docílíme – a to i bez použití počítače.

Definice každého programovacího jazyka (podobně jako každého přirozeného jazyka) zahrnuje dva aspekty:

- **syntax** – způsob zápisu jednotlivých prvků,
- **sémantika** – význam jednotlivých prvků.

Na rozdíl od přirozeného jazyka je však obojí definováno zcela jednoznačně a navíc množství prvků je *konečné*. Nemůže se tedy stát, že se někde objeví konstrukce, která není nikde definována.

2.5.2 Provedení programu

Procesor počítače je schopen provádět pouze velmi jednoduché povely zvané **instrukce**. Instrukce však provádí velmi rychle, je schopen provést někdy až miliardy instrukcí za vteřinu. Seznam všech možných instrukcí, tzv. **instrukční sada** (nebo také instrukční repertoár) je pro každý typ procesoru jiný. Napíšeme-li program v nějakém programovacím jazyce, je nezbytné tento program přeložit na posloupnost instrukcí, které jsou definovány pro daný typ procesoru. Překlad provádějí specializované programy zvané **překladače** (viz dále).

Proč existuje více typů procesorů s různými instrukčními sadami? Vyvinout a vyrobit počítačový procesor nedokáže kdekdo, zabývá se tím několik firem na světě, mezi nimiž existuje konkurenční soupeření. Jde také o různá technická řešení s různými vlastnostmi, do toho musíme započítat procesory pro různá speciální zařízení apod.

Proč existuje více programovacích jazyků? Odpověď na tuto otázku má více částí: jednak vyvinout a realizovat programovací jazyk narozdíl od technického řešení procesoru dokáže prakticky kdekdo, jednak každý programovací jazyk sleduje určitý primární účel a dalším významným faktorem jsou různé módní trendy.

Máme tedy na jedné straně více různých procesorů a na druhé straně více různých programovacích jazyků. Jaký to může mít smysl?

- Jedním programovacím jazykem lze vytvářet programy pro více různých typů procesorů. To znamená, že práce věnovaná vývoji nějakého programu je uplatnitelná pro více technických zařízení.
- Jeden algoritmus lze napsat ve více programovacích jazycích pro stejný procesor. To naopak znamená, že jedno technické zařízení lze programovat více způsoby a lze vybrat takový způsob (jazyk), který je pro danou úlohu optimální.

Obě skutečnosti vždy ovšem platí jen do určité míry. Například program napsaný v určitém programovacím jazyce nelze vždy a za všech okolností použít bez jakékoliv změny pro různé procesory, také nelze říct, že stejný algoritmus napsaný v různých programovacích jazycích povede po překladu ke stejné posloupnosti instrukcí pro daný jeden procesor, a tedy k identickému chování, rovněž zápis stejného algoritmu v různých jazycích je různě obtížný.

Pro účely algoritmizace se v tomto textu nebudeme zabývat žádnými typy procesorů, pouze se podíváme na širší kontext programovacích jazyků. Jak vyplývá z uvedených skutečností, je při zápisu algoritmu mnohdy jedno, jaký programovací jazyk použijeme, výběr tohoto jazyka v praxi závisí v drtivé většině na jiných faktorech než na algoritmu (bohužel závisí v některých případech na tom, že programátor jiný jazyk neovládá nebo se firma rozhodla pro určitý nástroj kvůli licenčním podmínkám atd.).

2.6 Programovací paradigmatata

Pojmem **programovací paradigma**² se obvykle myslí základní programovací styl a přístup k řešení problému. V různých informačních zdrojích se můžeme setkat s následujícími paradigmaty:

- **Imperativní**, též **procedurální** – popisuje úlohu jako posloupnost jednotlivých kroků určujících přesný postup. Základní metodou imperativního paradigmatu je procedurální programování, proto jsou tyto dva pojmy obvykle chápány jako synonyma. Imperativnímu paradigmatu vyhovuje většina programovacích jazyků a pracuje se jím v drtivé většině případů.
- **Funkcionální** – úloha je vyjádřena pomocí funkcí, jejichž vyhodnocováním vzniká výsledek. Funkce vypočítávají své výsledky čistě na základě argumentů, nemají žádné vedlejší účinky. Příkladem funkcionálního programovacího jazyka je Haskell, mezi funkcionální jazyky se často počítá i Lisp (LISt Processing), i když oproti čisté teorii umožňuje modifikovat již definované proměnné, čímž pracuje poněkud hybridním způsobem. Jako příklad funkcionálního programu lze (s trochou nadsázky) chápat také tabulku v tabulkovém procesoru, kde výsledek výpočtu je získán vyčíslováním funkcí v jednotlivých buňkách.
- **Logické** – úloha je vyjádřena bází axiomů a odvozovacích pravidel, výpočet probíhá jako řešení otázky, které se dokazuje odvozovacím mechanismem. Představitelem logického paradigmatu je Prolog (PROgramming in LOGic)
- Někdy se jako zvláštní paradigma uvádí i **objektové** (objektově orientované) – program je psán jako sada různě spolupracujících objektů, které jsou vždy souhrnem vlastností (datových složek) a funkčních komponent (metod). Metody objektu jako jediné mají možnost manipulovat s datovými složkami. Těla metod jsou však fragmenty imperativního kódu, tato dvě paradigmatata tedy nejsou v přímém rozporu. Zásadní výhodou objektů je polymorfismus, pokud jej však aplikace nevyužívá, je objektové řešení oproti procedurálnímu podstatně méně efektivní.

Některé programovací jazyky jsou označovány jako multiparadigmatické – umožňují řešení různými paradigmaty. Řada jazyků původně procedurálních později dostala „objektovou“ nadstavbu a umožňuje pracovat v čisté procedurálním nebo objektovém paradigmatu (C++), záleží pak na programátorovi, pro jakou strategii řešení se v dané aplikaci rozhodne a jak čistě ji bude realizovat.

2.6.1 Překladače a implementace programovacích jazyků

K tomu, abychom mohli ovládat procesor, je potřebné mu předložit instrukce, jak už bylo uvedeno. Tyto **strojové instrukce** jsou pro každý typ procesoru specifické a v dnešní době se prakticky nikdy přímo nezapisují. Jediným způsobem získání strojových instrukcí je automatizovaný překlad z nějakého programovacího jazyka. Budeme-li se zabývat imperativními jazyky, lze mezi nimi rozpoznávat následující úrovně:

² **Paradigma** (z řeckého π α ρ ά δ ε ι γ μ α) = vzor, příklad, model, vzorec chování.

- **Jazyk symbolických instrukcí (JSI)** – programovací jazyk, který je nejbližší instrukcím daného procesoru. Instrukce jsou však zapisovány „lidským“ způsobem, tedy nikoliv jako čísla (binární), ale zkratky názvů, například ADD = sčítání, CALL = volání podprogramu, JMP = programový skok. Místo konkrétních číselně vyjádřených adres operandů v operační paměti lze použít symbolické názvy, proto se jazyk někdy nazývá **jazyk symbolických adres (JSA)**. Na této úrovni programování lze využít veškeré specifické vlastnosti procesoru a minimalizovat operační čas provedení úlohy, používá se často k vytváření časově kritických aplikací nebo pro řízení specifických zařízení (ovladače periférií apod.).
- **Vyšší programovací jazyk** – základním prvkem nejsou instrukce, ale vyšší celky – příkazy. Jazyk je vybaven datovými typy a umožňuje zapisovat úlohu způsobem, který je podstatně bližší člověku než stroji. Vyšší programovací jazyky (někdy zvané též jazyky 3. úrovně, je-li strojový kód 1. úroveň a JSI druhá úroveň) představují dlouhodobě největší podíl jazyků používaných pro programování.

Jazyky deklarativní (realizující částečně nebo úplně jiné paradigma než imperativní) jsou někdy v této souvislosti zařazovány do kategorie jazyků 4. úrovně (4GL = fourth generation languages), sem patří například hodně rozšířený jazyk SQL pro řízení databází.

2.6.2 Výběr programovacích jazyků

Pro určitou ilustraci oblasti programovacích jazyků se můžeme podívat na některé konkrétní reprezentanty a velmi zjednodušeně popsat jejich vlastnosti.

Fortran Zkratka slov „FORmula TRANslator“, tj. překladač vzorců. Jazyk vznikl původně jako nástroj pro realizaci numerických algoritmů, což souvisí s tehdy převažujícími aplikacemi (výpočty). Prvkem jazyka již je **příkaz**, vyšší forma povelu pro provádějící procesor nezávislá na typu procesoru a přibližující se lidskému vyjádření. Každý příkaz se zapisoval na samostatný řádek, program měl řádkovou organizaci a na řádku byla pevně dána pozice, kam se mohly zapisovat jednotlivé části příkazu. Vznik a doba používání tohoto jazyka přibližně odpovídá vzniku a největšímu rozšíření vývojových diagramů. Mezi prvky vývojového diagramu a příkazy jazyka Fortran existuje přímá vazba – přepis vývojového diagramu do programu (i opačně) je relativně snadný.

Basic Zkratka slov Beginners All purposes Symbolic Instruction Code. Programovací jazyk určený pro malé (osobní) počítače (mikropočítače). V době, kdy malé počítače postavené z několika málo součástek kolem jednoho procesoru měly nepatrné výkony a velmi omezenou paměť, byl Basic téměř jedinou formou komunikace s těmito stroji. Pro různé typy mikropočítačů existovaly různé klony tohoto jazyka. Jazyk měl podobné vlastnosti jako Fortran, ale s podstatně menšími možnostmi. Také šlo o řádkově orientovanou organizaci programu, takže vazba mezi vývojovým diagramem a programem byla rovněž přímá jako u jazyka Fortran. Z této doby například pochází dnes již zcela překonaná metoda výuky algoritmizace formou kreslení vývojových diagramů, které se pak přepisovaly do podoby programu.

Algol Zkratka slov „ALGO^rithmic Language“ – jeden z prvních jazyků, u nichž primárně šlo o vyjadřování algoritmů, nikoliv o rozhraní k technickému zařízení. Jazyk je koncipován jako volně psaný text, tedy nemá řádkovou organizaci a program lze psát tak, aby byl přehledný pro člověka, příkazy jsou komplexnější, existují tzv. klíčová slova. Není zde již žádná vazba na prvky vývojových diagramů, přepis grafické podoby algoritmu do programu se tak stává těžko řešitelným a zbytečným problémem. Název Algorithmic Language začíná přímo podporovat vytváření algoritmu zápisem programu, nikoliv kreslením grafické podoby.

Pascal Jazyk určený pro výuku algoritmizace. Vychází principiálně z jazyka Algol, jeho konstrukce jsou však „vyčištěny“ tak, aby zejména začátečníkům nepůsobily žádné překážky v pochopení zapsaného algoritmu a umožňovaly průzračně jasné vyjádření základní myšlenky. Tento jazyk se používal nejen k samotnému programování, ale také jako jednoznačný nástroj k vyjádření algoritmů pro člověka.

C/C++ Jazyk C vzniká jako „lidštější“ podoba jazyka JSI a ztrácí také vazbu na konkrétní typ procesoru. Doménou jazyka C bylo programování systémových součástí (příkazů operačního systému) Unix. Jazyk je koncipován rovněž s volnou organizací zápisu, jsou v něm definovány různorodé nástroje usnadňující vytváření programů rozmanitého zaměření a akcent je položen zejména na stručnost zápisu (mnohdy velmi na úkor přehlednosti a zřetelnosti). Dočkal se značného rozšíření, lze konstatovat, že dlouhodobě představuje jeden z nejpoužívanějších programovacích nástrojů. Kromě toho řada prvků byla převzata do dalších jazyků, takže lze hovořit spíše o zobecněné myšlence než o konkrétním jazyce. Jazyk C++ přidává ještě objektové nástroje a řadu dalších změn. Pravděpodobně se v oblasti informatiky nelze obejít bez znalosti tohoto jazyka. To je i jedním z podstatných důvodů, proč byl tento jazyk vybrán jako nástroj v tomto textu.

Java Jazyk vytvořený se záměrem psát programy, které budou použitelné na úplně všech možných typech procesorů, tedy programy přenositelné zcela kamkoliv na úrovni zdrojového zápisu. Vzniká tzv. **technologie Java**, složená ze dvou základních komponent: samotného jazyka Java (mnoho konstrukcí společných s jazykem C) a p-kódu, který vznikne překladem programu v jazyce Java. Tento p-kód (určitá obdoba zobecněných strojových instrukcí) se pak pomocí konkrétního interpretu dá spustit na jakémkoliv procesoru. Samozřejmě interpretace p-kódu je principiálně podstatně pomalejší než přímé spuštění strojového kódu, takže aplikace Java byly vždy velmi náročné na výkon technického vybavení, ovšem s rozvojem techniky se tato zásadní nevýhoda poněkud stírá a původní záměr téměř neomezené přenositelnosti se dostává stále více do popředí.

Python Jazyk vzniklý začátkem 90. let 20. století, jehož prudký rozmach začal kolem roku 2018. Shrnuje vhodné prvky z jazyků dřívějších (čisté konstrukce vhodné pro výuku, možnost využití objektového přístupu). Jeho syntax je myšlenkově postavena opět na jazyku C.

2.6.3 Typy překladačů

Překladače mohou principiálně pracovat dvěma způsoby:

1. **Generativní** – překladač přečte a zanalyzuje zdrojový text zápisu algoritmu a vygeneruje posloupnost strojových instrukcí. Tato posloupnost se následně spustí, čímž dojde k výpočtu a získání výsledků.

V tomto režimu dosahujeme maximální rychlosti zpracování. Překladač při překladu kontroluje veškeré syntaktické i sémantické zákonitosti a pokud možno se snaží reportovat všechny nalezené nesrovnalosti. Výsledný strojový kód je spustitelný. Je velmi výhodné, pokud hotový přeložený kód vícenásobně používáme.

2. **Interpretační** – překladač čte část zdrojového textu, analyzuje ji a ihned provádí (interpretuje). Často je tento režim spojen s jazykem orientovaným na jednoznačně oddělitelné celky – řádky, po odeslání jednoho řádku se objeví výsledek spuštění (interpretace). Jde o interaktivní proces, který je vhodný například pro okamžité jednorázové získání určitých výsledků. Interpretace je principiálně pomalejší – opakované příkazy se opakovaně analyzují a při interpretaci se často neuchovává kontext již vypočtených výsledků.

Určitý programovací jazyk je obvykle spojen pouze s jednou z uvedených forem překladu. Výjimečně existují programovací jazyky použitelné jak v generačním, tak i v interpretačním režimu. Jazyk C++ je jazykem typicky generovaným. Z toho přímo vyplývá, že přeložené programy se používají tzv. **dávkově**, často i automatizovaně a bez účasti uživatele sedícího u klávesnice a sledujícího výstupy. Budeme často předpokládat, že programy jsou přirozenou součástí operačního systému a komunikují přes příkazový řádek. Programy mohou být spouštěny automatizovaně a mohou si brát připravená data. Výsledky pak mohou vkládat do připravených souborů nebo je posílat jiným programům.

Nebudeme (až na vzácné výjimky) v programech řešit *komunikaci s uživatelem*, značně si tím zjednodušíme situaci a můžeme se soustředit pouze na hlavní algoritmus a na jazykové prvky, které k jeho realizaci potřebujeme.

3 Programovací jazyk C++

V této kapitole se seznámíme s vyjadřovacím nástrojem, jímž budeme zapisovat vytvářené algoritmy.

3.1 Charakter jazyka

Jazyk C, na jehož základě je postaven i jazyk C++, vznikl začátkem 70. let a jeho tvůrci, Ken Thompson a Dennis Ritchie, sledovali základní cíl – náhradu programování v jazyce symbolických instrukcí za nástroj bližší člověku, ale zároveň se schopnostmi nízkourovňových operací. Tím byl vytvořen jazyk podstatně čitelnější pro člověka, ale také snáze přenositelný, tj. méně závislý na technickém vybavení daného stroje. Jazyk C se vyznačuje minimalistickými konstrukcemi (nebyl nikdy určen pro širokou veřejnost), což vyžaduje poměrně detailní znalost chování každého prvku v různých kontextech.

Jazyk C++ vyšel z jazyka C v počátku jako rozšíření o objektové vlastnosti. Jeho autor, Bjarne Stroustrup, ze začátku udržoval konstrukci jazyka C++ tak, aby tvořil čistou nadmnožinu jazyka C. To se však později ukázalo jako neudržitelné a bránilo to dalšímu vývoji. Proto dnes jazyk C++ podporuje strukturovaný přístup, objektový přístup a tzv. generický přístup (šablonování) a překladače tohoto jazyka již nejsou vždy schopny přeložit současně i program zapsaný v čistém jazyce C.

V jazyce C++ nalezneme 18 základních (primitivních) datových typů, z nichž lze vytvářet typy další (uživatelské, strukturované). Jde o 10 typů pro zobrazení celých čísel, tři typy pro desetinná čísla, 4 typy pro znakové hodnoty a jeden typ pro logické hodnoty.

Řídící struktury vycházejí z jazyka C, byly přidány objekty, šablony a mnoho dalších prvků.

3.2 Lexikální jednotky

Chceme-li zapsat program v nějakém jazyce (a v této chvíli můžeme mít na mysli i jazyk přirozený, nejen programovací), potřebujeme znát vždy dvě oblasti:

- **syntax** – způsob zápisu, gramatiku, tedy jakými prvky lze vyjádřit, co potřebujeme;
- **sémantiku** – význam (obsah) zapsaných konstrukcí.

Jak už bylo zmíněno, programovací jazyk je oproti přirozenému jazyku zcela jednoznačný a mnohdy i uzavřený (konečný). Syntaktická pravidla jsou pevně dána, stejně jako „slovní zásoba“ – tzv. **lexikální jednotky**. Existují sice programovací jazyky, které umožňují vytvářet nové lexikální jednotky nebo měnit stávající, to ovšem v případě jazyka C++ neplatí, tam je množina stavebních kamenů, z nichž se skládají programy, pevná.

Lexikální jednotky se skládají ve vyšší celky – **syntaktické konstrukce**. Je to podobné jako v přirozeném jazyku: ze slov se skládají věty. Větná stavba (syntax) má v přirozených jazycích svá pravidla, podobně v programovacím jazyku je pevně stanoveno, v jakém pořadí se mohou „slova“ zapisovat.

Ke každé syntaktické konstrukci pak náleží (jak už bylo zmíněno) její jednoznačný význam – sémantika.

3.2.1 Syntaktické diagramy

Přesný způsob zápisu jakéhokoliv prvku musí být také jednoznačně definován. Lze k tomu použít mimo jiné i tzv. **syntaktický diagram** (zvaný též **graf syntaxe**). Je to velmi přehledná a často používaná grafická forma vyjádření možné posloupnosti prvků v zápisu syntaktické konstrukce i lexikální jednotky.

Syntaktický diagram se skládá z uzlů a orientovaných spojníc. Uzly mohou být provedeny jako kolečka nebo ovály – uvnitř nich je uveden prvek, který se doslovně zapisuje, a obdélníky – uvnitř nich je odkaz na jiný syntaktický diagram. Lze tedy nakreslit i složité konstrukce složené z celého systému diagramů. Diagram se používá tak, že z počátečního bodu se prochází orientovanými spojnicemi přes jednotlivé uzly a do výsledku se zapisují obsahy těchto uzlů. Tím dostáváme správný zápis daného prvku (lexikální jednotky nebo syntaktické konstrukce).

Popisujeme-li syntaktickým diagramem lexikální jednotku, pak mezi jednotlivými elementy nesmí být žádné oddělovače (mezery, konce řádků apod.). Naopak vyšší celky – syntaktické konstrukce – mají mezi jednotlivými elementy povoleny libovolné počty tzv. bílých znaků (mezer, konců řádků, tabelátorů), které umožňují přehledně zapisovat zdrojový text programu.

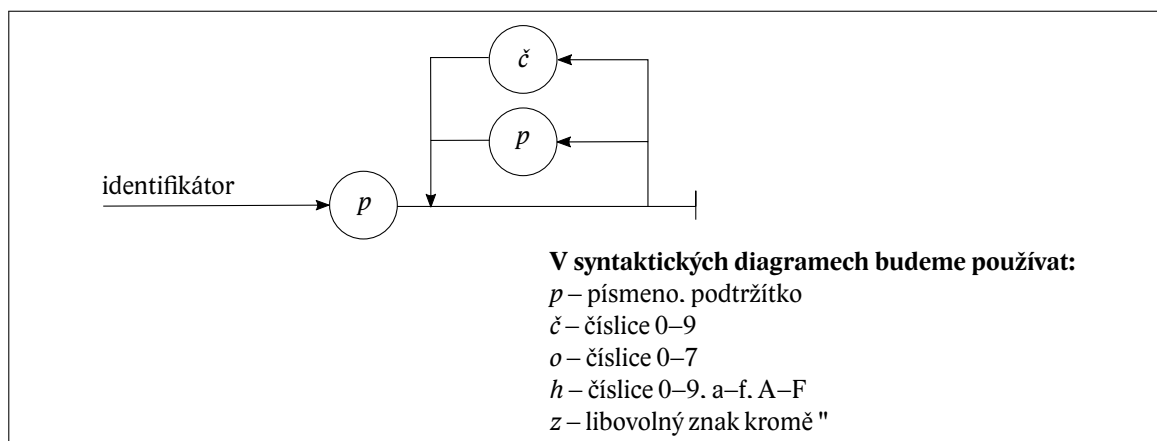
Použití syntaktických diagramů si ukážeme u lexikálních jednotek a budeme je uvádět i u syntaktických konstrukcí.

3.2.2 Vybrané lexikální jednotky

Uvedeme nyní lexikální jednotky, které budeme pro zápis programů hned ze začátku potřebovat. K nim později budeme přidávat další. V této souvislosti je potřebné ještě zmínit jednu důležitou skutečnost: jazyk C++ je **citlivý na sadu** (angl. „case sensitive“), to znamená, že mezi velkým a malým písmenem je vnímán rozdíl. Napíšeme-li například `pom` a jinde `Pom` a ještě jinde `POM`, jde o tři různé zápisy. V některých jiných jazycích na sadě nezáleží, můžeme tedy totéž vyjádřit velkými nebo malými písmeny podle libosti.

- **identifikátor** – název nějakého prvku (proměnné, funkce, konstanty atd.). Skládá se z písmen anglické abecedy, číslic a znaku podtržení, přičemž nesmí začínat číslicí. Některé identifikátory jsou už předdefinovány, ale identifikátor si typicky vytváří programátor pro všechny své nově definované prvky.

Syntaktický diagram definující způsob zápisu identifikátoru:



Příklady identifikátorů:

<i>správné zápisy</i>	<i>chybné zápisy</i>
Pocet	20dmocnina
_chyba3	Počet
dloUhaProMenNa	dva#tri
hodnota_vyrazu_po_secteni	Bude-li
A365B	d'Artagnan

- **klíčové slovo** – identifikátor, jehož význam je pevně určen a nelze jej změnit. Jde o sadu slov označující primitivní datové typy, prvky řídicích a datových struktur a některé operátory. Klíčová slova se pro přehlednost v tisku vyznačují **tučně** (při ručním psaní se obvykle podtrhávají).

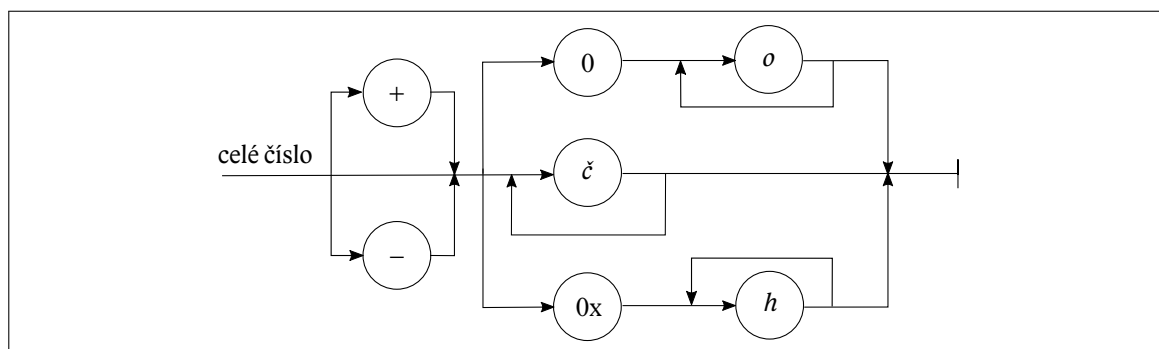
Výběr některých klíčových slov:

int float using namespace if while do else return

- **celočíselný literál** – pojem **literál** znamená přímý zápis hodnoty, celočíselný literál je pak celé číslo zapsané svou hodnotou. Skládá se v nejjednodušším případě z číslic 0 až 9 a může začínat znaménkem plus nebo minus. Pokud znaménko není uvedeno nebo je uvedeno plus, jde o nezáporné číslo. Číslo se vždy zapisuje v jednom celku, tisíce se neoddělují ani tečkou ani mezerou nebo jiným oddělovačem. Číslo zapisovaná tímto způsobem jsou v desítkové soustavě. Příklady: **-381 178000 +10**

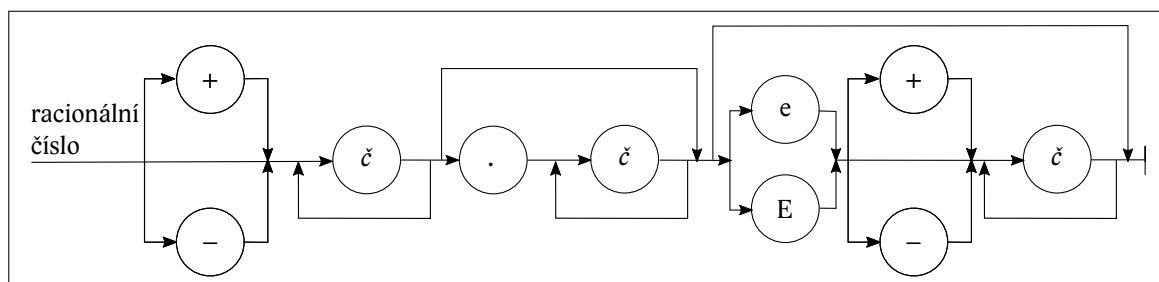
Kromě desítkových čísel lze zapisovat i čísla v osmičkové soustavě – ta začínají vždy nulou. Například **017** je desítkově 15.

Šestnáctková čísla se zapisují s předponou **0x** nebo **0X** – například číslo **0x21** je desítkově 33.



- **racionální literál** – racionální číslo zapsané svou hodnotou. Skládá se z celé části (část před desetinnou tečkou), z desetinné části (za desetinnou tečkou – pozor! nikoliv čárkou, jak je tomu v češtině) a z exponenciální části. Desetinná a exponenciální část je volitelná. Před číslem může být znaménko plus nebo minus. Příklady:

2	celočíselné desítkové literály jsou zároveň i racionálními čísly
-3.1415	celá a desetinná část
3e4	celá a exponenciální část; takto je zapsáno číslo $3 \cdot 10^4 = 30\,000$
-7.8E-3	celá, desetinná i exponenciální část; hodnota $-7,8 \cdot 10^{-3} = -0,007\,8$



- **znakový literál** – zápis libovolného jednoho znaku přítomného ve znakové sadě. Znak je ohraničen apostrofem. Přímě lze zapsat zobrazitelný znak, například `'b'` nebo `'7'`.

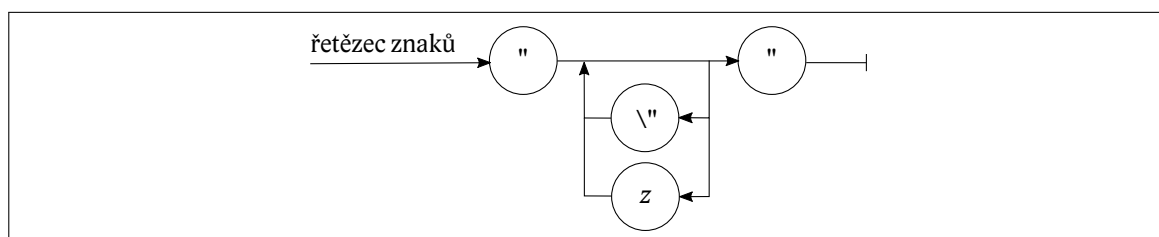
Chceme-li zapsat řídicí znak, který nemá žádný obraz, můžeme použít univerzální možnost zápisu pomocí osmičkového nebo šestnáctkového kódu znaku. Například `'\020'` je znak s kódem 16, nebo `'\x0F'` je znak s kódem 15. Některé význačné znaky mají i svoje speciální zápisy:

<code>\0</code>	NULL	znak s hodnotou nula
<code>\a</code>	Alert (Bell)	Pípnutí (na moderních počítačích už bohužel nefunguje)
<code>\b</code>	Backspace	návrat o jeden znak zpět
<code>\f</code>	Form feed	nová stránka nebo obrazovka
<code>\n</code>	Newline	přesun na začátek nového řádku
<code>\r</code>	Carriage return	přesun na začátek aktuálního řádku
<code>\t</code>	Horizontal tab	přesun na následující tabulační pozici
<code>\v</code>	Vertical tab	stanovený přesun dolů
<code>\\</code>	Backslash	obrácené lomítko
<code>\'</code>	Single quote	apostrof
<code>\"</code>	Double quote	uvozovky
<code>\?</code>	Question mark	otazník

Můžeme samozřejmě zapsat pomocí číselného kódu i jakýkoliv jiný znak, jen je to nevýhodné a tuto možnost pravděpodobně využijeme jen ve zcela speciálních případech.

- **řetězcový literál** – zápis posloupnosti znaků, tj. znakového řetězce. Zápis posloupnosti znaků je ohraničen uvozovkami, například `"vlek"`. Do řetězce lze zapisovat všechny znakové literály, zobrazitelné i řídicí znaky.

Příklad: `"Toto je řetězec\nna víc řádků\ns\"uvozovkami\".\n"`



- **logická hodnota** – literál `false` (nepravda) a `true` (pravda).
- **hranice bloku** – zapisuje se na začátku bloku levou složenou závorkou (`{`) a na konci bloku pravou složenou závorkou (`}`).
- **prvky výrazu** – výraz jako předpis pro výpočet nějaké hodnoty se podobně jako v matematice skládá z operandů (čísel, proměnných apod.) a operátorů (znamének pro jednotlivé operace). Kromě toho zde mohou být kulaté závorky pro vyjádření priority vyčíslování. Pro tento účel nelze na rozdíl od matematických výrazů použít ani hranaté ani složené závorky. Uvedeme některé operátory:

+	operátor sčítání, zřetězení
-	operátor odčítání
*	operátor násobení, znak pro odkaz na hodnotu dynamické proměnné
/	operátor dělení (celočíslné i reálné)
%	operátor zbytku po celočíselném dělení
==	operátor porovnání (je rovno)
!=	operátor porovnání (není rovno)
>=	operátor porovnání (větší nebo rovno)
<=	operátor porovnání (menší nebo rovno)
<	operátor porovnání (menší)
>	operátor porovnání (větší)
&&	operátor logického součinu
and	operátor logického součinu
 	operátor logického součtu
or	operátor logického součtu
!	operátor logické negace
not	operátor logické negace
xor	operátor výhradního logického součtu
++	operátor inkrementace (zvýšení o 1)
--	operátor dekrementace (snížení o 1)

Při vyčíslování výrazu hraje také podstatnou roli **priorita operátorů**. Platí, že operátor s vyšší prioritou se vyčísľuje *dříve*. Pravděpodobně všichni známe z matematiky prioritu násobení před sčítáním, která platí i zde – například ve výrazu `3 + 2 * 7` jednoznačně vypočteme výsledek 17 (napřed se vyčísľí násobení, pak se přičte). Pokud potřebujeme prioritu změnit, použijeme kulaté závorky, např. `(3 + 2) * 7` upřednostní sčítání před násobením a vede k výsledku 35.

V každém programovacím jazyce jsou však operátory prioritně poskládány poněkud jinak, navíc je jich podstatně více než v obvyklých matematických výrazech. Proto je vhodné se v případě nejistoty podívat do tabulky priorit. Kompletní tabulku priorit operátorů jazyka C++ uvádíme na s. 161.

Další informací nutnou ke správnému sestavení a pochopení výrazu je postup vyčíslování. Některé operátory se zpracovávají zleva doprava, jak by se dalo očekávat, některé se však zpracovávají zprava doleva. Příkladem takového operátoru je přiřazení – při zápisu více přiřazení v jednom výrazu se napřed provede přiřazení v nejpravější pozici, jeho výsledek se pak použije v levém zápisu atd.

- **znak přiřazení** – znak symbolizující vložení hodnoty výrazu do proměnné. Nejjednodušší forma přiřazení se zapisuje rovnítkem `=`. Operace přiřazení však může být kombinována s další aritmetickou operací, v tom případě se zapisuje `+=`, `-=` atd.

3.2.3 Bílé znaky

Pod pojmem **bílý znak** se v souvislosti se zápisem programů myslí veškeré oddělovače, které můžeme zapisovat libovolně mezi lexikální jednotky, a to především z důvodu přehlednosti zdrojového textu. Patří sem mezery, konce řádků, tabulátory a také poznámky. Veškeré bílé znaky překladač přeskakuje a nemají z hlediska funkce programu žádný význam.

Poznámky jsou poměrně důležitou součástí zápisu. Člověk, který čte zápis programu, může z dobře umístěné a formulované poznámky rychle zjistit, o co v daném místě jde, může tedy provést případný efektivní zásah. O tom, kolik poznámek a jakých se má nebo nemá uvádět v různých textech, bylo již popsáno mnoho papíru. Existují sice různá doporučení, ale nejdůležitější je řídit se především citem a vždy předpokládat, že program bude číst někdo jiný, kdo nemá třeba úplně stejné postupy myšlení.

V programech v tomto textu (a ve výukových textech obecně) jsou poznámky určeny především k pochopení právě probíraných jevů, takže jejich množství, umístění a obsah neodpovídá programátorské praxi. V některých firmách se například vyžaduje konkrétní forma a množství poznámek v typických situacích, protože je pak usnadněna zastupitelnost programátorů a snadnější přenášení úkolů od jednoho pracovníka ke druhému.

Poznámky v jazyce C++ mohou být řádkové nebo blokové. Řádková poznámka začíná dvěma lomítky `//` a vše, co se vyskytuje do konce daného řádku, je považováno za poznámku. Bloková poznámka má počáteční symbol `/*` a koncový symbol `*/` a její rozsah je libovolný (třeba jen část řádku nebo i více řádků).

3.3 První program

Z předchozí sekce už máme přehled, z jakých stavebních kamenů se může skládat zápis programu v jazyce C++. Ukážeme si nyní, jak vypadá jednoduchý program a jaký význam mají prvky, které tam potřebujeme. Obvykle se při prvním seznámení s nějakým programovacím jazykem používá program, který vypisuje jednoduchý anglický pozdrav „Hello, world!“. Protože budeme pracovat s českým prostředím, pozdrav napíšeme v češtině, zároveň můžeme vyzkoušet, že prostředí, které pro realizaci a spouštění programů používáme, má v pořádku národní znaky.

Minimální program může mít následující tvar:

```
1 | #include <iostream>
2 | //prostor pro globální definice
3 |
4 | int main(){
5 |     std::cout << "Zdravíme všechny céčkaře!" << std::endl;
6 |     return 0; /*předání hodnoty operačnímu systému*/
7 | }
```

Na řádku 1 připojujeme zápisem `#include` hlavičkový soubor – knihovnu, v níž se nacházejí mimo jiné všechny nástroje pro vstup a výstup dat. V našem prvním programu budeme potřebovat výstup

textu na terminál. Na řádce 4 začíná tzv. hlavní funkce programu – jde o funkci, představující program jako celek vzhledem k operačnímu systému a tato funkce také s operačním systémem komunikuje. Může přebírat údaje z příkazového řádku a vydává informaci o tom, jak celý program skončil. Parametry z příkazového řádku nyní nevyužijeme, takže toto spojení je zde vynecháno, ale informace o výsledku běhu se operačnímu systému předává příkazem `return` a domluvená hodnota nula představuje bezchybné ukončení.

Každý příkaz je zakončen znakem středník `;`.

3.4 Hlavičkové soubory

Hlavičkový soubor je soubor s rozšířením `.h`, který obsahuje deklarace a definice sdílené mezi několika zdrojovými soubory. Existují dva typy hlavičkových souborů: soubory, které si programátor napíše sám nebo od někoho zkopíruje, a soubory, které jsou součástí instalace překladače.

Použití hlavičkových souborů je nezbytné v téměř každém programu. Jedním z nejpoužívanějších hlavičkových souborů, který je součástí instalace, je soubor s definicemi vstupních/výstupních operací `iostream`.

Vložení hlavičkového souboru `jmeno.h` do programu se zapíše použitím tzv. preprocesorové direktivy `#include`. Zápis direktivy `#include` má následující dvě formy:

```
8 | #include <jmeno>
9 | #include "jmeno.h"
```

První zápis je určen pro vložení souboru, který je součástí instalace, druhý pak pro vložení vlastního souboru.

Vložení vlastního souboru může obsahovat i přístupovou cestu k souboru, zatímco předinstalované hlavičkové soubory jsou umístěny v systémových adresářích, jejichž cesty se neuvádějí (při instalaci překladače se nastavují automaticky). Soubory z těchto umístění se tedy vždy uvádějí bez rozšíření a bez cesty.

3.5 Deklarace proměnných a datové typy

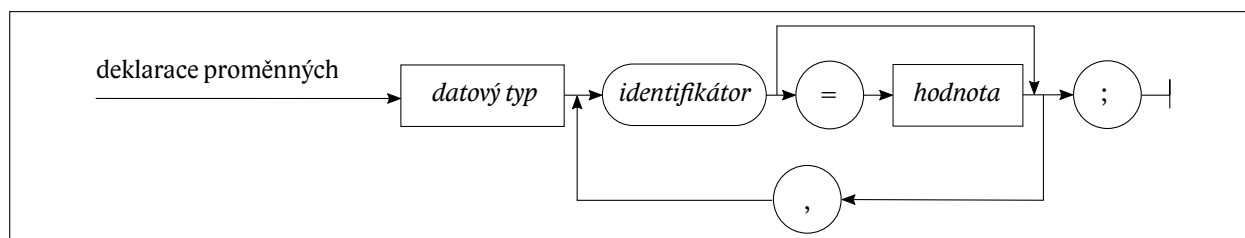
Pojem **proměnná** v souvislosti s programovacími jazyky představuje místo v operační paměti, kam lze uložit některou hodnotu odpovídajícího datového typu. **Datový typ** je specifikace povolených hodnot a povolených operací, které lze s těmito hodnotami provádět.

Pro zpracování jakýchkoliv dat potřebujeme paměťová místa, s nimiž následně můžeme manipulovat. Zvolíme si název takového místa (ten je ve tvaru identifikátoru) a požádáme systém, aby nám v paměti vyhradil prostor, kde se bude proměnná nacházet. Tento požadavek se řeší tzv. **deklarací proměnné**.

Při deklaraci je nutné uvést datový typ proměnné a její identifikátor. Těto proměnné se vyhradí v paměti příslušný prostor odpovídající hodnotám daného typu a adresa tohoto prostoru je vnitřně propojena s identifikátorem proměnné.

Chceme-li s proměnnou kdykoliv pracovat, uvedeme její identifikátor.

Způsob zápisu deklarace jednoduchých proměnných ukazuje příslušný syntaktický diagram.



Z diagramu je patrné, že můžeme u každé deklarované proměnné zároveň nastavit i její počáteční hodnotu, využijeme-li větve obsahující rovnítko a hodnotu. Pro jeden datový typ můžeme vypsát celý seznam proměnných oddělených čárkami.

Otázkou je, které proměnné deklarujeme v jednom seznamu a které v různých seznamech. Vodítkem je logická souvislost mezi proměnnými. K tomu uvedeme ilustrační příklad:

Předpokládejme, že máme nějaké číselné hodnoty a máme zjistit jejich součet a jejich počet. Zpočátku uvažujeme, že hodnoty jsou celočíselné (viz dále přehled datových typů). Deklarujeme:

```
10 | int Hodnota, Soucet = 0, Pocet = 0;
```

V tomto tvaru je ovšem porušena logická souvislost mezi proměnnými. Proč? Proměnné **Hodnota** a **Soucet** budou pravděpodobně stejného typu, protože předpokládáme, že součet hodnot určitého typu je hodnota téhož typu. Tyto dvě proměnné tedy budou v jednom deklaracím seznamu. Proměnná **Pocet** však nemá logicky tentýž obor – s hodnotami vůbec nesouvisí, protože uchovává pouze jejich počet. Ten navíc nemůže být záporný, což lze také v deklaraci vyjádřit. Proto je logicky daleko čistší zápis:

```
11 | int Hodnota, Soucet = 0;
```

```
12 | unsigned int Pocet = 0;
```

Když se navíc později rozhodneme, že číselné hodnoty budou zahrnovat třeba i desetinná čísla, stačí vyměnit pouze datový typ **int** na řádce 11 za typ **float** a opět bude vše v pořádku, protože obě proměnné tohoto typu jsou logicky svázány a přebírají stejný datový typ, zatímco proměnná **Pocet** stále zůstává stejného typu.

Uvedeme nyní některé datové typy, které jsou přímo k dispozici.

Identifikátor typu	Zkrácený identifikátor	Poznámka
<i>Celočíselné typy se znaménkem</i>		
<code>signed char</code>	<code>char</code>	1 B, rovněž pro znaky
<code>signed short int</code>	<code>short</code>	2 B, doplňkový kód
<code>signed int</code>	<code>int</code>	4 B, doplňkový kód
<code>signed long int</code>	<code>long</code>	jako <code>int</code>
<code>signed long long int</code>	<code>long long</code>	8 B, doplňkový kód
<i>Celočíselné typy bez znaménka</i>		
<code>unsigned char</code>	—	1 B (hodnoty 0–255)
<code>unsigned short int</code>	<code>unsigned short</code>	2 B
<code>unsigned int</code>	<code>unsigned</code>	4 B
<code>unsigned long int</code>	<code>unsigned long</code>	4 B
<code>unsigned long long int</code>	<code>unsigned long long</code>	8 B
<i>Desetinná čísla</i>		
<code>float</code>	—	4 B (IEEE single)
<code>double</code>	—	8 B (IEEE double)
<code>long double</code>	—	10 nebo 16 B (IEEE extended)
<i>Znakové typy</i>		
<code>char</code>	—	1 B, znaky ASCII
<code>char16_t</code>	—	2 B, UTF-16
<code>char32_t</code>	—	4 B, UTF-32
<code>wchar_t</code>	—	pro největší znakovou sadu
<i>Logický typ</i>		
<code>bool</code>	—	1 B, <code>false</code> , <code>true</code>

3.6 Výraz

Pojem **výraz** má v jazyce C++ zcela zásadní význam. Lze totiž říct, že téměř všechno, co napíšeme, je výraz, což je zde chápáno jako předpis, který po provedení dodá nějaký výsledek. Funkce `main` je výraz, jejím provedením získáváme hodnotu výstupního kódu. I příkazy jsou často chápány jako výrazy, neboť jejich provedením získáváme (číselnou) informaci o úspěchu nebo neúspěchu. V této souvislosti je potřebné zmínit jeden konkrétní případ:

Jaký je rozdíl mezi zápisem `v = 1` a `v == 1`?

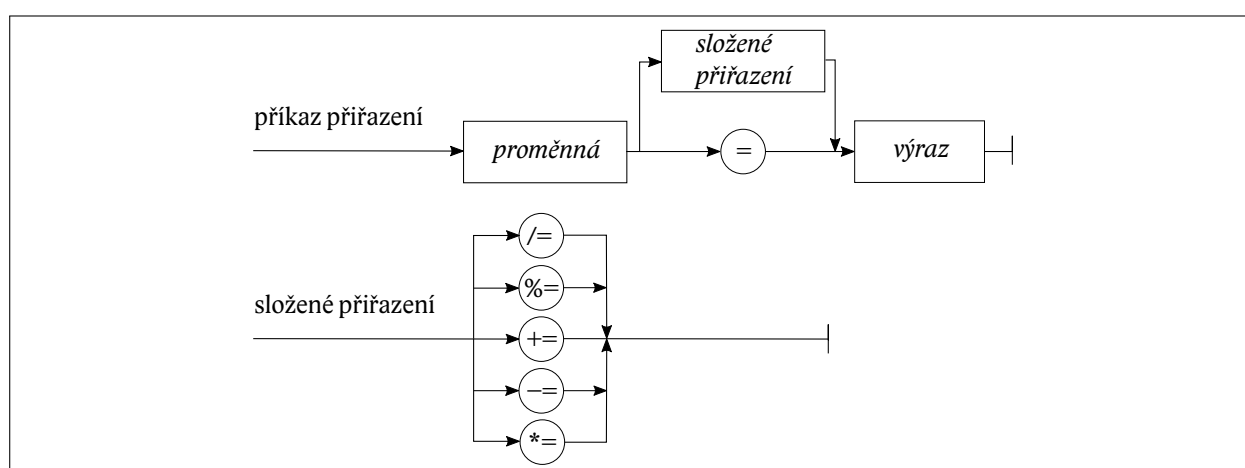
Důležité je, že oba zápisy jsou chápány jako výraz, takže uvedeme-li je v místě, kde se výraz očekává, nebude ohlášena žádná chyba. V prvním případě se však jedná zároveň o příkaz přiřazení, jehož efektem je, že hodnota 1 bude vložena do proměnné `v` a hodnotou celého výrazu bude ta přiřazovaná jednička, zatímco ve druhém případě se jedná o porovnání současné hodnoty proměnné `v` s jedničkou; výsledkem bude hodnota `false`, nebo `true` (je automaticky také interpretovatelná jako 0 a 1).

Záludnost této situace spočívá v tom, že záměnou uvedených výrazů se v mnoha případech *neprojeví chyba*. To vede při nedostatečném testování programu k mylnému přesvědčení, že je vše správně.

Budeme se však většinou na výraz dívat v poněkud užším smyslu, a to jako na posloupnost operandů a operátorů, jejímž provedením a výpočtem vznikne výsledná hodnota. Vybrané operátory byly již zmíněny na s. 26.

3.7 Příkaz přiřazení

V přehledu lexikálních jednotek byl zmíněn symbol pro přiřazení `=`. Ten je základem **příkazu přiřazení**. Jeho syntax je možné znázornit tímto diagramem:



Pod pojmem „proměnná“ v syntaktickém grafu si můžeme představit jednoduchou proměnnou zapísanou pouze identifikátorem, ale také to může být složka větší struktury nebo dynamické proměnné, pak je zápis složitější a není v této chvíli vhodné jej celý zobrazovat.

Tato forma zápisu příkazu představuje vždy proces složený ze tří kroků:

1. Vyčíslení výrazu na pravé straně přiřazení.
2. Kontrola datového typu výsledku a datového typu proměnné na levé straně, provedení případné konverze (viz dále).
3. Vložení výsledné hodnoty do proměnné na levé straně.

Příklad:

```

13 | Objem = hranaA * hranaB * vyska;
14 | Plocha = strana * vyska / 2;
15 | Soucet = Soucet + Hodnota;

```

V příkladu na řádce 15 je uvedena proměnná `Soucet` na levé i na pravé straně. Musíme si zde uvědomit proces přiřazení: nejprve se vyčíslí hodnota výrazu na pravé straně. V té chvíli má proměnná `Soucet` původní hodnotu. K té se přičte obsah proměnné `Hodnota` a tento součet se následně vloží

do proměnné `Soucet`. Tato proměnná figuruje v přiřazovacím příkazu ve dvou časech – s původní hodnotou na pravé straně, s novou hodnotou na levé straně.

Uvedená situace s přičítáním se vyskytuje poměrně často a navíc se místo přičítání může objevit i jiná aritmetická operace. Proto existuje tzv. **složené přiřazení** naznačené v syntaktickém diagramu. Jde o zkrácení zápisu takových případů, v nichž se proměnná použitá na levé i pravé straně zapisuje pouze na levou stranu. Takže následující zápisy jsou navzájem ekvivalentní:

```
P = P + A; ↔ P += A;
P = P - A; ↔ P -= A;
P = P * A; ↔ P *= A;
P = P / A; ↔ P /= A;
P = P % A; ↔ P %= A;
```

Další zkrácení zápisu můžeme použít v případě, že přičítáme nebo odečítáme jedničku. Použijeme k tomu operátor inkrementace, resp. dekrementace.

```
P = P + 1; ↔ P += 1;
P = P + 1; ↔ P++;
P = P - 1; ↔ P -= 1;
P = P - 1; ↔ P--;
```

Aby to ovšem nebylo tak jednoduché, lze tento operátor zapsat před proměnnou nebo za proměnnou a má to poněkud rozdílný efekt. Ten se projeví při použití ve výrazu. Zapišeme-li inkrement před proměnnou (nazývá se to *preinkrement*), zvýší se hodnota proměnné o 1 a teprve *potom* se tato hodnota použije ve výrazu. Zapišeme-li inkrement za proměnnou (nazývá se to *postinkrement*), *napřed* se použije původní hodnota proměnné ve výrazu a teprve potom se zvýší o jedničku. Analogicky totéž platí pro dekrement (*predekrement*, *postdekrement*).

Příklady:

```
16 | int x = 10; y;
17 | y = x++ * 5; //y je 50 a nová hodnota x je 11
18 | y = ++x * 5; //y je 60 a nová hodnota x je 12
```

Vidíme, že použijeme-li inkrement/dekrement ve výrazu, udělají se dva výpočty: jeden výpočet souvisí se zvýšením/snížením hodnoty proměnné a druhý výpočet souvisí s celým výrazem.

3.8 Konverze a přetypování

V souvislosti s různými datovými typy často potřebujeme, aby se v určitém místě provedla změna typu. Tuto změnu můžeme vyřešit dvěma poněkud odlišnými způsoby:

1. **Přetypování** – změna datového typu, *aniž by se měnil paměťový obsah*. Jde pouze o jiné chápání stejné posloupnosti nul a jedniček uložených v paměti počítače. Příkladem může být změna celočíselné hodnoty na logickou hodnotu – fyzicky uložená binární hodnota `00000001` je jed-

nou chápána jako celočíselná hodnota **1**, po přetypování je pak chápána jako logická hodnota **true**.

2. **Konverze** – změna datového typu a současně *změna – přepočítání paměťového obsahu, případně i změna velikosti obsazené paměti*. Příkladem může být změna celočíselné hodnoty na desetinné číslo, kdy hodnota **1** typu **char** na jednom bajtu je přepočtena na hodnotu **1.0** typu **float** na čtyřech bajtech.

Přetypování nebo konverze může být provedena automaticky, tuto situaci nazýváme **implicitní konverze (implicitní přetypování)**, například:

```
19 | int cele = 48; char kratke; float realne;  
20 | realne = cele; //konverze celočíselné hodnoty na desetinnou  
21 | kratke = cele; //konverze (zkrácení) 4bajtové hodnoty na 1bajtovou  
22 | retez = kratke + '1'; //přetypování "kratke" na znak a zřetězení
```

Pokud potřebuje změnit datový typ a automatická změna z nejrůznějších důvodů nefunguje, můžeme provést **explicitní konverzi (explicitní přetypování)**. Zápis explicitní změny typu existuje dvojnásob:

```
23 | int cele = 10, j = 3; float podil;  
24 | podil = cele / j;  
25 | podil = float(cele) / j;  
26 | podil = (float)cele / j;
```

Řádek 24 ukazuje důvod, proč vůbec potřebujeme změnu typu někdy vynutit explicitním způsobem – podívejme se podrobněji, co se zde děje: Nejprve se vypočítá výraz na pravé straně přiřazení. Je tam celočíselná proměnná **cele**, operátor dělení a celočíselná proměnná **j**. Z toho vyplývá, že operátor dělení má z obou stran celočíselné datové typy, realizuje se v tomto případě jako *celočíselné* dělení, jehož výsledkem je hodnota 3. Pak se provede (v soulase s principem činnosti příkazu přiřazení) konverze výsledku na desetinné číslo typu **float** kvůli proměnné na levé straně přiřazení. V ní se objeví výsledek **3.0**. Je ovšem pravděpodobné, že toto nebylo úplně zamýšleno – potřebovali jsme desetinné dělení a desetinný výsledek uložit do proměnné tohoto typu. Musíme překladači sdělit, že datový typ alespoň jednoho operandu kolem lomítka je desetinné číslo, což provedeme explicitní změnou typu.

Na řádce 25 je tzv. funkcionální zápis změny typu použitelný pouze pro datové typy zapsatelné jedním identifikátorem nebo klíčovým slovem. Na řádce 26 je změna typu použitelná pro jakékoliv datové typy, tato syntax je převzata z jazyka C. Ať už použijeme kterýkoliv z uvedených způsobů, nyní se výraz vyhodnotí jinak: konverzí celočíselné proměnné **cele** na typ **float** se dělení provede jako desetinné, datový typ výrazu na pravé straně bude také **float** (hodnota 3,333333333) a výsledek se vloží do proměnné stejného typu na levé straně.

Kromě uvedených způsobů existují další možnosti změny typu definované pouze v jazyce C++:

- `reinterpret_cast` – slouží k pouhému přetypování (změně chápání určitého paměťového místa). Nekontroluje se, jaký je původní a jaký je výsledný typ, to musí programátor pečlivě pohlídat sám. Zápis: `reinterpret_cast<typ>(výraz)`
- `static_cast` – slouží stejným způsobem jako zápis `(typ)`. Zapisuje se ovšem jinak: `static_cast<typ>(výraz)`.
- `dynamic_cast` – slouží výhradně k přetypování ukazatelů nebo referencí a jeho hlavní význam je v objektovém programování, nebudeme se jím v tomto textu proto zabývat. Zápis je obdobný jako v předchozích dvou případech: `dynamic_cast<typ>(výraz)`.

3.9 Vstupy a výstupy

V jazyce C++ je pro vstup a výstup hodnot k dispozici skupina tzv. proudů. K jejich použití musíme (jak už bylo zmíněno) připojit knihovnu `iostream`.

Vstupní proud pro standardní vstup se jmenuje `cin` a výstupní proud pro standardní výstup se jmenuje `cout`.

Objekty pro vstup a výstup jsou uloženy ve jmenném prostoru `std`, proto musíme před každou operací psát kvalifikaci jmenného prostoru `std::`.

Pro jednoduchý vstup můžeme použít příkaz v následující formě:

```
std::cin >> proměnná;
```

Jméno „cin“ je zkratka slov „console input“. Hodnota ze standardního vstupu je uložena do proměnné podle jejího datového typu, automaticky dochází k potřebné konverzi ze znakového tvaru do tvaru v operační paměti. Použijeme-li více operátorů `>>`, můžeme číst více hodnot. Toto je nejjednodušší forma pro obecný vstup.

Příklad:

```
27 | std::cin >> prvni >> druha;
```

Analogicky lze pro jednoduchý výstup použít následující příkaz:

```
std::cout << výraz;
```

kde „výraz“ je napřed vyčíslen a jeho hodnota pak vystupuje do standardního výstupu. Jméno „cout“ je zkratkou „console output“. Operátor `<<` má určitou prioritu – poznamenejme, že operátory ve výrazu mohou mít nižší prioritu a pak dojde k chybné interpretaci. Proto je někdy nutné výraz uzavřít do závorek. Pro výstup více výrazů lze operátor `<<` použít vícekrát.

Příklad:

```
28 | std::cout << "Výsledná hodnota je " << prvni * druha << ".\n";
```

Zápis `\n` vloží do výstupu přechod na nový řádek.

Výstupní proud má ještě vnitřní metody dostupné tečkovou notací, například:

- `precision(n)` – nastavuje počet platných číslic reálných čísel na n ;
- `width(n)` – nastavuje šířku výstupu následujícího výrazu na n znaků;
- `fill(c)` – nastavuje znak, kterým bude doplněn vystupující výraz místo mezer.

Příklad:

```
29 float polomer=2.752, objem;  
30 objem = 4/3 * M_PI * polomer * polomer * polomer;  
31 std::cout.precision(4); std::cout.width(40); std::cout.fill(' ');  
32 std::cout << objem << std::endl;
```

Výstupní hodnota bude:

```
////////////////////////////////////65.48
```

Některé z těchto formátovacích nastavení jsou dostupné i prostřednictvím tzv. manipulátorů dostupných v knihovně `iomanip`. Detailní formátování výstupu bude diskutováno později.

Kromě standardního vstupu a výstupu může program vypisovat různá chybová hlášení a další zprávy do tzv. **standardního chybového výstupu**, `cerr` (console error). Tento objekt se chová stejně jako „obyčejný“ standardní výstup `cout`. V programech jej budeme využívat na všechny případy, kdy nevypisujeme výstupní data, ale jakoukoliv jinou zprávu, nejčastěji záznam o chybě (nejsou data, v datech jsou nesrovnalosti, výsledek je mimo předpokládaný rozsah, program nebylo možné v důsledku selhání dokončit atd.). Pokud chceme vypisovat nějaké popisné zprávy, například informace o úspěšném dokončení procesu, záznam o čase, kdy byl program spuštěn apod., můžeme použít objekt `clog` (console log = výpis). Jeho použití je opět stejné jako u standardního výstupu a standardního chybového výstupu.

Zásadně nemícháme servisní zprávy mezi data! Uvažujeme-li například, že na běh programu jsou navázány další procesy, které předpokládají pouze validní data, vyprodukování chybového hlášení do datového proudu způsobí, že následující proces zhavaruje.

Všechny standardní soubory (vstup, výstup, chybový a logový výstup) lze v operačním systému přesměrovat. Ušetříme tím spoustu zbytečné práce s opakovaným vkládáním dat – data jednou připravíme do souboru a ten při spuštění vždy jen přesměrujeme na standardní vstup. Program uvnitř vůbec neví, zda jsou data přímo vkládána z klávesnice (terminálu), nebo jsou přesměrována ze souboru – chování vstupu je (až na drobné odlišnosti) totožné. Podobně nemusíme delší výsledky číst na obrazovce, ale můžeme je přesměrovat do souboru, který následně lze zobrazovat vhodným programem (editorem) nebo zpracovávat dalšími programy. Standardní chybový a logový výstup může sloužit jako záznamník všech servisních, chybových a doplňkových zpráv. Operační systém je schopen tyto standardní soubory sám ošetřovat a řešit běžné chybové stavy. Pro programátora je pak jejich využití velmi výhodné.

Zásadně nepoužíváme výstupy jako prostředek dialogu s uživatelem. To souvisí jednak s předchozím doporučením, že se do výstupu nemají kromě validních dat míchat jakékoliv jiné zprávy, jednak také s tím, že výstupy mohou být přesměrovány a pak je žádný uživatel sedící u příkazového řádku stejně neuvidí. Tyto dialogy (např. „Zadej hodnotu:“ nebo „Vkládejte čísla oddělená mezerou, posledním číslem je 100“ apod.) pocházejí z dnes již pravěkých programů, které v primitivních operačních systémech typu MS-DOS byly schopny komunikovat pouze prostřednictvím textových zpráv. V dnešní době si můžeme vybrat – buď aplikace bude komunikovat s uživatelem, který u ní sedí, ale v tom případě použijeme některý z mohutných systémů řešících celé uživatelské rozhraní včetně nabídek, různých dialogových oken apod., nebo bude aplikace provozována víceméně v automatickém režimu, bude mít za úkol zpracovat nějaká vstupní data a vygenerovat výsledky, aniž by ji někdo pozoroval při běhu a nějak s ní komunikoval. Tato druhá varianta je zároveň velmi vhodná k výuce algoritmů, protože neobsahuje mohutný balast uživatelského rozhraní³, ale soustřeďuje se pouze na vlastní činnost (algoritmus), přičemž získávání a produkci dat zjednodušuje na nejvyšší možnou míru.

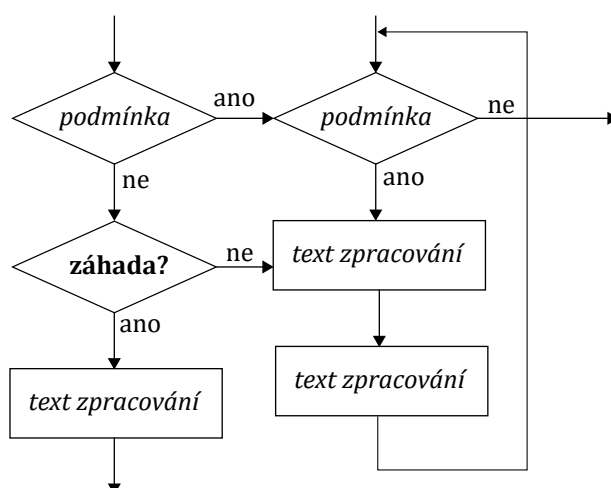
³ Říká se, že kód související s uživatelským rozhraním tvoří až 90 % celého kódu programu.

4 Základní algoritmické obraty

Mezi základní algoritmické obraty počítáme takové prvky, z nichž se většinou skládají algoritmy. Představíme tyto prvky obecně a doplníme k nim jejich reprezentaci v jazyce C++.

4.1 Větvení

S větvením jsme se setkali již při konstrukci vývojových diagramů. Tam se ale větvení řeší pouze značkou, do níž se zapisuje podmínka, ale již se dále neřeší, kam vedou a kudy probíhají příslušné větve. Podívejme se na úsek vývojového diagramu na obr. 3. Větvení je zde použito několikrát, věnujme pozornost bloku s nápisem „záhada“. Co všechno se vlastně provádí, pokud podmínka není splněna? Vidíte, že není jasné, co ještě patří do větvení a co patří k jiné struktuře. Ve vývojovém diagramu je takové spojení bez problémů možné, zapsat ovšem něco podobného v jazyce, jako je C++, není vůbec jednoduché. Kde je příčina tohoto zmatku? Jde především o konstrukční součástky, ze kterých algoritmus skládáme. Pokud jsou tak elementární, jako jsou součástky vývojových diagramů, můžeme dospět k uvedenému nepřehlednému řešení. Pokud ale používáme vyšší celky, nebudeme schopni z nich složit něco podobného, musíme nutně algoritmus koncipovat jinak.



Obrázek 3: Úsek vývojového diagramu s libovolným propojením elementárních prvků

Větvení, s nímž budeme dále pracovat, tedy bude obsahovat víc než jen rozhodovací blok s podmínkou. Budou do něj zahrnuty i obě větve a do „okolí“ bude celý prvek zapojen pouze jediným vstupním a jediným výstupním bodem.

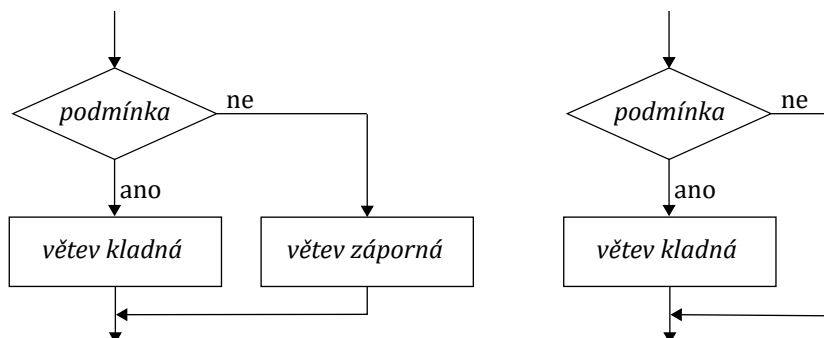
4.1.1 Úplné větvení

Jedná se o větvení, kdy v případě splnění podmínky se provádějí nějaké příkazy, v případě nesplnění se provádějí jiné příkazy. Po skončení příkazů obou větví se řízení spojuje do jednoho bodu.

4.1.2 Neúplné větvení

Jde o větvení, v němž jedna z větví je prázdná. Obvykle se toto větvení zapisuje tak, aby prázdnou větví byla ta, která přichází na řadu při nesplnění podmínky.

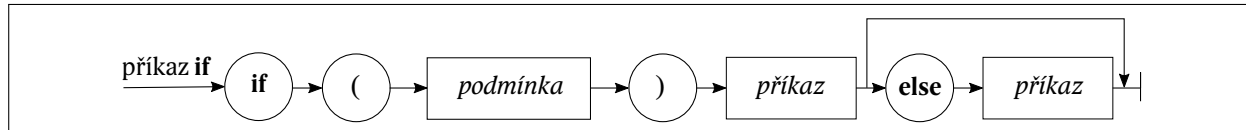
Obě varianty větvení jsou naznačeny vývojovým diagramem na obr. 4.



Obrázek 4: Úplné (vlevo) a neúplné větvení jako komplexní blok s jedním vstupním a jedním výstupním bodem

4.1.3 Větvení v jazyce C++, příkaz if

V jazyce C++ máme pro oba uvedené bloky k dispozici příkaz **if**. Nejprve uvedeme jeho syntaktický diagram.



Příkaz začíná klíčovým slovem **if**, za nímž následuje podmínka uzavřená do kulatých závorek (jsou tam povinné). Za podmínkou se zapisuje příkaz, jenž se provádí při splnění podmínky. Následuje volitelná větev začínající klíčovým slovem **else**, za nímž je příkaz prováděný při nesplnění podmínky. Chceme-li zapsat neúplné větvení, bude část se slovem **else** vynechána.

Příklad: V proměnných **A** a **B** jsou uložena celá čísla. Zjistěte, které z nich je větší, a vypište tuto informaci na výstup.

```

33 | if (A > B) std::cout << "Větší je A";
34 |     else std::cout << "B je větší nebo rovno A";

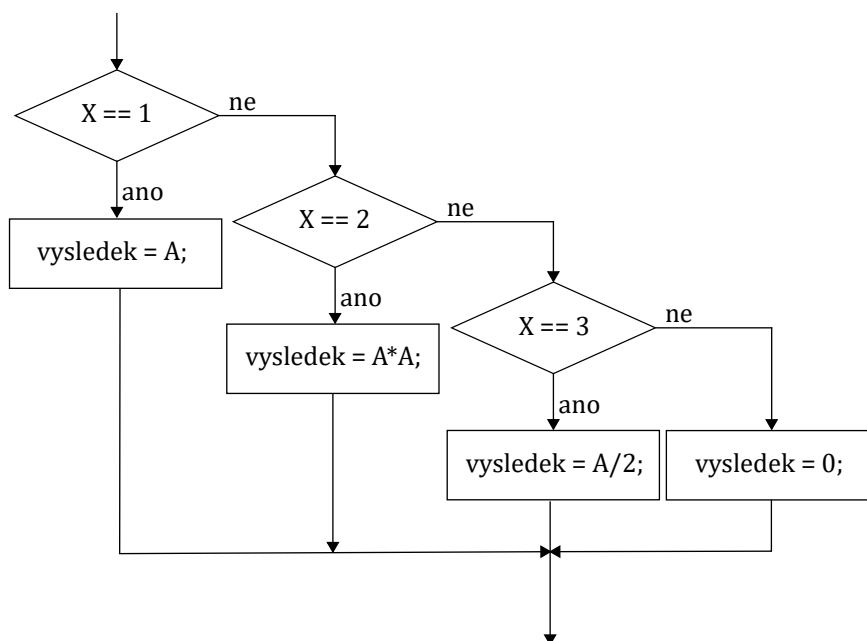
```

Všimněte si i způsobu zápisu. Vzhledem k tomu, že jazyk C++ má volnou strukturu, můžeme kdekoliv mezi lexikálními jednotkami udělat mezery, využíváme toho pro přehledné zformátování zápisu tak, abychom byli schopni odhadnout na první pohled, které části zápisu k sobě logicky patří. Protože část **else** logicky patří k příkazu **if**, zapisujeme ji s levostrannými mezerami.

Na styl zápisu zdrojového textu existuje mnoho pohledů, důležité je, abychom dodržovali jednotný postup, který usnadní čtení programů a rychlé pochopení jejich podstaty.

4.1.4 Vícenásobné větvení

Uvažujme případ, kdy máme celočíselnou proměnnou `X`, jejíž některé hodnoty budou určovat, jak máme zpracovat obsah proměnné `A`. Pro `X==1` se proměnná `A` použije přímo, pro dvojku ve druhé mocnině a pro trojku v polovině. Je-li hodnota `X` jiná, nepoužije se proměnná `A` vůbec, výsledkem zpracování bude nula. Popsaný algoritmus znázorňuje vývojový diagram na obr. 5.



Obrázek 5: Vícenásobné větvení – speciální případ výběru několika hodnot

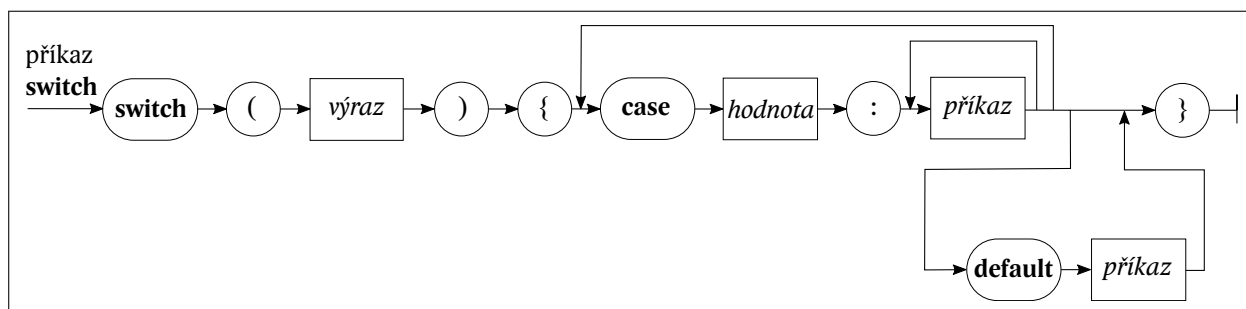
Pro zápis takového větvení můžeme použít několik do sebe vnořených příkazů `if`:

```

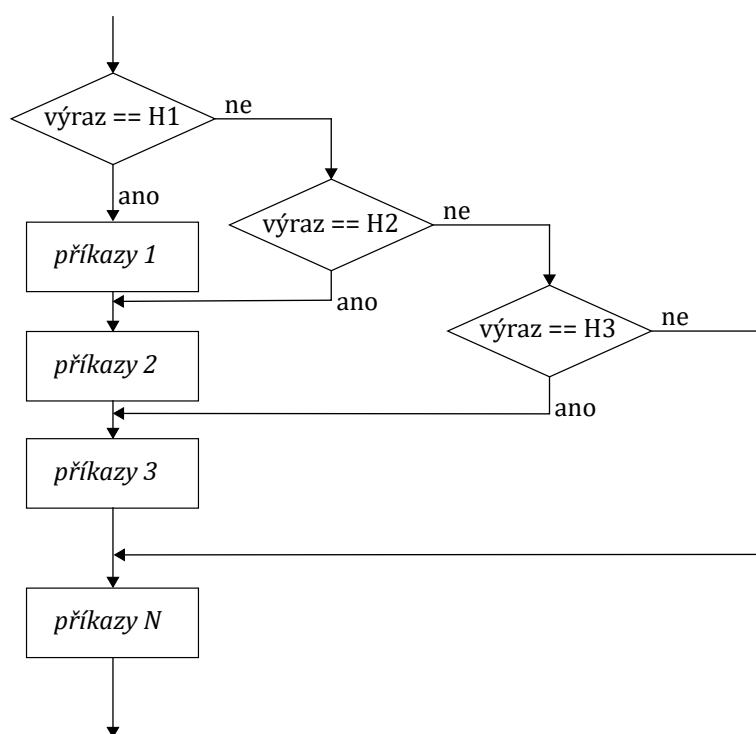
35 int X;
36 float A, vysledek;
37 cin >> X >> A;
38 if (X == 1) vysledek = A;
39     else if (X == 2) vysledek = A*A;
40         else if (X == 3) vysledek = A/2;
41             else vysledek = 0;

```

V jazyce C++ však existuje ještě i příkaz `switch`, pomocí něhož lze tuto situaci zapsat mírně přehledněji. Jeho syntaktický diagram má následující tvar:



Přesné chování příkazu **switch** ukazuje vývojový diagram na obr. 6. To ovšem není proces, který odpovídá uvedenému příkladu – rozdíl spočívá v tom, že po provedení příkazů jedné větve se automaticky provádějí všechny příkazy všech následujících větví.



Obrázek 6: Chování příkazu **switch**

Abychom po provedení některé z větví mohli celý příkaz ukončit, používá se k tomu poněkud nečistý mechanismus – speciální příkaz **break**. Tento příkaz umožňuje z jakéhokoliv místa strukturovaného příkazu skočit přímo na jeho konec. Jako poslední příkaz každé větve je tedy nutné psát **break**.

Zápis příkladu pomocí příkazu **switch**:

```

42 int X;
43 float A, vysledek;
44 cin >> X >> A;
45 switch (X) {
46     case 1: vysledek = A; break;
47     case 2: vysledek = A*A; break;

```

```

48 |     case 3: vysledek = A/2; break;
49 |     default vysledek = 0;
50 | }

```

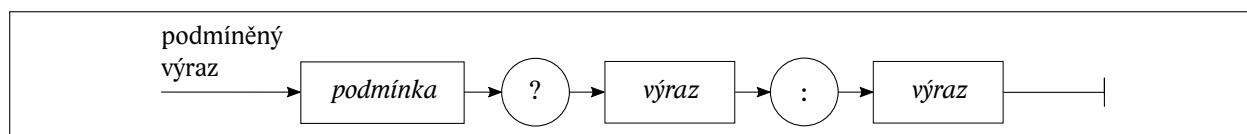
Vzhledem ke skutečnosti, že můžeme v každé větvi řešit pouze jednu hodnotu výrazu, je použití této konstrukce velmi omezeno a nevzniká tím ani žádné výrazné zkrácení zápisu. Použijeme-li místo ní posloupnost příkazů `if`, dosáhneme prakticky téže délky zápisu a jeho přehlednost bude jen mírně horší.

4.1.5 Podmíněný výraz

Je-li výsledkem větvení pouze výpočet výrazu, je možné takové větvení zapsat jednodušším způsobem místo příkazu `if`.

K tomuto účelu slouží **podmíněný výraz**. Jak vyplývá z názvu, jde o výraz, který můžeme zapsat všude tam, kde se výraz očekává. Protože však v jazyce C++ může výraz sloužit i jako příkaz, lze podmíněný výraz zapsat i jako příkaz.

Způsob zápisu podmíněného výrazu znázorňuje následující syntaktický diagram:



Je-li podmínka vyhodnocena s výsledkem `true`, celek nabývá hodnoty prvního výrazu mezi otazníkem a dvojtečkou. V opačném případě celek nabývá hodnoty druhého výrazu (za dvojtečkou).

Kombinace operátorů `?` a `:` je často nazývána **ternární operátor** – má tři operandy: podmínku, první výraz a druhý výraz.

Potřebujeme například vypočítat jízdné závisující na procestované vzdálenosti. Od nuly do 19 km se platí 1 €, nad 20 km se platí 2 €. Výpočet bychom mohli zapsat pomocí příkazu `if`:

```

51 | if (Vzdalenost < 20) Jizdne = 1;
52 | else Jizdne = 2;

```

Totéž ovšem můžeme zapsat pomocí podmíněného výrazu:

```

53 | Vzdalenost < 20 ? Jizdne = 1 : Jizdne = 2;

```

V této variantě využíváme skutečnosti, že přiřazení je zároveň výrazem, ale i příkazem, celý zápis je pak chápán jako příkaz.

Můžeme se vyjádřit ještě stručněji:

```

54 | Jizdne = Vzdalenost < 20 ? 1 : 2;

```

V tomto okamžiku jde o zápis přiřazovacího příkazu, na jehož pravé straně je podmíněný výraz, tentokrát ale v roli výrazu nabývajícího hodnoty 1, nebo 2 (podle hodnoty podmínky).

4.2 Cyklus

Cyklus představuje možnost opakování nějaké posloupnosti příkazů. Je velmi důležitou a také velmi frekventovanou součástí algoritmů. Stejně jako větvení, je i cyklus komplexnějším blokem, jehož správná aplikace je pro většinu algoritmů zcela zásadní. Proto se budeme zabývat všemi důležitými vlastnostmi, které umožní do detailu pochopit, jak cyklus konstruovat, abychom se vyhnuli nepříjemným chybám a umožnili co nejsnadnější případné další úpravy vytvořeného algoritmu.

Cykly můžeme rozdělit na dvě kategorie:

1. cykly podmíněné;
2. cykly počítané.

4.2.1 Podmíněný cyklus

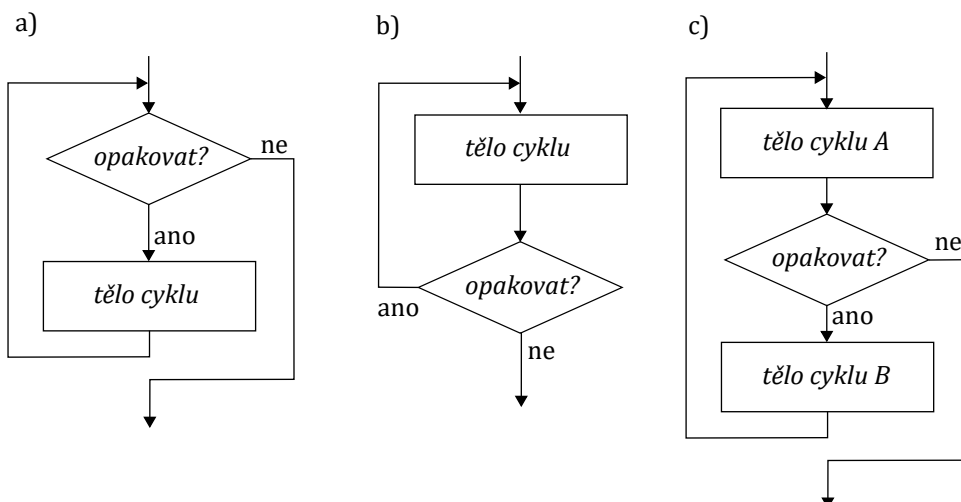
Základní vlastností jakéhokoliv cyklu (a současně i algoritmu) je konečnost. Při opakování určité posloupnosti příkazů musíme vždy zajistit, že toto opakování někdy skončí, součástí smyčky nutně musí být alespoň jedno větvení.

Podmíněný cyklus opakuje příkazy ve svém těle právě tak dlouho, dokud to umožňují podmínky všech větvení zahrnutých do cyklové smyčky. Chceme-li, aby se cyklus vytvářel co nejjednodušeji a měl co nejpřehlednější strukturu, zahrnujeme do cyklové smyčky *právě jedno větvení*, které má vliv na ukončení cyklu (tzv. koncovou podmínku). Téměř nikdy neuvažujeme více výstupních bodů – to je vždy možné chápat jako logickou chybu konstrukce algoritmu, či výsledek diletantského zásahu při nějakých opravách. Jediná koncová podmínka zaručuje, že v jednom místě můžeme sledovat, kdy má cyklus provádět opakování a kdy má skončit.

Celý cyklus s jedinou koncovou podmínkou se opět (podobně jako již diskutované větvení) připojuje k okolním částem jediným vstupním bodem a jediným výstupním bodem. Větvení ukončující průchod cyklem můžeme obecně položit do tří míst (viz ilustrační vývojové diagramy na obr. 7):

- a) na úplný začátek cyklu;
- b) na úplný konec cyklu;
- c) do jiného místa v těle cyklu.

Podívejme se teď na vlastnosti těchto tří možností. Cyklus, jehož podmínka ukončení je na začátku, má jednu podstatnou výhodu: není-li tato podmínka splněna, tělo cyklu neproběhne ani jednou. Lze tedy zabránit zpracování nesprávných hodnot.



Obrázek 7: Možnosti umístění větvení ukončujícího průchod cyklem

Cyklus s podmínkou na konci naopak vždy proběhne minimálně jednou. Hodí se tedy pro případy, kdy jedno provedení těla cyklu nevadí, nebo je dokonce žádoucí.

Ve třetím případě však podmínka ukončení rozděluje tělo cyklu na dvě části (A a B). Přitom je zřejmé, že část B neprobíhá tolikrát, jako část A, nýbrž má o jeden průchod méně. To může vést k záludným chybám, protože na první pohled nemusí být přesné umístění výskoku z cyklu dobře patrné, programátor počítající s tím, že určitý příkaz se provede vždy, pak může být nepříjemně překvapen. Těto variantě se proto vždy vyhýbáme.

To, že jsme ze všech možných řešení cyklu vybrali pouze dva případy, a to cyklus s podmínkou na začátku a cyklus s podmínkou na konci, není žádná náhoda. Přesně na tyto dva případy jsou připraveny základní jazykové konstrukce programovacích jazyků, včetně jazyka C++. I když existují různé další „berličky“, umožňující porušení tohoto bezpečného stavu, budeme se jim (určitě v tomto textu, ale doufejme i v běžné praxi) systematicky vyhýbat.

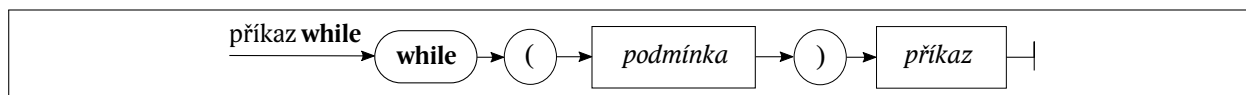
Důležitým prvkem konstrukce cyklu je záruka, že koncová podmínka bude někdy splněna. Proto v těle cyklu (nebo i v samotné podmínce) vždy musíme mít příkaz, který koncovou podmínku ovlivňuje, a musíme být schopni prokázat, že takový příkaz dokáže za všech okolností (a konfigurace dat) zajistit výskok ze smyčky cyklu.

4.2.2 Počítaný cyklus

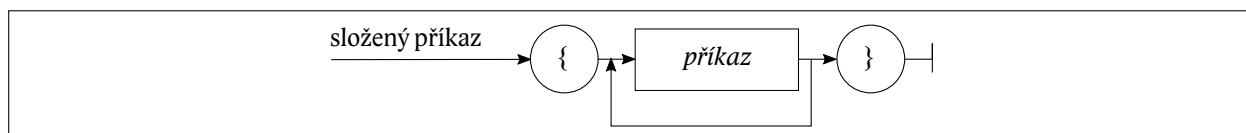
U počítaného cyklu již dopředu víme, kolikrát se má jeho tělo opakovat. Obvykle je počet opakování dán hodnotou nějaké proměnné, která je vypočtena a naplněna ještě před začátkem cyklu. Existuje řada algoritmů založených na tomto typu opakování. Protože již v dávných dobách rozkvětu vývojových diagramů byly v tehdejších programovacích jazycích hotové konstrukce realizující počítané cykly, existuje i komplexní značka vývojového diagramu naznačující tuto operaci. Je to na rozdíl od větvení nebo cyklu i v dnešní době prvek, připojující se jediným vstupním a jediným výstupním bodem ke svému okolí a zaručující konečnost provedení, což umožňuje jeho zahrnutí ke konstrukcím platným pro tvorbu a realizaci algoritmu.

4.2.3 Zápis cyklů v jazyce C++

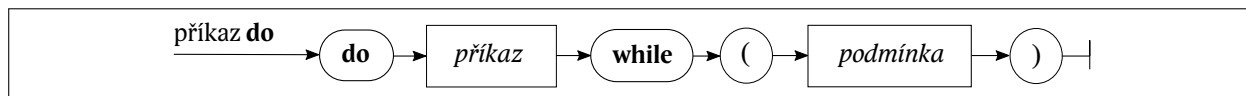
Podmíněné cykly jsou realizovány dvěma příkazy: cyklus s podmínkou na začátku se zapíše příkazem **while**, cyklus s podmínkou na konci pak příkazem **do**. Uvedeme syntaktické diagramy a popíšeme sémantiku obou příkazů.



Podmínka za klíčovým slovem **while** je umístěna povinně v kulatých závorkách (obdoba umístění podmínky u příkazu **if**). Uzavírací kulatá závorka jednoznačně identifikuje konec podmínky a začátek příkazu, jenž tvoří tělo cyklu. V diagramu je napsán jeden příkaz v těle cyklu – skutečně více příkazů tam není povoleno. Musíme si ale uvědomit, že tímto příkazem může být tzv. **složený příkaz**, který se navenek chová jako jeden příkaz, ale ve svém těle může mít příkazů více. Jeho syntax je jednoduchá: příkazy jsou uzavřeny do složených závorek:



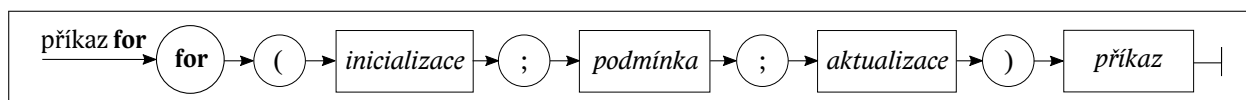
Druhým podmíněným cyklem je příkaz **do**:



Příkaz **do** může mít ve svém těle rovněž pouze jeden příkaz, může to opět být i příkaz složený obsahující více příkazů. Obě vyjádření podmíněných cyklů mají podmínku postavenou tak, že je-li pravdivá, provádí se tělo cyklu, a jakmile se stane nepravdivou, cyklus končí.

Pro počítaný cyklus není bohužel v jazyce C++ žádná odpovídající konstrukce k dispozici. Skutečný počítaný cyklus by měl být realizován příkazem, který automaticky provede stanovený počet průchodů tělem cyklu, aniž by bylo možné zvnějšku modifikovat okamžik ukončení nebo počet průchodů uvnitř těla. Nejblíže je tomuto prvku příkaz **for**, umožňuje však zapsat libovolnou variantu cyklu s podmínkou na začátku. Tvar zápisu při dodržení určité programátorské disciplíny soustřeďuje všechny podstatné informace do jednoho místa, a umožňuje tak skutečný automaticky pracující počítaný cyklus alespoň napodobit.

Syntaktická konstrukce je poněkud obsažnější:



Za klíčovým slovem **for** jsou opět v kulaté závorce uvedeny tři výrazy:

- *inicializace* – nastavení počátečních hodnot. Obvykle se zde nastavuje počáteční hodnota tzv. **řídící proměnné**, jejíž změna „počítá“ jednotlivé průchody tělem cyklu. V tomto místě může

být provedena zároveň deklarace této proměnné. V tom případě se jedná o proměnnou, která je k dispozici pouze v těle cyklu a mimo cyklus zaniká. To je ve většině případů výhodné, neboť tato řídicí proměnná je obvykle určena jen k tomu, aby odpočítala příslušný počet koleček a jinak pozbývá smyslu.

- *podmínka* – určuje svou pravdivou hodnotou opakování těla cyklu. Jakmile se podmínka stane nepravdivou, cyklus končí. V podmínce obvykle figuruje řídicí proměnná a porovnává se s koncovou hodnotou.
- *aktualizace* – požadovaná změna, která se provede vždy automaticky *na konci těla cyklu*. Jde nejčastěji o zvýšení (snížení) řídicí proměnné o jedna (někdy i o jinou hodnotu).

Za závorkou následuje opět jeden příkaz představující tělo cyklu.

Příklady:

1. Zapišme cyklus, který bude vypisovat všechny členy menší než tisíc geometrické posloupnosti s počátečním členem 1 a kvocientem 2.

```
55 int clen=1; //deklarace proměnné "clen" a nastavení na hodnotu 1
56 do {
57     cout << clen << ", "; //výpis aktuální hodnoty členu
58     clen = clen * 2; //nový člen - násobení kvocientem 2
59 } while (clen<1000);
```

2. Uvažujme, že ze vstupu jsou zadávána různá desetinná čísla, víme však, že posledním číslem je 10. Tato hodnota však logicky k předchozím číslům nepatří, jde pouze o „zarážku“ – indikaci, že vstupní hodnoty už skončily. Vypišme součet všech zadaných čísel.

```
60 float cislo, soucet=0; //deklarace dvou proměnných
61 std::cin >> cislo; //přečtení hodnoty ze vstupu do proměnné cislo
62 while (cislo != 10) { //čtené číslo není koncovou hodnotou
63     soucet = soucet + cislo; //k hodnotě soucet se přičte čtené číslo
64     std::cin >> cislo; //přečtení dalšího čísla ze vstupu
65 }
66 std::cout << "Součet je " << soucet << std::endl;
```

3. Máme na výstup vypsat druhé mocniny všech čísel od 20 do 40.

```
67 for (int cislo = 20; cislo <= 40; cislo = cislo + 1)
68     //dočasná proměnná cislo má počáteční hodnotu 20
69     //dokud je hodnota cislo menší nebo rovna 40, cyklus pracuje
70     //na konci těla se hodnota cislo zvýší o jedna
71     std::cout << "číslo " << cislo << " na druhou je "
72         << cislo * cislo << std::endl;
73     //vypisujeme proměnnou cislo a její druhou mocninu
```

4.3 Převody cyklů

V předchozím výkladu jsme se snažili ukázat, že na určitou situaci se nejlépe hodí vždy jeden ze tří možných typů cyklů. Tím se prakticky vždy budeme řídit, neboť cílem je psát efektivní programy a používat vhodné nástroje.

Můžeme si ale nyní naopak zase odvodit, že cykly jsou použitelné prakticky univerzálně. Z teoretické informatiky je například dobře známo, že veškeré opakování lze vždy popsat jediným typem cyklu, a tím je typ **while**. Proto platí, že zbylé dva cykly lze přepsat vždy na tvar cyklu **while**. Jak takový převod vypadá?

Mějme cyklus typu **do** v obecné formě:

```
74 | do S; //S = tělo cyklu
75 | while (B); //B = podmínka
```

Jak bude vypadat zápis pomocí cyklu **while** s ekvivalentním chováním:

```
76 | S; //tělo musí být provedeno alespoň jednou
77 | while (B) S;
```

Nyní mějme cyklus **for** v obecné formě:

```
78 | for (IS; B; A) S;
79 | //IS = inicializační příkaz, B = podmínka, A = aktualizací příkaz
```

Použitím cyklu **while** s ekvivalentním chováním dostáváme:

```
80 | IS; //nutné provést před cyklem
81 | while (B) {
82 |     S;
83 |     A; //aktualizace je na konci těla
84 | }
```

Při opačném převodu ze zápisu cyklu **while** na cyklus **for** je potřebné v textu přesně detekovat, co vše patří do inicializační části a co vše patří do aktualizace, která ovlivňuje koncovou podmínku cyklu. Protože zápis cyklu **while** je obecnější, nemusí být převod na cyklus **for** viditelný na první pohled a občas je nutná určitá úprava tvaru.

Mějme nakonec ještě cyklus **while** v obecné formě:

```
85 | while (B) S;
```

a převedme jej na zápis s cyklem **do** a s ekvivalentním chováním:

```
86 | do {  
87 |     if (B) S;  
88 | } while (B);
```

Ze všech zápisů by mělo být patrné, že základní obecný tvar každého typu cyklu je poněkud kratší než tvar získaný po převodu na jiný typ cyklu. To jen dokazuje už zmíněný fakt, že správná aplikace určitého typu cyklu na odpovídající situaci vede k optimálnímu (nejkratšímu a nejprehlednějšímu) zápisu. Proto budeme vždy dbát na to, abychom zvolili odpovídající typ cyklu, i když nyní víme, že zapsat libovolnou cyklickou situaci lze jakýmkoliv typem cyklu.

4.4 Poškození průběhu cyklu

Již v počátečních částech této kapitoly týkajících se cyklů jsme odvodili, že optimálním tvarem cyklu je varianta s podmínkou pouze na začátku, nebo varianta s podmínkou pouze na konci. Je to velmi důležité pro správnou, přehlednou a snadno udržitelnou konstrukci programu. Cyklus s podmínkou uprostřed těla nebo dokonce s více podmínkami umožňujícími ukončení těla je z uvedených důvodů nevhodný. Z didaktických důvodů jej **v žádném případě** nebudeme v tomto kurzu používat.

Pro úplnost je však potřeba se zmínit o dvou příkazech, které umožňují tyto patologické cykly vytvořit. Jednak proto, abychom se v praxi dokázali takovým konstrukcím vyhnout (a případně je odstranit), jednak proto, že v opravdu složitých a velmi výjimečných případech (nikoliv ve výuce!) je jejich použití možná omluvitelné. Vše zahrneme do obecnějšího rámce v následující sekci.

4.5 Nepodmíněný skok

Když čtenář tuto sekci zcela přeskóčí a nikdy se k ní nebude vracet, bude to pravděpodobně ku prospěchu jeho „programátorským“ schopnostem. Proč? Tato sekce se totiž zabývá těmi nejtemnějšími stránkami programování – jinými slovy, chcete-li do logiky programu hodit vidle, čtete dále...

Vraťme se na chvíli k programovacím prostředkům na strojové úrovni (jazyk symbolických instrukcí nebo přímo strojový kód). Tyto jazyky nemají žádné „inteligentní“ prostředky pro větvení, cyklus apod. Vše se musí řešit primitivními nástroji – instrukcemi pro podmíněné a nepodmíněné skoky. Těmto možnostem odpovídají i nástroje vývojových diagramů, kde lze kreslit jakékoliv spojnice a vazby, a dospět tak například k paskvilu znázorněnému na obr. 3. Uvědomme si ovšem, že se nacházíme nejen v jiné úrovni programovacích prostředků, ale také v jiné době. Tehdy se v operaci paměti o velikosti třeba 32 KB opravdu žádná hitparáda neodehrávala a tehdejší programy řešily z dnešního hlediska nepatrné pidiprobémy. Takže i velmi neohrabaně napsaný program bylo možné v reálném čase odladit a případně i udržovat.

Jak se tehdy vyjádřil cyklus? Byly k tomu potřeba dvě rekvizity: **podmíněný skok** a **nepodmíněný skok**. Oba porušují „přirozený“ běh programu, tedy sekvenční zpracování po sobě jdoucích instrukcí. Jakýkoliv skok je elementární instrukce umožňující „násilně“ změnit adresu, na níž se nachází následující instrukce. Podmíněný skok to udělá jen tehdy, nachází-li se na určeném místě smluvená

hodnota (obvykle v nějakém registru procesoru, často nula). Nepodmíněný skok to udělá vždycky. Cyklus se pak napsal v jazyce symbolických instrukcí touto posloupností:

```
A1: ...
    ...      (nějaké instrukce sloužící pro vyčíslení podmínky)
    JZ  A2    (je-li ve střadači nula, skoč na instrukci označenou A2)
    ...
    ...      (nějaké instrukce představující tělo cyklu)
    JMP A1    (skoč na instrukci označenou A1 - návrat k~podmínce)
A2: ...      (první instrukce za cyklem)
```

Každý řádek obsahuje jedinou instrukci. Adresy, na nichž se jednotlivé instrukce nacházely, bylo možné označit na daném řádku tzv. **návěstími** (např. **A1:**). Šlo o možnost provést na tuto adresu nějaký skok. Není asi potřeba zdůrazňovat, že i v poměrně krátkém programu se to takovými skoky jen hemžilo – ideální situace na „výrobu“ nejruznějších chyb. Se vzrůstající složitostí programů se tento systém stával zcela neúnosným, proto se v programovacích jazycích vyšší úrovně implementovaly příkazy nepotřebující žádná návěstí a žádné skoky. Jenže to všechno nešlo změnit ze dne na den. Proto i dnes máme ve vyšších programovacích jazycích tuto obskurní možnost – provést nepodmíněný skok. A to dokonce v několika variantách.

4.5.1 Nepodmíněné skoky v C++

Základním nepodmíněným skokem je příkaz **goto**. Jde o přímý ekvivalent strojové instrukce **JMP**. Za slovem **goto** se nachází návěstí příkazu, na nějž se skáče. Toto návěstí se může zapsat kamkoliv jako identifikátor zakončený dvojtečkou. Zmíněný cyklus znázorněný strojovými instrukcemi můžeme podobně zapsat i v C++ takto:

```
89 | A1: vysledek = a - pocet; //vyčíslení nějaké podmínky
90 |     if (vysledek == 0) goto A2; //obraz podmíněného skoku
91 |     ...
92 |     ... //příkazy těla cyklu
93 |     goto A1; //nepodmíněný skok zpět na podmínku
94 | A2: ... //příkaz následující za cyklem
```

Z příkladu je zřetelně vidět, že jazyk C původně vznikl jen jako „lepší“ strojový jazyk a bylo možné v něm programovat podobně jako v JSI. Podobnost se zápisem v JSI zdaleka není čistě náhodná.

Zatímco příkaz **goto** dnes již prakticky nikdo nevyužívá a každý programátor, který je schopen alespoň trochu přemýšlet v souvislostech, se mu vyhne, zůstávají v jazyce C++ ještě další dvě rekvizity představující téměř stejné zlo, ale není viditelné na první pohled:

1. Příkaz **break** – příkaz zmíněný již v souvislosti s příkazem **switch** – jde o nepodmíněný skok za konec příkazu **switch** nebo za konec jakéhokoliv příkazu cyklu (**while**, **do**, **for**). Umožňuje realizovat libovolný počet výskoků uvnitř těla. Když se podíváme na příklad s cyklem „rozloženým“ do instrukcí, pak příkaz **break;** je ekvivalentní příkazu **goto A2;**.

2. Příkaz **continue** – použitelný vždy jen uvnitř těla libovolného cyklu; zajišťuje nepodmíněný skok zpět na začátek cyklu. Je to ekvivalence příkazu **goto A1;**.

Z rozboru je patrné, že programátor používající dnes tyto dvě rekvizity, se vlastně pohybuje v temném počítačovém pravěku šedesátých let minulého století a vnáší do svých programů dávno vyčichlou a nefunkční černou magii obskurních kódů té doby.

V tomto textu se pochopitelně v žádném z příkladů *zásadně* žádný podobný paskvil vyskytovat nebude.

K nepodmíněným skokům je potřebné přidat ještě jeden případ konstrukce jazyka C++:

3. Příkaz **return** – okamžité ukončení podprogramu a nepodmíněný skok na první příkaz následující za voláním podprogramu.

Na rozdíl od předchozích se tomuto příkazu nelze vyhnout, je však potřeba mít na paměti jeho nebezpečnost: také porušuje přirozenou posloupnost příkazů, což vede k tomu, že veškeré příkazy zapsané za ním se *neprovedou!*

Proto je silně doporučeno příkaz **return** uvést vždy *nejvýše jednou*, a to *na úplném konci podprogramu*. Podprogramy budou diskutovány dále, tam se ještě k problému příkazu **return** vrátíme.

5 Elementární algoritmy

Řešení reálných úloh a hledání algoritmu je tvůrčí proces, v němž se uplatňuje mimo jiné i lidská intuice a nápaditost. Na druhé straně se však také jedná o rutinní záležitost, pokud řešíme úlohy, které už někdo před námi řešil také.

Je velmi efektivní, pokud přebereme již existující, kvalitní a úsporné řešení alespoň některých částí zpracovávané úlohy. Tím není myšleno porušování autorských práv a „vykrádání“ softwaru, ale o převzetí dobré myšlenky, neopakování chyb a již známých slepých uliček a využití ověřené a optimální cesty. Podobně například konstruktér při návrhu stroje již neřeší, jak má vypadat ložiskové uložení, protože ví, že tímto problémem se zabývala již řada konstruktérů mnoho let a dospělo se k možnostem, jejichž vlastnosti jsou známy a jsou taky známy všechny problémy, které při hledání tohoto řešení nastaly. Na druhé straně jsou ovšem situace, kdy ložiskové uložení známé konstrukce nevyhovuje a je nutné konstrukci upravit. I tehdy však konstruktér vychází ze známých postupů a těží ze zkušeností druhých.

Ukážeme si algoritmy pro řešení typických jednoduchých situací – pracovně je nazveme **elementární algoritmy**. Ukážeme si také jejich skládání. Můžeme pak vytvářet komplexnější úlohy, na nichž opět můžeme aplikovat podobné myšlenky, a dospět tak přímočarou cestou k cíli. Proces řešení úlohy není tak závislý na intuici, ale spíše se blíží dobře zvládnutému řemeslu.

5.1 Způsoby získání posloupnosti dat

První skupinkou elementárních algoritmů je řešení vstupu většího (variabilního) množství dat. Mohou nastat tři případy:

1. Víme, která hodnota je poslední. V tom případě musíme ještě rozhodnout, zda tato poslední hodnota je pouze umělou „zarážkou“, nebo je to hodnota, kterou musíme (nebo můžeme) zahrnout do zpracovávaných dat.
2. Víme z jiných informací, že nastal konec dat – například testujeme konec souboru, konec seznamu nebo jiné struktury.
3. Víme, kolik hodnot máme zpracovat. Tuto informaci můžeme přepočítat, vypočítat nebo znát z podstaty věci (například zpracování denních průměrných teplot za poslední rok bude obsahovat 365 nebo 366 údajů).

Lze si ještě představit, že mohou nastat různé kombinace, například víme, že hodnot je více než potřebujeme, hodnoty mají více sekcí, hodnoty mohou být složeny z dvojic údajů atd. Vše lze pak vyřešit také kombinacemi algoritmů, které pro ty základní tři případy vytvoříme.

Nyní si jednotlivé situace rozebereme.

5.1.1 Posloupnost zakončená nezahrnutelnou hodnotou

Typická úloha tohoto typu: Je připravena posloupnost čísel představujících měsíční platy zaměstnanců oddělení zakončená hodnotou `-99999`. Máme zjistit celkový měsíční mzdový náklad pro oddělení. Kdybychom tu závěrečnou hodnotu zahrnuli do zpracovávaných dat, dojde k poškození výsledku, jde v tomto případě o nezahrnutelnou hodnotu.

Získání posloupnosti hodnot v této situaci musí být nutně řešeno cyklem `while`. Cyklus pracuje tak dlouho, dokud se získávané hodnoty v posloupnosti nerovnájí smluvené koncové hodnotě. Předpokládejme, že je k dispozici proměnná `Hodnota`, v níž se bude nacházet ta hodnota zpracovávané posloupnosti, která je právě na řadě, a dále máme konstantu `KH` představující koncovou hodnotu. Můžeme pak psát:

```
95 //získání prvního členu posloupnosti a vložení do proměnné Hodnota
96 while (Hodnota != KH) {
97     //zde se obsah proměnné Hodnota může zpracovat
98     //získání dalšího členu posloupnosti a uložení do proměnné Hodnota
99 }
```

Všimněte si, že podmínka, kterou potřebujeme napsat v cyklu `while`, porovnává aktuální člen posloupnosti (v proměnné `Hodnota`) s koncovou hodnotou. Abychom mohli tuto podmínku napsat, musíme zajistit, že v každém okamžiku bude proměnná `Hodnota` správně naplněna. Musíme tedy získávání členů zpracovávané posloupnosti psát dvakrát: první člen získáváme před cyklem, uvnitř těla cyklu pak musíme již získaný člen *napřed* zpracovat a teprve *pak* (úplně na konci těla cyklu) získat další člen. Jeho hodnota bude vzápětí testována při zahájení nového průchodu cyklem.

Obecné operace naznačené pomocí komentářů pak v konkrétní situaci nahradíme příslušným příkazem (nebo posloupností příkazů). Získání členu můžeme provést čtením ze standardního vstupu, čtením ze souboru, výpočtem atd. Princip tohoto algoritmu však zůstává stejný. Když například aplikujeme postup na zmíněnou úlohu s platy zaměstnanců oddělení, můžeme upřesnit:

```
100 int Hodnota;
101 cin >> Hodnota; //získání první hodnoty
102 while (Hodnota != -99999) {
103     //zde se budou hodnoty sčítat
104     cin >> Hodnota; //získání další hodnoty
105 }
```

5.1.2 Posloupnost zakončená zahrnutelnou hodnotou

Vyjdeme z předchozího algoritmu. Je-li poslední hodnota zpracovatelná s ostatními, není nutné ji oddělovat od ostatních, cyklus musí proběhnout alespoň jednou (minimálně s tou koncovou hodnotou) a čtení hodnot nemusí být zapisováno nadvakrát. Cyklus, jehož tělo se provede alespoň jednou, pak bude `do-while`.

```

106 | do {
107 |     //získání členu posloupnosti a uložení do proměnné Hodnota
108 |     //zde se obsah proměnné Hodnota může zpracovat
109 | } while (Hodnota != KH);

```

Jako příklad můžeme použít předchozí úlohu s platy zaměstnanců, změníme-li koncovou hodnotu na nulu. Nula neovlivní součet, může být tedy zahrnuta do zpracování. Aplikujeme obecný tvar:

```

110 | do {
111 |     cin >> Hodnota;
112 |     //zde se budou hodnoty sčítat
113 | } while (Hodnota != 0);

```

5.1.3 Zjištění konce dat jinými prostředky

Tato situace se pravděpodobně nejvíce blíží nejčastějšímu způsobu získávání dat. Jde o případ, kdy jsou data ve struktuře, na jejíž stav se můžeme dotázat, lze zjišťovat pozici čtení nebo množství dat, která ještě nebyla zpracována.

Zpočátku se omezíme pouze na variantu, kdy získáváme data ze vstupního proudu (obvykle standardní vstup) a můžeme zde zjistit, zda jsou nějaká data ještě k dispozici. Podobně jako v předchozím algoritmu pak provedeme získání jedné hodnoty a tu následně zpracujeme. Obecný tvar algoritmu je následující:

```

114 | while (/*jsou ještě připravena data*/) {
115 |     //získání členu posloupnosti a uložení do proměnné Hodnota
116 |     //zde se obsah proměnné Hodnota může zpracovat
117 | }

```

Podmínka, zjišťující stav vstupního proudu, může být realizována několika způsoby. Zmíníme se o dvou využitelných pro vstupní proud `cin`. Jedním z nejpoužívanějších je využití výstupní informace čtecího příkazu `cin >> Hodnota`. Pokud je čtení úspěšné, příkaz samotný vydá kladnou informaci, kterou můžeme v podmínce využít. V opačném případě se čtení neprovede a cyklus lze ukončit. Čtecí příkaz dělá současně dvě operace – jak vyhodnocení podmínky, tak samotné čtení. Dva prvky obecného algoritmu se tak sloučí do jednoho.

Druhým způsobem zjištění je například využití metody, která se přímo dotazuje na stav vstupního proudu `cin.eof()`. Tento stav však může být ovlivněn existencí bílých znaků (oddělovačů), takže vstupní proud je neprázdný, ale neobsahuje již relevantní data, což znamená, že následný pokus o získání hodnoty skončí neúspěchem.

Aplikujme nyní obecný tvar opět na úlohu s platy zaměstnanců, jejíž formulace bude mírně změněna: Na standardním vstupu jsou připraveny hodnoty představující měsíční platy zaměstnanců určitého oddělení. Zjistěte celkový měsíční mzdový náklad daného oddělení. Uvedeme pouze variantu s využitím výstupní informace čtecího příkazu.

```
118 | int Hodnota;  
119 | while (cin >> Hodnota) //test konce vstupu a zároveň čtení  
120 |     //zde se budou hodnoty sčítat
```

5.1.4 Počet hodnot je dopředu znám

V situaci, kdy máme k dispozici dopředu informaci o požadovaném počtu zpracovávaných hodnot, můžeme s výhodou využít počítaného cyklu. Algoritmus je obdobou předchozích algoritmů, v cyklu opět potřebujeme dva kroky – získání hodnoty a její zpracování.

Informace o počtu zpracovávaných údajů může být dopředu dána povahou dat (víme například, že na oddělení pracuje 25 lidí a bude to zapsáno jako konstanta), nebo může být uvedena jako první údaj v posloupnosti. V tom případě je nutné ještě před cyklem tuto informaci získat.

Obecný tvar algoritmu je následující:

```
121 | //získání informace o počtu dat N  
122 | for (int RP=1; RP<=N; RP++) {  
123 |     //získání členu posloupnosti a uložení do proměnné Hodnota  
124 |     //zde se obsah proměnné Hodnota může zpracovat  
125 | }
```

Aplikujme tento obecný tvar na úlohu s platy zaměstnanců s tímto zněním: Na vstupu se nachází celočíselná hodnota udávající počet zaměstnanců oddělení. Za ní je připravena posloupnost měsíčních platů jednotlivých zaměstnanců. Určete celkový měsíční mzdový náklad daného oddělení.

```
126 | int N;  
127 | int Hodnota;  
128 | //získání informace o počtu dat N:  
129 | cin >> N;  
130 | for (int RP=1; RP<=N; RP++) {  
131 |     cin >> Hodnota;  
132 |     //zde se budou hodnoty sčítat  
133 | }
```

5.2 Extrémy

Úlohy, patřící do této skupinky se týkají např. nalezení maximálního/minimálního čísla z řady, ale i jakéhokoli jiného atributu v extrémní hodnotě (nejdelší řetězec, válec s největším objemem, osoba s nejnižším platem atd.).

Obecně lze předpokládat, že zpracováváme řadu hodnot, u každé hodnoty sledujeme určitou vlastnost a z těchto vlastností pak hledáme extrémní hodnotu. V této chvíli je podstatné pouze to, že zmíněnou řadu zpracováváme v nějakém cyklu, je ale lhostejné, o jaký typ cyklu jde.

Variantou těchto úloh je nalezení *pozice* extrému. Nezajímá nás například, které číslo z řady je nejmenší, ale na které pozici (v jakém pořadí) se nachází. Případně nás zajímá datový celek, jehož některý z atributů má extrémní hodnotu, například číslo, jehož ciferný součet je největší.

Principem algoritmu je zjišťování u každé hodnoty z řady, zda překonává hodnotu, kterou v dané chvíli považujeme za extrémní (tedy je větší než dosavadní maximum nebo menší než dosavadní minimum). Pro dosavadní extrém máme jednu proměnnou (`extrem`), pro zpracovávanou hodnotu pak druhou proměnnou (`H`). Výsledek můžeme použít až po skončení cyklu, jak je uvedeno v příslušné poznámce na řádce 140.

Základní tvar algoritmu je následující:

```

134 /*typ dat*/ H, extrem;
135 extrem = /*opačný extrém nebo libovolná hodnota zkoumané řady*/;
136 //začátek cyklu
137     //získání hodnoty do proměnné H
138     if (H /*překonává*/ extrem) extrem = H;
139 //konec cyklu
140 //použití výsledku v proměnné extrem

```

Před začátkem cyklu je nutné nastavit počáteční hodnotu do proměnné `extrem`. Musíme přitom zajistit, aby test vztahu mezi běžnou hodnotou a extrémem uvnitř cyklu fungoval správně. Máme dvě možnosti:

- Počáteční hodnotou je opačný extrém. V tom případě se při porovnání *jakékoliv* hodnoty z řady vždy tato hodnota stává dosavadním extrémem. Nevýhodou někdy je, že nevíme, která hodnota tvoří opačný extrém.
- Počáteční hodnotou je libovolná hodnota zkoumané řady. V tom případě máme jistotu, že nastavení je správné. Pokud je zvolená hodnota současně i extrémem, pak sice žádný test uvnitř cyklu nebude platný, ale výsledek bude správný.

Aplikace algoritmu na konkrétní případ: Je potřebné zjistit, která hodnota ze vstupní řady desetinných čísel je nejmenší.

```

141 float H, extrem;
142 extrem = 1e10; //předpokládáme, že to je dostatečně velká hodnota
143 //začátek cyklu
144     //získání hodnoty do proměnné H
145     if (H < extrem) extrem = H;
146 //konec cyklu
147 //použití výsledku v proměnné extrem

```

Varianta algoritmu zjišťující pozici extrému potřebuje ještě dvě proměnné: jednu proměnnou, v níž se uchovává pozice, na níž byl nalezen extrém, a druhou, v níž je udržována aktuální pozice. Obecný tvar rozšíříme:

```
148 /*typ dat*/ H, extrem;  
149 /*typ pozice*/ bezna, extremni;  
150 extrem = /*opačný extrém nebo libovolná hodnota zkoumané řady*/;  
151 bezna = /*pozice začátku*/;  
152 //začátek cyklu  
153     //získání hodnoty do proměnné H  
154     if (H /*překonává*/ extrem) {  
155         extrem = H;  
156         extremni = bezna; //uchování pozice extrému  
157     }  
158 //konec cyklu  
159 //použití výsledku v proměnné extremni
```

Proměnná pro pozici extrému nemá přiřazenou počáteční hodnotu – v tomto tvaru algoritmu ji nepotřebujeme, protože musí vždy proběhnout alespoň jedno přiřazení uvnitř cyklu (hned první hodnota řady překoná hodnotu opačného extrému).

V uvedené úloze bychom mohli zjišťovat, jaké pořadí má minimální číslo:

```
160 float H, extrem;  
161 unsigned int bezna=0, extremni;  
162 extrem = 1e10; //předpokládáme, že to je dostatečně velká hodnota  
163 //začátek cyklu  
164     //získání hodnoty do proměnné H  
165     bezna++; //aktualizace běžné pozice  
166     if (H < extrem) {  
167         extrem = H;  
168         extremni = bezna;  
169     }  
170 //konec cyklu  
171 //použití výsledku v proměnné extremni
```

Pokud máme libovolnou hodnotu zpracovávané řady k dispozici ještě před cyklem, můžeme této skutečnosti využít pro nastavení počáteční hodnoty proměnné `extrem`. Pokud navíc sledujeme pozici, musíme rovněž nastavit i počáteční hodnotu proměnné `extremni`.

5.3 Součty, součiny

Jde o zpracování řady hodnot (tj. cyklický algoritmus), kdy zjišťujeme jejich celkový součet (součin nebo podobnou veličinu). K řešení potřebujeme proměnnou (`celek`), do níž v průběhu zpracování přidáváme postupně další získávané hodnoty. Přidáváním zde můžeme myslet součet, součin, rozdíl, spojování do seznamu, zřetězení do řetězce apod. Počáteční hodnota sumární proměnné musí být nastavena ještě před zahájením cyklu, a to na hodnotu, která je vzhledem k dané operaci počáteční.

Pro součet je to nula, pro součin jednička, pro zřetězení prázdný řetězec atd. Základní tvar můžeme formulovat takto:

```

172  /*typ dat*/ H, celek = /*počáteční souhrnná hodnota*/;
173  //začátek cyklu
174      //získání hodnoty do proměnné H
175      celek = celek /*operace přidání*/ H;
176  //konec cyklu
177  //použití výsledku v proměnné celek

```

Aplikujme obecný tvar na následující úlohu: Je potřebné sečíst všechny celočíselné hodnoty zpracovávané řady.

```

178  long int H, celek = 0;
179  //začátek cyklu
180      //získání hodnoty do proměnné H
181      celek += H; //zkrácený zápis: celek = celek + H;
182  //konec cyklu
183  //použití výsledku v proměnné celek

```

Podoba algoritmu může být ještě mírně modifikována v případě, že operace přidání není komutativní. Pak je pochopitelně zásadní rozdíl ve výsledku, napíšeme-li `celek = celek OPERACE H;` nebo `celek = H OPERACE celek;`. Této skutečnosti zároveň můžeme i využít, protože tím v podstatě měníme pořadí zpracovávaných dat.

Jako příklad můžeme uvést nekomutativní operaci zřetězení. Pro datový typ `string` existuje operace zřetězení zapisovaná operátorem `+`. Uvažujme, že zpracováváme řadu jednotlivých znaků, z nichž máme vytvořit jeden řetězec. Můžeme psát:

```

184  char H;
185  string celek = ""; //prázdný řetězec
186  //začátek cyklu
187      //získání jednoho znaku do proměnné H
188      celek = celek + H; //nový znak se připojuje na konec
189  //konec cyklu
190  cout << celek << endl;

```

Když bychom na řádce 188 ovšem napsali: `celek = H + celek;`, dostaneme na výstupu řetězec se znaky v opačném pořadí, než byly získávány.

5.4 Počty, výběry

Úlohy tohoto typu zjišťují počet prvků ve zpracovávané řadě. Speciálnějším případem je zjišťování počtu prvků vybraných podle nějaké podmínky (spojení algoritmu pro výběry). Pro uchování po-

čtu prvků potřebujeme celočíselnou proměnnou pro nezáporné hodnoty, jejíž počáteční hodnotou je nula. V cyklu zpracovávajícím hodnoty pak s každou hodnotou zvyšujeme počet o jedna.

Obecný tvar algoritmu:

```
191  /*typ dat*/ H;  
192  unsigned int pocet = 0;  
193  //začátek cyklu  
194      //získání hodnoty do proměnné H  
195      pocet++; //zvýšení počtu o jedna  
196  //konec cyklu  
197  //použití výsledku v proměnné pocet
```

Podobným jednoduchým algoritmem je algoritmus pro jakékoliv zpracování vybraných hodnot. Výběr je proveden uvnitř cyklu zápisem výběrové podmínky. Můžeme zapsat jeho obecný tvar:

```
198  /*typ dat*/ H;  
199  //začátek cyklu  
200      //získání hodnoty do proměnné H  
201      if (/*podmínka výběru H*/) /*potřebné zpracování*/  
202  //konec cyklu
```

Aplikaci uvedených algoritmů můžeme demonstrovat příkladem: Je potřebné zjistit počet čísel ve zpracovávané řadě desetinných čísel.

```
203  double H;  
204  unsigned int pocet = 0;  
205  //začátek cyklu  
206      //získání hodnoty do proměnné H  
207      pocet++; //zvýšení počtu o jedna  
208  //konec cyklu  
209  //použití výsledku v proměnné pocet
```

Variantou této úlohy může být zjištění počtu čísel zpracovávané řady, která přísluší do intervalu $\langle 10; 20 \rangle$:

```
210  double H;  
211  unsigned int pocet = 0;  
212  //začátek cyklu  
213      //získání hodnoty do proměnné H  
214      if (H >= 10 and H<20) pocet++; //zvýšení počtu o jedna  
215  //konec cyklu  
216  //použití výsledku v proměnné pocet
```

5.5 Metoda skládání elementárních algoritmů

Představili jsme některé základní algoritmy, z nichž lze sestavit větší celky. Postup řešení nějaké úlohy tedy můžeme rozložit na části, které povedou na elementární algoritmy, jejichž řešení již známe a nemusíme je znovu objevovat. Rovněž při zápisu programu můžeme postupovat po logických celcích, což umožňuje zrychlit řešení, vyhnout se typickým chybám a usnadnit případné další modifikace.

Metodu ukažme na příkladu:

Na standardním vstupu se nachází řada nezáporných celočíselných hodnot. Vypište na výstup všechny hodnoty, jejichž ciferný součet je větší než 5.

Postup můžeme rozdělit na tři následující kroky:

1. V zadání úlohy je potřebné rozpoznat jednotlivé elementární algoritmy.
2. Rozpoznané algoritmy je potřebné aplikovat na konkrétní případ.
3. Aplikované algoritmy je potřebné složit do výsledného programu.

První krok – které elementární algoritmy jsou v zadání obsaženy? Jde o tyto čtyři: Z věty „Na standardním vstupu se nachází řada celočíselných hodnot.“ vyplývá, že potřebujeme algoritmus pro vstup hodnot, jejichž počet ani koncovou hodnotu neznáme, konec dat budeme zjišťovat jiným způsobem. Algoritmus byl rozebrán v sekci 5.1.3.

Z druhé věty zadání vyplývá, že potřebujeme zpracovat vybrané hodnoty. Výběrovou podmínkou je „ciferný součet větší než 5“, kterou aplikujeme do algoritmu pro výběr – viz sekce 5.4.

Třetím algoritmem je výpočet ciferného součtu – jde o algoritmus pro součet řady hodnot, tentokrát jsou tou řadou jednotlivé číslice zpracovávaného čísla (viz 5.3).

Čtvrtým algoritmem je zpracování řady číslic nezáporného celého čísla. Jde o algoritmus zpracování řady, u níž máme alespoň jednu hodnotu (každé číslo má alespoň jednu číslici) a známe koncovou hodnotu zahrnutelnou do zpracování – odebíráme-li postupně z čísla jednotlivé číslice, končíme získáním nuly – všechny číslice byly již odebrány (viz 5.1.2).

Ve druhém kroku aplikujeme rozpoznané algoritmy. Algoritmus pro čtení celých čísel do konce vstupu (ALG1):

```
217 unsigned long int H;  
218 while (cin >> H) {  
219     //zpracování hodnoty H  
220 }
```

Algoritmus pro výpis vybraných hodnot (ALG2):

```
221 //začátek cyklu  
222 //zjištění ciferného součtu H
```

```

223 |     if (/*ciferný součet H*/ > 5) cout << H << ", ";
224 | //konec cyklu

```

Pro účel ciferného součtu si uložíme obsah proměnné `H` do pomocné proměnné `cislo`, abychom z ní mohli postupně „odřezávat“ jednotlivé číslice, a nepoškodili tak proměnnou `H`, kterou budeme potřebovat později pro výpis. Algoritmus pro získání součtu jednotlivých cifer (ALG3):

```

225 | unsigned int soucet = 0;
226 | //začátek cyklu
227 |     soucet += cislo % 10; //získání číslice a přičtení do součtu
228 |     //odstranění zpracované číslice z čísla H
229 | //konec cyklu

```

Získání řady číslic z proměnné `cislo` (ALG4):

```

230 | unsigned long int cislo; //cislo je stejného typu jako H
231 |     cislo = H;
232 |     do {
233 |         //zpracování číslice
234 |         cislo = cislo / 10; //odstranění zpracované číslice
235 |     } while (cislo != 0); //ještě jsou nenulové číslice

```

Nyní máme všechny čtyři algoritmy v konkrétní podobě, zbývá třetí krok – složení do jednoho celku. Tento krok je znázorněn na obr. 8.

ALG1:

```

unsigned long int H;
while (cin >> H) {
    // zpracování hodnoty H
}

```

ALG2:

```

// začátek cyklu
{
    // zjištění ciferného součtu H
    if (/*ciferný součet H*/ > 5) cout << H << ", ";
    // konec cyklu
}

```

ALG4:

```

unsigned long int cislo;
cislo = H;
do {
    // zpracování číslice
    cislo = cislo / 10;
} while (cislo != 0);

```

ALG3:

```

{
    unsigned int soucet;
    soucet = 0;
    // začátek cyklu
    soucet += cislo % 10;
    // konec cyklu
}

```

Obrázek 8: Složení čtyř elementárních algoritmů do jednoho celku

Ve výsledném programu pak ještě můžeme provést drobné úpravy – například přesuneme všechny deklarace do jednoho místa kvůli přehlednosti. Doplníme hlavní funkci `main`, připojení knihovny `iostream` a otevření jmenného prostoru `std`. Získáváme pak následující tvar:

```

236 | #include <iostream>
237 | using namespace std;

```

```

238
239 int main (){
240     unsigned long int H, cislo; //deklarace do jednoho místa
241     unsigned int soucet;
242     while (cin >> H) {
243         cislo = H;
244         soucet = 0;
245         do {
246             soucet += cislo % 10;    //získání číslice a součet
247             cislo /= 10;            //odstranění zpracované číslice
248         } while (cislo != 0);        //ještě jsou nenulové číslice
249         if (soucet > 5) cout << H << ", ";
250     }
251     return 0;
252 }

```

Tak, jako jsme skládali algoritmy v obrázku, je možné postupovat i při zápisu zdrojového textu programu. Je velmi efektivní, pokud v každém okamžiku zapisujeme *vždy jeden elementární algoritmus*. Vyhnete se tím všem možným chybám pramenícím z toho, že řešíme více věcí současně – něco zapomeneme, něco uděláme jinak atd. Nepíšeme tedy program „od začátku do konce“ tak, jak ho zpracovává překladač, ale po logických celcích – každý editor nám umožňuje vepisovat do libovolného místa nové řádky.

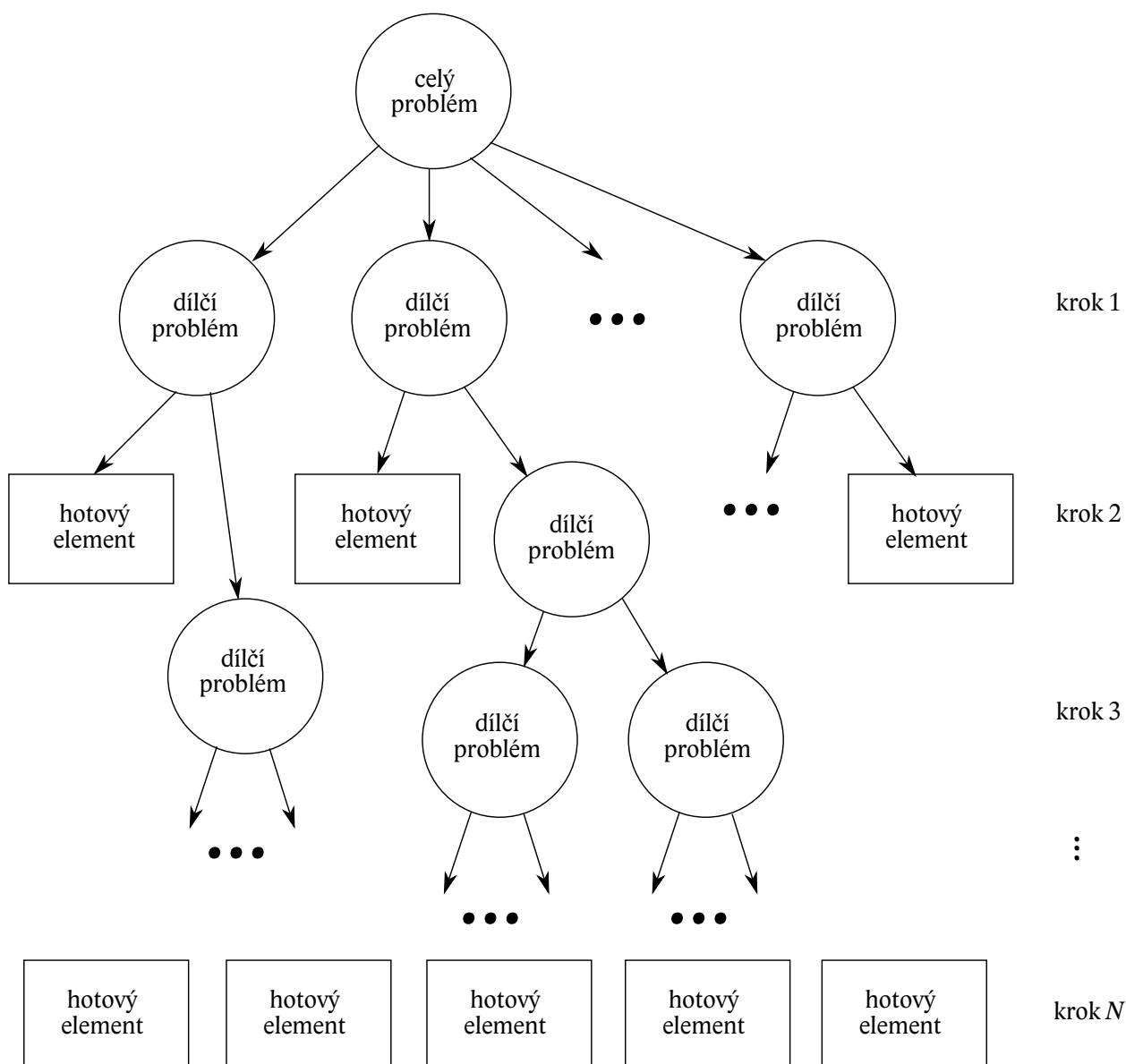
Je samozřejmé, že po získání určité rutiny je možné (zvlášť v jednoduchých případech obsažených v tomto kurzu) zapisovat rovnou celý program. Myšlenkově bychom ale vždy měli dokázat rozdělit celé dílo na elementy, které jsme schopni obsáhnout a beze zbytku a bez chyb implementovat.

K algoritmům uvedeným v této kapitole budeme později přidávat další, princip psaní programů však zůstává stejný.

5.6 Princip shora dolů

Konkrétní metoda skládání elementárních algoritmů je příkladem obecného principu, který lze využít v téměř jakékoliv fázi psaní jednoduchých i složitých programů. Tento princip je znám pod několika názvy – tím nejznámějším, nejúdernějším a bezpochyby nejstarším je starořímské *rozděl a panuj*. V programování se používá název **shora dolů**, nebo též princip **dekompozice**. Obojí znamená totéž: dekompozice (rozdělení) celého problému na menší logické části, jejichž řešení už přímo známe, nebo je jednodušší a snáze uchopitelné. Části, které ještě nejsou elementární, pak v dalším kroku rozložíme na menší atd. Schematicky celou metodu znázorňuje obr. 9.

Postupovat lze i opačným způsobem: tzv. **syntéza** (skládání) je metoda, v níž postupně skládáme elementární prvky do větších celků, až „složíme“ celý program. Tuto metodu známe i pod názvem **zdola nahoru**.



Obrázek 9: Princip postupné dekompozice celého problému na elementární prvky

Čistá metoda shora dolů a podobně i čistá metoda zdola nahoru se v praxi používá málokdy, často využíváme *kombinaci* obou přístupů. Při dekompozici se postupné rozdělování problému snažíme směřovat na celky, které už jsou někde k dispozici, i když to nejsou vysloveně elementy (například knihovní funkce). Podobně se dopředu („naslepo“) programují části (moduly), které nejsou v dané chvíli potřeba, ale je předpoklad, že se z nich budou nějaké programy skládat.

5.7 Trasování

Z procesu sestavování konkrétního programu se nyní dostáváme k praktické realizaci. Předpokládáme, že máme hotový spustitelný program. Běžně potřebujeme zjišťovat, zda pracuje podle oček-

kávání. Musíme v tom případě analyzovat jeho činnost. Při použití odpovídajících algoritmů a jejich správné implementaci je pravděpodobné, že náš program bude splňovat všechna očekávání, může se v něm však vyskytnout jakákoliv chyba, kterou musíme najít a opravit. Také můžeme podobným způsobem analyzovat i cizí programy a usuzovat, jak přesně fungují.

K této analýze se často používá technika **trasování** (sledování). Programová vybavení s integrovaným vývojovým prostředím používaná k vytváření programů mívají nástroje na trasování a zobrazují potřebné informace. Princip trasování můžeme ale aplikovat i bez takového vybavení, a to konstrukcí **trasovací tabulky** třeba na papíře.

Co obsahuje trasovací tabulka? Jde o záznam simulace činnosti zkoumaného programu z hlediska obsahů proměnných a případných vstupů a výstupů. V jednoduchém případě vytvoříme pro každou proměnnou jeden sloupec tabulky, do něhož postupně zaznamenáváme hodnoty této proměnné a jejich změny. V tabulce je pak vidět i historie změn, ze které můžeme usoudit, jak program pracuje. Zaznamenáváme i data na vstupu a jejich zpracování, vypisujeme taktéž záznamy o výstupních datech.

Na jednoduchém příkladu si ukážeme trasování se zvolenými daty.

Příklad: Vstup obsahuje řadu celočíselných hodnot. Zjistíte pořadí největšího lichého čísla a celkový součet všech sudých čísel vstupní řady.

Hotový program má následující možný tvar:

```

253 #include <iostream>
254 using namespace std;
255
256 int main (){
257     long int H, MaxLH=-1000000, Soucet=0;
258     unsigned int PozMaxLH, Pozice=0;
259     while (cin >> H) {
260         Pozice++;
261         if (H % 2 != 0) {    //liché číslo
262             if (H > MaxLH) {
263                 MaxLH = H;
264                 PozMaxLH = Pozice;
265             }
266             } else Soucet += H;
267     }
268     cout << "Největší liché číslo je na pozici " << PozMaxLH
269         << ", součet sudých čísel je " << Soucet << endl;
270     return 0;
271 }
```

Zvolíme vstupní data, např. `-101 42 13 -20 16 -11`

Vytvoříme trasovací tabulku: bude mít 5 sloupců pro pět proměnných (nyní oceňujeme, že deklarace všech proměnných jsou v jednom místě programu) a do těchto sloupců uvedeme počáteční hodnoty. Pokud není při deklaraci přiřazena do proměnné žádná hodnota, je nedefinovaná (nevypĺňujeme nic). Rovněž si poznamenejme stav vstupních dat – znakem svislícíko | budeme pracovně označovat místo, odkud se bude číst – to se bude v průběhu zpracování posouvat.

Počáteční tvar trasovací tabulky:

Stav vstupu: | -101 42 13 -20 16 -11

H	MaxLH	Soucet	PozMaxLH	Pozice
	-1 000 000	0		0

Celý proces trasování spočívá v „simulaci“ činnosti programu. Jdeme od začátku, od prvního příkazu, do tabulky zaznamenáváme veškeré změny hodnot a případně posouváme bod čtení ve vstupních datech. V našem příkladu se první příkaz nachází na řádce 259 – provede se kontrola existence vstupního proudu a přečte se první hodnota. Pokud je kontrola úspěšná (to víme, že je, protože vstup obsahuje data), vstoupíme do cyklu a provedeme příkazy v jeho těle. Po skončení prvního průchodu tělem cyklu dostáváme následující stav trasovací tabulky:

Stav vstupu: -101 | 42 13 -20 16 -11

H	MaxLH	Soucet	PozMaxLH	Pozice
	-1 000 000	0		0
-101	-101		1	1

Každá nová hodnota v daném sloupci přemazává hodnotu předchozí (to odpovídá počítačové realitě – nová hodnota vložená do proměnné přemazává hodnotu původní). Okamžitý obsah proměnných je dán posledním záznamem v jejich sloupci. Hodnoty můžeme zapisovat po řádcích například tak, jak to uvádíme v tomto případě, tj. jeden průchod tělem cyklu je jeden řádek, ale stejně tak můžeme novou hodnotu zapisovat těsně pod předchozí, to spíše záleží na tom, co bude pro vlastní potřebu přehlednější.

Po provedení příkazů v těle cyklu se opět vracíme na vyčíslení podmínky. Opět přečteme hodnotu ze vstupu (tentokrát 42) a posuneme pozici čtení, vykonáme příkazy těla cyklu (podmínka byla splněna) atd. Totéž provádíme tak dlouho, dokud se nedostaneme na poslední příkaz celého programu. Dostaneme následující výslednou trasovací tabulku:

Stav vstupu: -101 42 13 -20 16 -11 |

H	MaxLH	Soucet	PozMaxLH	Pozice
	-1 000 000	0		0
-101	-101		1	1
42		42		2
13	13		3	3
-20		22		4
16		38		5
-11				6

Výstup: Největší liché číslo je na pozici 3, součet sudých čísel je 38

6 Časová a prostorová složitost algoritmů

6.1 Složitost algoritmů

Při řešení programátorských úloh je vhodné mít nástroj, kterým lze porovnat efektivitu a rychlost provádění jednotlivých algoritmů. Pro tento účel byl zaveden pojem **složitost algoritmu**.

Složitost určuje operační náročnost algoritmu tak, že sleduje, jak se bude chování algoritmu měnit v závislosti na změně velikosti nebo množství vstupních dat. V této souvislosti nás ovšem nezajímají konkrétní velikosti spotřebovaných zdrojů, ale *funkce závislosti* spotřeby zdrojů na vstupních datech.

Složitost často slouží jako kritérium pro porovnávání kvality programů a algoritmů a ke zjištění jejich praktické použitelnosti.

Časová složitost se odvíjí od počtu provedených operací v závislosti na počtu vstupních dat.

Prostorová složitost se odvíjí od velikosti datových struktur, které algoritmus využívá, v závislosti na počtu vstupních dat.

Složitost se vyjadřuje jako určitá (jednoduchá) matematická funkce, popisující závislost daného parametru (paměťového prostoru nebo spotřebovaného výpočetního času) na množství vstupních dat.

6.2 Způsob zápisu

Složitost se obvykle zapisuje pomocí tzv. O-notace jako $O(f(N))$ – složitost je funkcí vstupních dat. Jde o *třídu složitosti* a typ funkční závislosti, nikoli o přesné vyjádření. Je to tzv. **asymptotická složitost**, která vyjadřuje asymptotické chování funkce.

Při malém množství vstupních dat nejsou rozdíly mezi třídami složitostí tolik patrné. Při větších objemech dat (větší N) jsou aditivní a multiplikativní konstanty vzhledem k N natolik nevýznamné, že je můžeme prakticky zanedbat. Nemíí důležité, zda je složitost vyjádřena například jako $O(100N)$, $O(N + 1000)$ nebo $O(100N + 1000)$, všechny tyto případy mají charakter lineární funkce $O(N)$.

6.3 Varianty složitosti

Složitost závisí také na zpracovávaných datech a podle jejich charakteru se rozlišuje horní, dolní a střední (průměrný) odhad složitosti algoritmu.

Horní odhad složitosti algoritmu udává složitost algoritmu v nejhorším případě – algoritmus probíhá asymptoticky stejně rychle nebo rychleji než funkce $f(N)$. Nejčastěji se používá toto vyjádření, protože zahrnuje všechny, i ty nejhorší případy. Označuje se jako $(O(N))$.

Dolní odhad složitosti udává ideální složitost algoritmu, která ovšem nastává jen pro určité případy vstupních dat. Označuje se jako $\Omega(f(N))$. Známe-li dolní odhad složitosti, je zřejmé, že danou úlohu nelze řešit efektivnějším algoritmem (který by spotřeboval méně času nebo prostoru). Algoritmus

probíhá asymptoticky stejně rychle nebo pomaleji než funkce $f(N)$. Dokázat, že neexistuje lepší algoritmus, je ale těžší než odhadnout horší složitost.

Průměrná (očekávaná) složitost je střední hodnota složitosti při nějakém (náhodném) rozložení vstupních dat, algoritmus probíhá asymptoticky stejně rychle jako funkce $f(N)$. Označuje se jako $\Theta(f(N))$. Pokud algoritmus dosahuje horní složitosti jen zřídka, pak o složitosti algoritmu lépe vypovídá průměrná složitost.

6.4 Stanovení složitosti

Stanovení složitosti algoritmu lze provést *experimentálně* nebo *analýzou zdrojového textu* algoritmu. Při experimentálním stanovení se měří spotřebovaný čas (prostor) pro vhodné hodnoty nezávisle proměnné. Stanovení provádíme tak, aby požadovaný charakter bylo možné interpolovat i extrapolovat z tabulky získaných hodnot.

Při analýze zdrojového textu vyhledáváme cykly, jejichž počet opakování závisí na nezávisle proměnné (množství vstupních dat), a sledujeme těla takových cyklů.

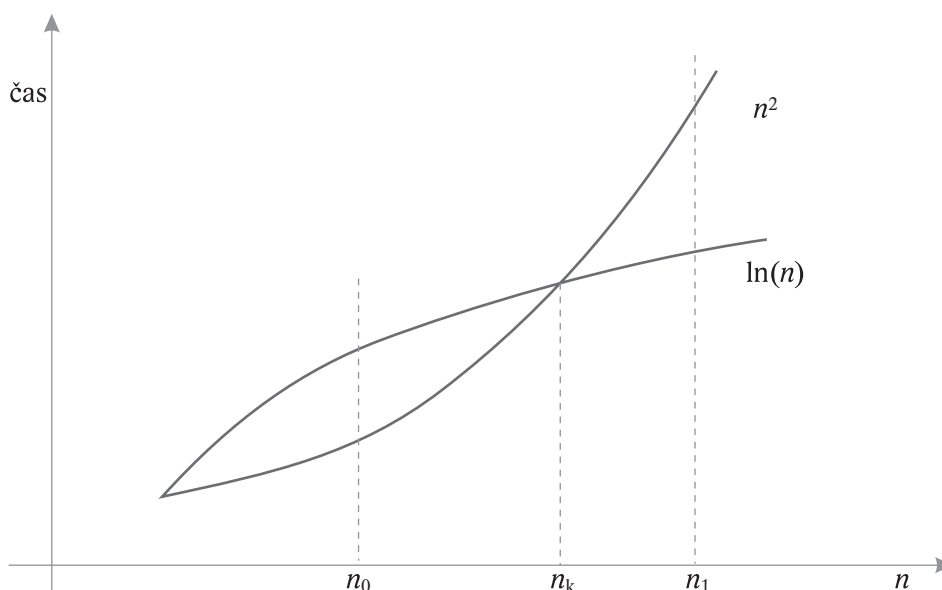
Při stanovení časové složitosti nás zajímá počet operací, který je dán počtem opakování cyklu a implementační konstantou. Při vnoření cyklů se počty operací násobí. Stejně tak sledujeme rekurzivní volání – rekurze závislá na množství vstupních dat je ekvivalentní cyklu.

Stanovení prostorové složitosti znamená analýzu prostoru, který bude zabrán:

- pro statické proměnné (hledáme v deklaracích) – statické proměnné se obvykle na charakteru složitosti projevují jen konstantou,
- pro dynamické proměnné (hledáme alokace paměti v příkazech, zejména v cyklech) – cyklické alokace jsou nejpodstatnější,
- pro rekurzivně zabírané prostory (parametry volané hodnotou a lokální proměnné rekurzivních podprogramů) – podobně jako u cyklů hledáme počet rekurzivních zanoření v závislosti na množství vstupních dat.

6.5 Porovnání složitostí programů

Stanovení složitosti daného algoritmu nebo celého programového modulu je z hlediska celkového hodnocení kvality díla velmi důležité. Uvedme jednoduchý příklad – dva algoritmy mající kvadratickou časovou složitost a logaritmickou časovou složitost. Situace je znázorněna na obrázku 10.



Obrázek 10: Průběh kvadratické a logaritmické funkce u dvou algoritmů

Nyní si představme, že neznalý programátor napíše algoritmus s kvadratickou složitostí a testuje jeho funkci. Použije při tom testovací data o velikosti n_0 . Testovací provoz se mu jeví naprosto vyhovující. Použije-li algoritmus s logaritmickou složitostí, který mu například někdo doporučil, pravděpodobně bude zapsán daleko komplikovaněji a navíc při testovacím provozu bude dávat horší výsledky. Programátor snadno dojde k závěru, že složitý logaritmický algoritmus je pěkný nesmysl, a zamítne jej. Při nasazení aplikace v praxi se však nepoužívá malá testovací množina dat, ale rozsáhlejší soubory například o rozsahu n_1 , které prakticky vždy překročí kritickou velikost n_k , při níž jsou vlastnosti obou algoritmů zhruba vyrovnány. Dojde tedy k tomu, že v praxi bude toto diletantské řešení zcela nepřijatelné, protože spotřeba času může úplně překročit přijatelné hodnoty. Při interaktivním chování se například očekává odezva programu do 5 vteřin – uživatel však zde bude moci pouze pořádně prostudovat ikonu s přesýpacími hodinami ...

6.6 Přehled typických složitostí

Algoritmy můžeme rozdělit do tříd složitostí podle $O(N)$. Přitom platí, že algoritmus z vyšší třídy je pomalejší (prostorově náročnější) než algoritmus z předchozí třídy.

Pro malá N nemusí být toto pravidlo zřetelné. Pokud ale patří dva algoritmy do různých tříd složitosti, pak vždy existuje takové množství dat, od kterého je asymptoticky lepší algoritmus (s nižší třídou složitosti) vždy efektivnější. Příklad dvou takových algoritmů byl již uveden.

Konstanty, kterými se násobí funkce nebo se posunují vůči počátku souřadnic, nejsou vůbec podstatné. Není tedy důležité, jak rychlý počítač použijeme pro testování, není ani důležité, na jakém operačním systému a s jakými pomocnými prvky nebo přidavnými modifikacemi testovací úloha běžela. Často ani není možné efektivně určit kritické množství n_k , důležité je pouze zařadit algoritmus do dané třídy.

6.6.1 Třídy složitostí

Následující seznam uvádí několik nejvýznamnějších složitostních tříd:

- $O(1)$ – konstantní
- $O(\log N)$ – logaritmická
- $O(N)$ – lineární
- $O(N \cdot \log N)$ – lineárně logaritmická
- $O(N^2)$ – kvadratická
- $O(N^3)$ – kubická
- obecně $O(N^k)$ – polynomiální
- obecně $O(k^N)$ – exponenciální
- $O(N!)$ – faktoriálová

Třídy jsou seřazeny vzestupně a platí:

$$1 \ll \log(N) \ll N \ll N \cdot \log(N) \ll N^k \ll k^N \ll N!$$

6.6.2 Typické příklady časové složitosti algoritmů

Uvedeme v tomto místě algoritmy, k nimž se budeme postupně dostávat i v dalších kapitolách.

Konstantní složitost – $O(1)$

- indexování prvku v poli

Logaritmická složitost – $O(\log N)$

- vyhledání prvku v seřazeném poli metodou půlení intervalu
- zařazení prvku do uspořádaného stromu

Lineární složitost – $O(N)$

- vyhledání prvku v neseřazeném poli sekvenčním vyhledáváním
- součet vstupní řady hodnot

- výpočet členů Fibonacciho posloupnosti iterativním algoritmem

Lineárně logaritmická složitost – $O(N \cdot \log N)$

- řazení binárním uspořádaným stromem (průměrná složitost)
- řazení hromadou, řazení slučováním, Quick sort

Kvadratická složitost – $O(N^2)$

- některé jednodušší řadicí metody (Bubble sort, Select sort, Insert sort...)
- zpracování matice čísel (součet matic, nulování, transpozice), závislost je na řádu matice

Kubická složitost – $O(N^3)$

- násobení matic, závislost na řádu matic

Exponenciální složitost – $O(2^N)$

- přesné řešení problému obchodního cestujícího hrubou silou
- rekursivní výpočet členů Fibonacciho posloupnosti

6.6.3 Snižování složitosti a efektivita algoritmů

Složitost zvoleného algoritmu má přímý vliv na to, jak bude daný algoritmus efektivní. Nejčastěji je potřebné si uvědomit alespoň jednoduché případy, kdy při konstantní složitosti na množství vstupních dat nezáleží a čas potřebný pro zpracování je stále stejný, při lineární složitosti je spotřebovaný čas nebo prostor přímo úměrný množství dat a při kvadratické složitosti se při dvojnásobném vstupu zvýší potřeba času nebo prostoru čtyřikrát.

Cílem tedy je pracovat s algoritmy, které mají v nejhorším případě polynomiální složitost. Vyšší třídy (exponenciální, faktoriálová) jsou pro vyšší N nepoužitelné a nepomohou ani rychlejší stroje.

Tabulku s odhadem trvání algoritmů různé složitosti uvádí např. R. Klein v článku *Asymptotická časová složitost algoritmů*⁴.

K nižší složitosti algoritmů přispívá jak optimální využití paměti, tak vhodná volba datových struktur – způsob uložení dat určuje i časovou a paměťovou složitost práce s těmito daty, což může ovlivnit i výslednou složitost celého algoritmu.

Obě kritéria (čas, prostor) obvykle pracují proti sobě, proto si programátor musí zvolit, čemu dá přednost. V dnešní době se obvykle upřednostňuje zkrácení času – paměťový prostor se může koupit, čas ovšem nikoliv.

⁴ <http://radeklein.cz/asymptoticka-casova-slozitost-algoritmu>

7 Operace s jednotlivými bity

Celočíselné datové typy umožňují kromě uvedených běžných aritmetických operací ještě pracovat poněkud jinak – s jednotlivými bity. Proč to potřebujeme? Aplikací je hned několik:

- *Uložení více logických hodnot do jednoho celku.* Do jednoho bajtu se vejde 8 bitů, každý může vyjadřovat nějakou logickou hodnotu. V současné době ani tak nejde o úsporu místa v paměti, ale spíše o to, že seznam takových logických hodnot se může chovat jako jedna proměnná, může se snadno přenášet, měnit a vyhodnocovat. Velmi často se tento systém používá jako stavová informace – například po provedení nějaké operace se do speciálního registru uloží, že výsledek přetekl z nejvyššího řádu, výsledek je nulový, při výpočtu se provedl přenos mezi 16. a 17. řádem atd. Každá tato informace je vyjádřena hodnotou jednoho bitu tohoto stavového registru.
- *Indikace určité množiny hodnot.* Například znaky v kódu ASCII lze jednoduše zařadit do určité kategorie, podíváme-li se, jak vypadají jejich binární hodnoty. Řídící znaky s kódy od nuly do 31 mají vždy nejvyšší tři bity nulové. Všechny číslice mají horní čtyři bity s hodnotou **0011**, velká písmena **010** a malá **011**. Přechod mezi velkými a malými písmeny spočívá pouze v nastavení nebo nulování jednoho bitu.
- *Náhrada aritmetických operací.* Posun bitové podoby čísla o jednu pozici vlevo odpovídá násobení dvěma, podobně posun vpravo dělení dvěma.
- *Možnosti zobrazení dat uložených v paměti.* Podíváme-li se na hodnotu v paměti po jednotlivých bitech, můžeme sledovat způsoby zobrazení hodnot jednotlivých typů.
- *Zabezpečení.* Při přenosu dat potřebujeme dodat zabezpečující informaci, která se obvykle vypočítává z bitové podoby dat. Nejjednodušší formou zabezpečení je parita, kde se bitová hodnota doplňuje na sudý (lichý) počet binárních jedniček.
- *Komprimace.* Uložení informace na minimálním prostoru vyžaduje ukládat data nikoliv jako proud bajtů, ale spíše jako proud bitů.
- *Šifrování.* Jednou z nejjednodušších možností, jak zašifrovat data, je provést nějakou logickou operaci s jednotlivými bity (například výhradní součet).

7.1 Operace s bity

Protože jeden bit představuje zobrazení pouhých dvou stavů, přímo se nabízejí operace odpovídající logickým operacím: logický součet po bitech, také logický součin, logická negace, výhradní součet.

Príslušné operátory a příklady jsou uvedeny v následujícím přehledu. S výhodou můžeme zapisovat hodnoty v šestnáctkové soustavě, odkud je přímo vidět, jak vypadá jejich bitová podoba:

Operace	Operátor	Příklad								
bitový součet	 binárně:	0x2c 0x44 = 0x6c								
		<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>	0	0	1	0	1	1	0	0
		0	0	1	0	1	1	0	0	
		<table><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	0			
= <table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>	0	1	1	0	1	1	0	0		
0	1	1	0	1	1	0	0			
bitový součin	& binárně:	0x2c & 0x44 = 0x04								
		<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>	0	0	1	0	1	1	0	0
		0	0	1	0	1	1	0	0	
		<table><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	0			
= <table><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	0	0	0	0	1	0	0		
0	0	0	0	0	1	0	0			
negace	~ binárně:	~0x2c = 0xd3								
		<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>	0	0	1	0	1	1	0	0
		0	0	1	0	1	1	0	0	
= <table><tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr></table>	1	1	0	1	0	0	1	1		
1	1	0	1	0	0	1	1			
výhradní součet	^ binárně:	0x2c ^ 0x44 = 0x68								
		<table><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>	0	0	1	0	1	1	0	0
		0	0	1	0	1	1	0	0	
		<table><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	1	0	0	0	1	0	0
0	1	0	0	0	1	0	0			
= <table><tr><td>0</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	1	0	1	0	0	0		
0	1	1	0	1	0	0	0			

7.2 Posuvy (rotace)

Pojmem **posuv** (někdy též *rotace*) označujeme přemístění jednotlivých bitů v rámci daného prostoru. Posuv může být prováděn doleva, tj. nejvyšší bit se ztrácí, na jeho místo se dostane dosavadní druhý nejvyšší bit atd., na uvolněné místo nejnižšího bitu se dostává binární nula. Analogicky pracuje posuv doprava – nejnižší bit se ztrácí, na uvolněnou pozici nejvyššího bitu se vloží binární nula.

Posuvy mohou být interpretovány v celočíselných datech jako celočíselné násobení (dělení) dvěma. U přirozených čísel platí to, co bylo uvedeno v předchozím odstavci. Uvažujeme-li však i záporná čísla, která jsou zobrazována v doplňkovém kódu, posuv vpravo musí fungovat jinak, aby reprezentoval dělení dvěma.

Posuv vpravo u záporných čísel funguje tak, že do nejvyššího bitu se vkládá stejná hodnota, která tam byla předtím. Záporné číslo tak zůstává stále záporným číslem, jeho hodnota je poloviční. Pokud je však hodnota interpretována jinak než záporné číslo, potřebujeme ponechat posuv v původní podobě (do nejvyššího bitu se vloží vždy nula). Rozlišujeme **aritmetický posuv** – respektuje záporná čísla v doplňkovém kódu – a **logický posuv** – posloupnost bitů se nechápe jako celočíselná hodnota. O jaký typ posuvu se bude jednat, to určuje datový typ: celé číslo bez znaménka (**unsigned**) má logický posuv, celé číslo se znaménkem pak aritmetický posuv.

Operace s příklady jsou soustředěny v následující tabulce:

Operace	Operátor	Příklad
posuv vlevo	<<	<code>0x2c << 2 = 0xb0; 44 · 4 = 176</code>
	binárně:	<div> <div><< 2</div> <div> <div>0</div><div>0</div><div>1</div><div>0</div><div>1</div><div>1</div><div>0</div><div>0</div> </div> </div> <div>=</div> <div> <div>1</div><div>0</div><div>1</div><div>1</div><div>0</div><div>0</div><div>0</div><div>0</div> </div>
logický posuv vpravo	>>	<code>0xac >> 2 = 0xb; 172 ÷ 4 = 43</code>
	binárně:	<div> <div>>> 2</div> <div> <div>1</div><div>0</div><div>1</div><div>0</div><div>1</div><div>1</div><div>0</div><div>0</div> </div> </div> <div>=</div> <div> <div>0</div><div>0</div><div>1</div><div>0</div><div>1</div><div>0</div><div>1</div><div>1</div> </div>
aritmetický posuv vpravo	>>	<code>0xac >> 2 = 0xb; -84 ÷ 4 = -21</code>
	binárně:	<div> <div>>> 2</div> <div> <div>1</div><div>0</div><div>1</div><div>0</div><div>1</div><div>1</div><div>0</div><div>0</div> </div> </div> <div>=</div> <div> <div>1</div><div>1</div><div>1</div><div>0</div><div>1</div><div>0</div><div>1</div><div>1</div> </div>

Operátory pro bitový posuv jsou shodné s proudovými operátory pro vstup a výstup. Jaká operace se provede, to je určeno okolními operandy. V případě možného dvojího pochopení je nutné použít kulaté závorky pro upravení priorit vyčíslování výrazu.

7.3 Masky

Pro práci s jednotlivými bity se často používají tzv. **masky**. Masky podobně jako na maškarním plese něco odhaluje a něco zakrývá. Zde máme na mysli masku v podobě posloupnosti jedniček a nul, kde například jedničky odkrývají a nuly zakrývají nějakou hodnotu na příslušných bitových pozicích.

Masku můžeme použít několika způsoby – ke zjištění hodnoty určitého bitu, k nastavení binární jedničky na určitou pozici nebo k nastavení binární nuly na určitou pozici. Z toho je vidět, že maskou můžeme libovolně operovat s libovolným bitem v daném prostoru.

Například potřebujeme zjistit hodnotu třetího nejnižšího bitu zadaného čísla. Mějme následující úsek programu:

```

272 int cislo; bool bit;
273 cin >> cislo;
274 bit = (cislo & 4) > 0;
275 cout << "Třetí nejnižší bit má hodnotu " << bit << endl;

```

Podstatná operace je na řádce 274. Když si situaci ukážeme binárně (uvažujme pouze nejnižší bajt proměnné `cislo`, hodnotou `x` naznačujeme, že na hodnotě daného bitu nezáleží), dostáváme:

	x	x	x	x	x	1	x	x
& maska	0	0	0	0	0	1	0	0
=	0	0	0	0	0	1	0	0

nebo

	x	x	x	x	x	0	x	x
& maska	0	0	0	0	0	1	0	0
=	0	0	0	0	0	0	0	0

Pokud je výsledek nenulový, byl původní bit v hodnotě 1, podle toho určujeme výsledek v logické proměnné **bit**.

Chceme-li určitý bit nastavit na hodnotu 0, analogicky použijeme masku, v níž budou všude binární jedničky, jen v dané pozici bude nula. Logickým součinem hodnoty a této masky bude daný bit vynulován. Příklad: Nastavení hodnoty nula na pozici 5. nejnižšího bitu:

	x	x	x	x	x	x	x	x
& maska	1	1	1	0	1	1	1	1
=	x	x	x	0	x	x	x	x

Podobně postupujeme při nastavení bitu na určité pozici na hodnotu jedna. Masku má v tomto případě jedničku na dané pozici, jinde nuly a s hodnotou se provede logický součet. Například pro nastavení jedničky ve 4. nejnižším bitu:

	x	x	x	x	x	x	x	x
maska	0	0	0	0	1	0	0	0
=	x	x	x	x	1	x	x	x

Podobně můžeme maskováním zjišťovat hodnoty půlbajtu (vhodné např. při zhuštěném kódování BCD), binárního prefixu 10 pro pokračující bajty v kódování UTF-8 apod.

7.4 Zjišťování hodnot významných bitů

Kromě maskování můžeme ve dvou případech zjistit hodnotu daného bitu i jinak.

Hodnotu nejnižšího bitu lze zjistit testem na lichost čísla. V binární soustavě mají lichá čísla nejnižší bit s hodnotou 1, sudá čísla s hodnotou nula. Lichost/sudost čísla zjistíme zbytkem po dělení dvěma.

Hodnotu nejvyššího bitu u čísel v proměnných s celočíselným datovým typem se znaménkem (interpretuje se v doplňkovém kódu) zjistíme testem na velikost čísla menší než nula.

8 Uživatelské datové typy

V jazyce C++ máme přímo k dispozici několik datových typů dostupných pomocí rezervovaných identifikátorů, jako je například **float**. Připomeňme, že každý datový typ je určen množinou povolených hodnot a skupinou povolených operací s těmito hodnotami. Předdefinované datové typy mají pevně určeno, jaké hodnoty zahrnují (je to většinou dáno velikostí paměťového prostoru) a jak se tyto hodnoty interpretují (jeden a tentýž obsah paměťového prostoru může být chápán rozdílně – vzpomeňme například na binární hodnotu **10100110** – může jít o číslo 166 nebo o číslo –90).

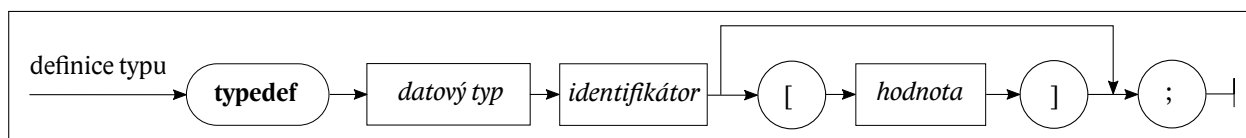
Přestože počítač je schopen fyzicky zpracovávat pouze binární čísla a všechny myslitelné hodnoty všech datových typů jsou vždycky zobrazeny jako odpovídající binární posloupnosti, člověk potřebuje pracovat s hodnotami, které více odpovídají modelované realitě než strojovému vnímání. Potřebuje pracovat i s hodnotami, které žádnému předdefinovanému typu neodpovídají.

Ve většině programovacích jazyků (zejména univerzálních) existuje z tohoto důvodu možnost vytvořit další datové typy podle přání uživatele. Tato možnost nebývá neomezená, obvykle lze vytvářet nové datové typy omezeným způsobem a často nelze vytvořit úplně funkční obdobu těch předdefinovaných, ale lze tím pokrýt běžné potřeby ukládání dat. Kromě toho lze využívat i dynamických struktur, jejichž konstrukce poskytují téměř neomezený prostor pro modelování nejrůznějších složitých datových seskupení – tomuto problému se věnuje příslušná kapitola dále.

Pro definici uživatelského typu lze využít několika prvků – jednak je možné vytvořit nový identifikátor typu použitelný pro deklarace příslušných proměnných, jednak existují možnosti konstrukce hodnot nového datového typu postavených na typech předdefinovaných.

8.1 Definice identifikátoru datového typu

V jazyce C++ existuje k tomuto účelu příznačné klíčové slovo **typedef**. Jeho použití ilustruje následující zjednodušený syntaktický diagram:



V diagramu představuje blok „datový typ“ identifikátor některého existujícího datového typu (nebo odvození jiného typu) a tomuto typu je přiřazeno nové jméno. Za novým jménem ještě mohou být hranaté závorky představující pole (viz pozdější kapitola). Za celou definicí je pak středník jako za obvyklou deklarací.

Příklady:

```

276 | typedef long long int TypDat;
277 | typedef TypDat TypPole[400];

```

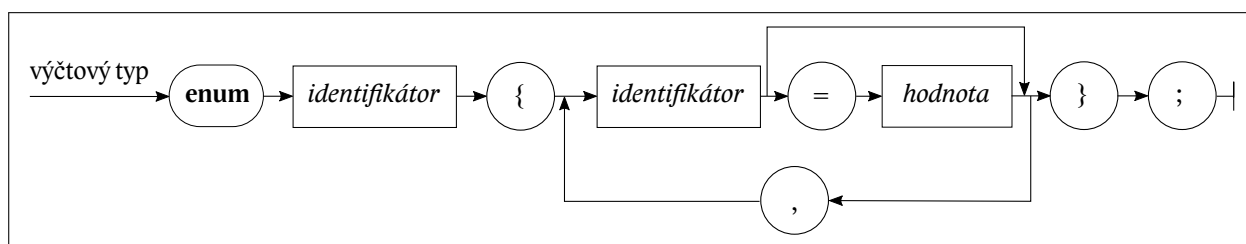
První řádek představuje často využívanou možnost „přejmenování“ existujícího typu. Důvody pro takový krok jsou vcelku pádné: nový identifikátor `TypDat` se pak používá všude tam, kde se deklarují proměnné nějakým způsobem svázané se zpracováváním daty včetně formálních parametrů podprogramů apod. Pokud se později rozhodneme, že data budou odlišná (chceme například přejít na bezznaménkový typ), stačí změnit tuto definici a v celém programu se budou data chápat správným (novým) způsobem.

Druhý řádek využijeme zejména v kapitole, která se věnuje polím. Jde o definici datového typu pole o 400 prvcích složených ze složek již vytvořeného nového typu `TypDat`. Toto je příklad situace, kdy se zároveň s novým identifikátorem konstruuje nová struktura.

8.2 Uživatelský výčtový typ

Datový typ, u něhož může uživatel definovat vlastní hodnoty (a provést tak jejich výčet), nazýváme **výčtový typ** a je definován pomocí klíčového slova `enum` (zkratka slova „enumerate“). Jde o datový typ patřící do kategorie jednoduchých datových typů, jehož hodnotami jsou *identifikátory* představující celočíselné hodnoty, pod kterými jsou uloženy v paměti.

Následující syntaktický diagram ukazuje způsob deklarace proměnné takového typu.



V tomto diagramu se první identifikátor považuje za proměnnou, další identifikátory uváděné v seznamu ve složených závorkách pak za hodnoty, kterých může tato proměnná nabývat. Každá hodnota v závorce je automaticky očíslována. Pokud neuvedeme nic dalšího, bude mít první identifikátor hodnotu nula, další jedna atd. Pokud uvedeme rovnítko a číselnou hodnotu, bude daný identifikátor představovat tuto hodnotu, následující bude o jedničku větší atd. Můžeme ale specifikovat hodnotu u každého identifikátoru, hodnoty tedy nemusí tvořit souvislou řadu.

Příklady:

```
278 enum {bila, cerna, cervena, zelena, modra} barva;
279 enum {po=1, ut, st, ct, pa, so, ne} Den;
280 enum {readonly=1, archive=2, directory=4, shared=8, link=16} FlagBit;
```

S hodnotami výčtového typu lze provádět všechna běžná porovnání, bohužel je však nelze přímo číst ze standardního vstupu a při výpisu se objeví pouze číslo, pod kterým jsou uloženy v paměti. Hlavním důvodem používání takových výčtových hodnot je však zpřehlednění zdrojového textu. Napíšeme-li například `if(Den>pa)`, máme na první pohled jasno, že při platnosti této podmínky je v proměnné `Den` sobota nebo neděle. Kdybychom místo toho psali `if(X>5)`, není bez znalosti širšího kontextu vůbec patrné, co hodnota „5“ znamená.

8.3 Definice identifikátoru typu výčet

U definice proměnné typu výčet zřetelně vidíme, že se jedná o poměrně složitý zápis, jehož opakování v programu na různých místech je zcela jistě naprosto nežádoucí. Proto je velmi vhodné definovat nový identifikátor takového typu a ten pak použít všude (různá místa při deklaracích proměnných, při definici formálních parametrů podprogramů atd.)

Klíčové slovo **enum** má schopnost také definovat nový typ – jeho jméno zapisujeme *před* složenou závorku se seznamem hodnot. Navážeme-li na uvedené příklady, můžeme zapisovat:

```
281 enum TypBarva {bila, cerna, cervena, zelena, modra};
282 enum TypTyden {po=1, ut, st, ct, pa, so, ne};
283 enum TypPriznak {readonly=1, archive=2, directory=4, shared=8, link=16};
284 ... //následně můžeme deklarovat proměnné:
285 TypBarva barva;
286 TypTyden Den;
287 TypPriznak FlagBit;
```

K definici nového identifikátoru datového typu samozřejmě lze využít i konstrukce s klíčovým slovem **typedef**. Jak už bylo uvedeno, **typedef** považuje poslední identifikátor zápisu za nový datový typ, vše, co je před tím, je pak definice tohoto typu. Ukážeme alternativní zápis předchozích definic:

```
288 typedef enum {bila, cerna, cervena, zelena, modra} TypBarva;
289 typedef enum {po=1, ut, st, ct, pa, so, ne} TypTyden;
290 typedef enum {readonly=1, archive=2, directory=4, shared=8, link=16} TypPriznak;
```

8.4 Datový typ void

Výjimečným předdefinovaným datovým typem v jazyce C++ je typ **void**. Představuje tzv. prázdný typ – nemá žádné hodnoty a žádné operace. Nelze vytvořit proměnnou tohoto typu, protože ani není určena žádná paměťová velikost, kterou by tento typ zabíral.

K čemu pak něco takového vůbec je a proč se o tom zmiňujeme v části zabývající se uživatelskými datovými typy?

Napřed odpověď na první část uvedené otázky: stručně řečeno, datový typ **void** využijeme tam, kde je nějaký datový typ vyžadován, ale v tom místě nic takového nepotřebujeme. V kapitole o podprogramech uvidíme nejčastější aplikaci tohoto principu.

Jak se může datový typ **void** podílet na uživatelských typech? Odpověď je obsažena v kapitole zabývající se pojmem ukazatel – ukážeme si, že pomocí **void*** lze vytvořit obecnou strukturu, do níž lze vložit jakákoliv data, takže ji lze uživatelsky libovolně využít.

8.5 Definice konstant

Obecně platí, že téměř jakákoliv konstanta uvedená jako literál přímo v textu programu je potenciálním problémem – ať už se jedná o zmíněnou nejasnost při čtení kódu, nebo jde o tikající časovanou bombu, která vybuchne v okamžiku, kdy začneme provádět v programu nějaké změny.

Napišeme-li například:

```
291 unsigned int barva, pocet;
292 ... //proměnná barva je naplněna nějakou hodnotou
293 if (barva == 2 or barva == 4) VymenBarvu(barva);
294 ...
295 while (pocet < 35) ZpracujData(pocet);
```

nemáme na první pohled ani tušení, o jaké barvy jde, když je volána procedura `VymenBarvu`. Tento problém jsme již zmínili v souvislosti s možnostmi výčtového typu. V podmínce cyklu na řádce 295 však jde ještě o další případ. Je z kontextu celkem jasné, že hodnota 35 představuje nějaký počet, takže nejsme na pochybách, co se děje – dokud je počet nižší než 35, volá se procedura `ZpracujData` a ta určitě ten počet zvyšuje. Ale proč je ta mez zrovna 35? Nemůže přijít zadavatel a říct, že potřebuje třeba 50? Co když na té hodnotě 35 závisí ještě dalších pět procesů a budeme chtít zákazníkovi vyhovět a zvýšit hodnotu na 50? Budeme prolézat celý program a pracně zjišťovat, ve kterých místech je potřeba přepsat 35 na 50? Co když tam bude hodnota 35 ještě v jiném významu (třeba počet vypisovaných řádků tabulky na stránku) než počet zpracovávaných dat a my ji chybně přepíšeme taky na 50? Zaneseme opravou jedné chyby někam jinam chybu další.

Tomu se určitě chceme vyhnout! Hodnotě 35 dáme symbolické jméno (identifikátor), které bude jednoznačně říkat, o co jde, ale hlavně změnou jeho definice v jediném místě se všude v programu automaticky začne pracovat s novou hodnotou. A k tomu slouží definice konstant.

Konstanty se definují tak, že k obyčejné deklaraci proměnné se vlevo přidá klíčové slovo `const`. Deklarace musí obsahovat vložení počáteční hodnoty, která se pak dále v programu nemůže změnit. Příklady:

```
296 const unsigned int MaxPocet = 35;
297 const string Konec = "konec dat.";
298 const char Zarazka = '.';
299 const float stupen = M_PI/180;
```

V posledním příkladu je vidět, že jako počáteční hodnota může být uveden výraz, jehož hodnotu ovšem musí překladač spočítat ještě v době překladač, musí se v něm proto objevovat pouze jiné konstanty.

9 Podprogramy

Pojem **podprogram** lze definovat jako logicky ucelenou a pojmenovanou část programu, která může komunikovat s okolím prostřednictvím parametrů.

Podprogram můžeme chápat jako relativně samostatný celek. Je to velmi důležitý nástroj, který umožňuje řešený problém rozdělit na menší podproblémy, jejichž implementace je již známa, nebo je natolik jednoduchá, že ji snadněji nalezneme – je to tedy nástroj realizace metody shora dolů. Implementované podproblémy ve formě podprogramů mohou být pak využívány ve více programech, protože jsou soustředěny do dostupných knihoven, mohou také být součástí objektů (tzv. metodami) a sloužit pro obsluhu jejich datových složek apod.

9.1 Druhy a definice podprogramů

Podle účelu a logiky činnosti můžeme podprogramy rozdělit na dva základní druhy:

- **procedura** – podprogram, jehož cílem je provedení nějaké činnosti, přičemž výsledkem je více hodnot, nějaké změny v paměti, výstup výsledků apod.;
- **funkce** – podprogram, jehož hlavním cílem je získání jedné výsledné hodnoty prostřednictvím provedení nějaké činnosti.

Některé programovací jazyky důsledně syntakticky oddělují procedury a funkce; v jazyce C (a C++) se technicky mohou implementovat pouze funkce, proto se můžeme setkat s názorem, že pojem podprogram je synonymem pojmu funkce. Uvidíme však, že procedura je zřetelně definována jako zvláštní případ funkce, a protože nám jde především o logiku věci, budeme vždy mluvit o obou druzích podprogramů – o procedurách i funkcích.

Podobně jako například u proměnných vždy musíme napřed sdělit, jak se proměnná jmenuje, jakého je typu a pak ji teprve můžeme použít, je u podprogramů napřed potřebné podprogram definovat (sdělit, co má podprogram dělat) a pak jej teprve můžeme využít, tj. aktivovat = volat.

Definice a volání podprogramu spolu musí pochopitelně souhlasit, a to ve všech aspektech – je to poněkud složitější než u obyčejné proměnné. Dále se budeme zabývat oběma částmi: jak se provádí definice, jak se zapisuje volání podprogramu.

Definice podprogramu se zapisuje podle následujícího syntaktického diagramu:

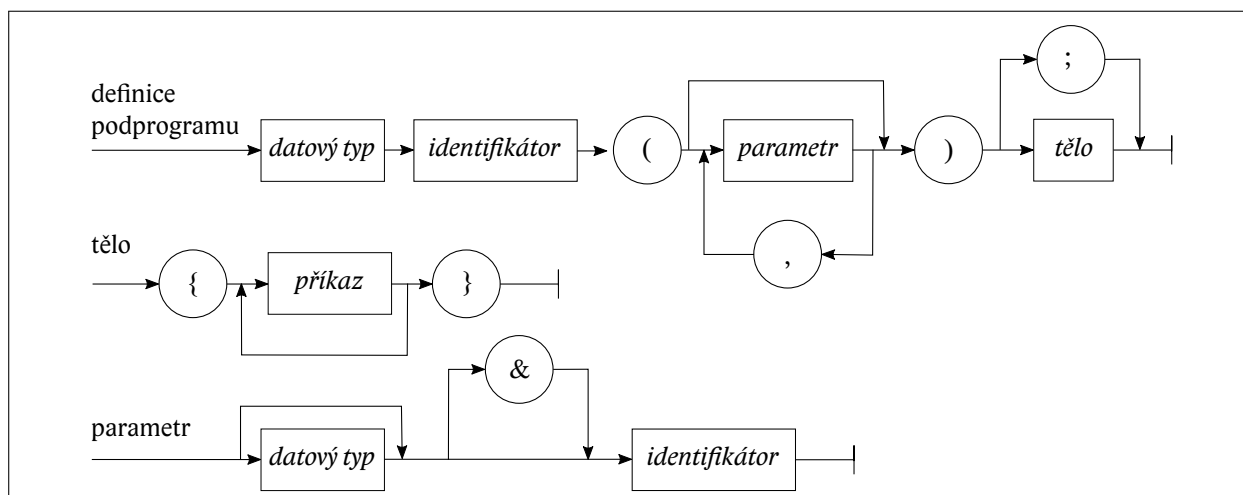


Diagram je tentokrát složen ze tří částí, jejichž významy popíšeme v následujícím textu. Definice začíná datovým typem – protože technicky jsou všechny podprogramy chápány jako funkce, jde o datový typ výsledné hodnoty, kterou funkce vypočítává. Pokud je tímto typem **void**, „funkce“ nevrací žádnou hodnotu – jedná se tedy o proceduru.

Následuje identifikátor, tj. jméno podprogramu. Pod tímto jménem bude možné podprogram později použít. Za jménem následují kulaté závorky, v nichž může být uvedena řada parametrů oddělených čárkami (nemusí tam být žádný, ale závorky tam musí být vždycky). Pomocí parametrů podprogram komunikuje s okolím, k tomu se ještě vrátíme. Část definice od začátku až po parametry se někdy nazývá **hlavička podprogramu**.

Za hlavičkou může následovat tělo podprogramu – v něm (jak vidíme ve druhé části diagramu) jsou příkazy, které implementují řešený algoritmus podprogramu. Můžeme však místo těla zapsat pouze středník – pak se jedná o tzv. **odloženou definici** – překladač zná všechny informace potřebné ke kontrole, zda je případné volání v pořádku, nezná jen tělo. To ovšem musí být zapsáno někdy později. Odloženou definicí můžeme oddělit hlavičku podprogramu od těla, což se někdy velmi hodí. Dodatečná definice těla začíná stejnou hlavičkou (překladač kontroluje, zda jsme ji opsali správně) a pak následuje to odložené tělo.

Tělo podprogramu je složeno z posloupnosti příkazů uzavřených do složených závorek. Už víme, že složené závorky ohraničují posloupnost příkazů, můžeme si představit, že se jedná o něco podobného. Podprogram však musí mít tělo ohraničeno složenými závorkami i v případě, že obsahuje pouze jediný příkaz. V těchto příkazech je možné používat parametry a provádět s nimi naznačený algoritmus.

Parametry si můžeme představit jako proměnné, které jsou v hlavičce deklarovány (proto tam potřebujeme psát datový typ) a které budou v okamžiku použití podprogramu naplněny nějakými hodnotami zvnějšku. Pokud neuvedeme datový typ (syntax to dovoluje), dosadí se tam automaticky typ **int**.

Parametry uvedené v definici podprogramu se nazývají **formální parametry**. Představíme si je jako prázdná nachystaná místa, s nimiž pracují příkazy v těle podprogramu a s nimiž se odehrává celý algoritmus. K přesné činnosti parametrů se ještě dostaneme při vysvětlení volání podprogramu.

V těle podprogramu můžeme deklarovat proměnné, definovat nové datové typy a se vším pak pracovat – ovšem všechno, co uvnitř těla vytvoříme, automaticky *zanikne* s ukončením podprogramu, a to včetně formálních parametrů.

Na následujících příkladech ukážeme některé možnosti definice podprogramů:

```
300 int Podil(int a, int b){ //celočíslná funkce
301     if (b!=0)
302         return a/b;      //výsledkem funkce je podíl hodnot parametrů
303     else return 0;      //nebo nula, když jmenovatel je nulový
304 }
305 void Vypis(float jedna, dve); //odložená definice procedury Vypis
306 bool Srovnani(float cislo1, float cislo2){
307     //funkce, do níž vstupují dvě racionální čísla
308     //a vydává logickou výstupní hodnotu
309     if (cislo1 < 0) cislo1 = -cislo1;
310     Vypis(cislo1, int(cislo2));
311     return cislo1 < cislo2;
312 }
313 void ZobrazCaru(){ //procedura bez parametrů
314     cout << "-----" << endl;
315     //procedura nepotřebuje příkaz return
316 }
317 void Vypis(float jedna, dve) { //doplnění těla odložené definice
318     //Pozor! Druhý parametr není typu float, ale int!
319     cout << setprecision(dve) << jedna << endl;
320 }
```

9.2 Volání podprogramu

Volání podprogramu je okamžik, kdy se ze současného místa přesouvá řízení do podprogramu, po jeho provedení se zase řízení vrací zpět přesně za místo, ze kterého se do podprogramu „odskočilo“.

Zápis volání podprogramu je relativně jednoduchý: zapíše se identifikátor podprogramu a v kulatých závorkách pak odpovídající počet parametrů. Parametry zapisované v okamžiku volání se nazývají **skutečné parametry**.

Volání procedury, kdy se nevrací žádná hodnota, se uvádí v místě *příkazu*. Naopak volání funkce je chápáno jako operand ve výrazu – uvádí se všude tam, kde výslednou hodnotu můžeme nějak zpracovat (v odpovídajícím místě výrazu).

V okamžiku volání se provádí **substituce parametrů** – skutečné parametry se dosadí do odpovídajících míst formálních parametrů, s nimiž se pak provede tělo podprogramu. Tato substituce může být provedena dvěma způsoby:

1. Hodnota skutečného parametru se zkopíruje do místa formálního parametru. Skutečným parametrem může být jakýkoliv výraz odpovídajícího typu: tento výraz se napřed vypočítá a pak se jeho výsledná hodnota dosadí do formálního parametru. Zkráceně tento způsob substituce nazýváme **volání hodnotou**.
2. Formální parametr nepřebírá hodnotu, ale adresu skutečného parametru. Je jasné, že v tomto případě nemůže být skutečným parametrem výraz, protože ten žádnou adresu nemá, ale skutečným parametrem musí být proměnná, jejíž adresa se ztotožní s adresou formálního parametru. To znamená, že skutečný a formální parametr v tomto okamžiku mají v paměti stejné umístění (stejnou adresu). Přířímým důsledkem tohoto stavu je, že jakákoliv operace provedená v těle podprogramu s formálním parametrem se projeví jako změna hodnoty i ve skutečném parametru. Zkráceně tento způsob substituce nazýváme **volání odkazem**. Formální parametr, který chceme volat odkazem, je zapsán v definici hlavičky podprogramu se znakem `&` (operátor odkazu, tj. získání adresy; princip odkazu je blíže vysvětlen v kapitole 12).

Parametry volané hodnotou se využívají jako *vstupy* do podprogramu. Uvědomíme-li si, že formální parametry vzniknou v okamžiku volání, zkopírují se do nich hodnoty skutečných parametrů, provede se s nimi potřebná operace a na konci podprogramu zaniknou, nemáme možnost ty vypočítané hodnoty dostat zpět do místa, odkud byl podprogram vyvolán.

Parametry volané odkazem však fungují poněkud jinak. Opět platí, že formální parametry zaniknou s koncem podprogramu, ale je-li parametr předán odkazem, v době činnosti podprogramu se adresa formálního a skutečného parametru rovnají, takže po skončení podprogramu zůstane v místě skutečného parametru *změněná* hodnota. To můžeme také chápat jako *výstup* z podprogramu, proto se parametry volané odkazem považují za *výstupní*.

9.3 Skok tam, skok sem ...

Podívejme se nyní ještě na důležitý mechanismus, který umožňuje volat podprogram z různých míst programu. Zabývali jsme se v různých souvislostech programovými skoky, řešili jsme podmíněné a nepodmíněné skoky používané pro větvení a cyklus. Jak je ovšem řešen skok do podprogramu?

Představme si situaci, že v hlavním programu potřebujeme ve dvou různých místech volat stejný podprogram, například zmíněnou funkci `Podíl`. Jak je to zařízeno?

Uvažujme, že máme k dispozici instrukci pro nepodmíněný skok nazvanou `JMP` a že ji použijeme pro skok do podprogramu. Situaci ilustruje obr. 11.

```

...
    JMP Podil  (skok na adresu Podil)
Zpet1  ...    (sem se vrátíme po provedení podprogramu)
...
    JMP Podil  (opět skok na adresu Podil)
Zpet2  ...    (sem se vrátíme po provedení podprogramu)

Podil  ...    (tady začíná podprogram)
...
    JMP Zpet1  (??? problém: potřebujeme pokaždé jinam)

```

Obrázek 11: Skok do podprogramu pomocí JMP

Vidíme, že tento systém je nepoužitelný – v okamžiku, kdy se chceme z podprogramu vrátit zpět, bychom i při druhém volání museli skočit na adresu `Zpet1`, nikoliv na potřebnou adresu `Zpet2`. Abychom ovšem mohli volat podprogram z neomezeného počtu míst, je potřebné tyto návratové skoky vyřešit jiným způsobem než skokem typu `JMP`.

K úspěšnému návratu z podprogramu do hlavního programu na potřebné místo musíme změnit i okamžik, kdy se přesouváme z hlavního programu do podprogramu. Zde nemůžeme použít instrukci nepodmíněného skoku `JMP`, ale instrukci `CALL`. Co tato instrukce udělá? Shrňme její činnost do dvou kroků:

1. *Záznam návratové adresy* – do zvláštní části operační paměti zvané *systémový zásobník* se uloží adresa následující za instrukcí `CALL`, protože na tuto adresu bude potřebné skočit při návratu z podprogramu.
2. Skok do podprogramu.

A co se musí stát v okamžiku, kdy chceme podprogram ukončit? Ani tam nemůže zůstat instrukce `JMP`, musíme ji nahradit instrukcí `RET`. Tato instrukce napřed vybere ze systémového zásobníku návratovou adresu a pak ji použije ke skoku. Znamená to tedy, že pokaždé skočí někam jinam, a to podle toho, jakou adresu získá ze systémového zásobníku.

Celou situaci ilustruje obr. 12.

```

...
    CALL Podil  (uložení Zpet1 a~skok na adresu Podil)
Zpet1  ...    (sem se vrátíme po provedení podprogramu)
...
    CALL Podil  (uložení Zpet2 a~opět skok na adresu Podil)
Zpet2  ...    (sem se vrátíme po provedení podprogramu)

Podil  ...    (tady začíná podprogram)
...
    RET        (skok na návratovou adresu)

```

Obrázek 12: Skok do podprogramu pomocí CALL

Volání podprogramu využívá paměťovou oblast (systémový zásobník) nejen na úschovu návratových adres, ale také na dočasné uložení formálních parametrů a všech proměnných deklarovaných uvnitř podprogramu. To, co je uloženo na systémovém zásobníku, je v době provádění podprogramů dostupné. Jakmile podprogram končí, vše se ze zásobníku odebere, nakonec se odebere návratová adresa a skočí se zpět do volajícího místa. Zásobník je „uklizen“.

V jazyce C++ se instrukce `RET` zapisuje příkazem `return`. Tento příkaz může mít jako svůj atribut výraz, jehož vyčíslením vznikne *návratová hodnota*, tj. hodnota, kterou nabývá funkce. V případě procedury není nutné příkaz vůbec v těle zapisovat, instrukce `RET` se automaticky vygeneruje na konci těla.

Je potřeba si uvědomit, že příkaz `return` je určitým typem nepodmíněného skoku. Místo, kde jej uvedeme, představuje konec provádění těla podprogramu, jakýkoliv další příkaz za příkazem `return` se nikdy neprovede. V souvislosti s tím, co jsme již uvedli o nepodmíněných skocích, je patrné, že s příkazem `return` musíme zacházet opatrně. Dobrou praxí je jeho uvedení pouze v *jediném místě těla podprogramu*, a to na úplném konci. Máme pak jistotu, že všechny příkazy, které vidíme v těle podprogramu zapsány, se skutečně provedou. Je to podobné jako s naprosto nežádoucími vícenásobnými skoky zevnitř těla cyklu příkazem `break` nebo `continue`. Navíc v případě návratové hodnoty máme jednoznačný přehled o tom, kde se návratová hodnota vezme a co bude vráceno.

Podíváme-li se například na uvedenou funkci `Podil`, je zde toto pravidlo porušeno. Abychom získali požadovaný tvar, bylo by vhodné zápis těla funkce upravit například takto:

```
321 int Podil(int a, int b){  
322     int vysledek;  
323     if (b!=0) vysledek = a/b;  
324     else vysledek = 0;  
325     return vysledek;  
326 }
```

I když je zápis těla funkce o něco delší a je zde navíc deklarována pomocná proměnná `vysledek`, přesto je i s ohledem na další možné budoucí úpravy těla funkce zápis zřetelnější a bezpečnější.

9.4 Globální/lokální

Jak už bylo zmíněno, formální parametry a proměnné deklarované uvnitř podprogramu platí a jsou použitelné jen v jeho těle. Ukázali jsme si, že je to technicky realizováno umístěním těchto prvků do systémového zásobníku v okamžiku volání podprogramu a jejich odebráním v okamžiku ukončení podprogramu.

Hovoříme o tom, že formální parametry a uvnitř deklarované proměnné jsou vzhledem k danému podprogramu **lokální**. Pokud existují nějaké proměnné nebo datové typy definované a deklarované vně podprogramu, nazýváme je **globální**. S globálními prvky lze manipulovat i uvnitř podprogramu, jsou tam viditelné, zatímco naopak manipulovat vně podprogramu s lokálními prvky nelze (nejsou v té chvíli v systémovém zásobníku, takže neexistují).

Co se stane v okamžiku, kdy nějaká lokální proměnná bude mít stejné jméno jako proměnná globální?

Systémový zásobník funguje jako skutečný zásobník – prvky, které se přidaly jako poslední, jsou nalezeny a použity jako první. Z toho tedy plyne, že lokální prvek vložený do zásobníku bude mít vždy přednost před globálním, má-li stejné jméno, *zakryje* a zneprístupní prvek globální.

Lokalitu proměnné jsme už v jiné souvislosti využívali: u cyklu `for` lze provést v inicializační části deklaraci proměnné – tato proměnná rovněž padne do systémového zásobníku a bude platit tak dlouho, dokud neskončí tělo cyklu. Po skončení cyklu se tato lokální proměnná odstraní ze zásobníku a zanikne.

Snahou a dobrou praxí je, že *veškeré prvky*, s nimiž podprogram pracuje, jsou buď předány jako parametry, nebo jsou deklarovány uvnitř podprogramu. Zcela nevhodné je, když podprogram manipuluje s globálními prvky. Až na čestné výjimky se jedná o tzv. **vedlejší efekt** podprogramu, který zabráňuje znovupoužití v jiném kontextu a vede k tzv. špagetovému kódu – nežádoucímu tvaru, kdy jakákoliv změna může zasáhnout nečekaně i jinou část programu, znesnadňuje jakékoliv modifikace a opravy a je zdrojem nepříjemných chyb.

Těmi čestnými výjimkami prvků, s nimiž podprogram manipuluje a nejsou lokální, mohou být standardní soubory (`cin`, `cout`, `cerr`, `clog`) – jsou globální, ale jsou dostupné ve všech programech a nehrozí, že by přesunem podprogramu bylo jejich použití narušeno. Takovou typickou procedurou, která nemá žádné parametry a nic nevrací, je `ZobrazCaru` v uvedených příkladech definic podprogramů. V jiném případě je ovšem podprogram bez parametrů velmi podezřelý!⁵

9.5 Funkce `main` a komunikace s OS

V jazyce C++ je celý program reprezentován podprogramem `main`. Tím je ještě více akcentován princip podprogramů. Funkce `main` má však poněkud jiný účel – není volána z nějakého jiného místa programu, ale můžeme si představit, že je volána z prostředí operačního systému, když spustíme přeložený program. Svými parametry a svou návratovou hodnotou pak komunikuje s operačním systémem.

Další výjimkou je uvedení parametrů funkce `main`. V příkladech uvedených v tomto textu jsme většinou psali:

```
327 | int main(){  
328 |     ...  
329 |     return 0;  
330 | }
```

Funkce má celočíselnou návratovou hodnotu a nemá žádné parametry. Co představuje návratová hodnota, kterou obvykle nastavujeme v příkazu `return` na nulu? Jde o tzv. návratový kód proce-

⁵ V případě, že uvažujeme podprogram jako metodu objektu, je potřebné do lokálních prvků zahrnout i složky daného objektu, k nimž má metoda přímý přístup. Proto se často metody objektů objevují bez parametrů, ale přitom to není příznak vedlejšího efektu, neboť manipulují s lokálními datovými složkami, které lze chápat podobně jako lokálně deklarované proměnné.

su – operační systém je schopen vyhodnotit, s jakým kódem skončil spuštěný proces. Hodnota nula představuje bezchybné ukončení, hodnoty jiné mohou indikovat různé chyby. Můžeme tím operačnímu systému sdělit, že například nebyl nalezen soubor s daty, při výpisu se zaplnil disk nebo nebylo připojeno zařízení, z něhož jsme chtěli číst. Operační systém na to může zareagovat spuštěním (nebo nespouštěním) následné aplikace, záznamem do chybového žurnálu atd.

Funkce `main` může mít také parametry. Operační systém je schopen předat hodnoty uvedené na příkazovém řádku, které tam byly zapsány při volání programu. Chceme-li tyto informace uvnitř programu zpracovat, uvedeme například:

```
331 | int main(int pocet, char *param[]){
332 |     ...
333 | }
```

První parametr `pocet` udává počet parametrů zadaných z příkazového řádku. Protože i jméno samotného spouštěného programu se bere jako nultý parametr, je hodnota `pocet` vždy nejméně 1. Druhý parametr je pak odkaz na pole řetězců, v němž jsou uloženy hodnoty jednotlivých parametrů (polím se věnuje jiná kapitola tohoto textu). Jen pro ilustraci uvedeme jednoduchý program, který vypíše všechny parametry zadané na příkazovém řádku:

```
334 | int main(int pocet, char *param[]){
335 |     for (int i=0; i<pocet; i++)
336 |         cout << "parametr[" << i << "]: " << param[i] << endl;
337 |     return 0;
338 | }
```

9.6 Další možnosti komunikace s operačním systémem

Funkce `main` reprezentující celý program komunikuje s operačním systémem ještě prostřednictvím standardních souborů (bylo vysvětleno v sekci 3.9), dále můžeme využít práce s uživatelskými soubory (je obsaženo v příslušné kapitole) a ještě lze číst hodnoty z proměnných procesového prostředí.

Pojem **prostředí** v tomto významu představuje paměťový prostor přidělený každému běžícímu procesu, v němž lze pomocí příkazů operačního systému vytvářet proměnné a vkládat do nich (výlučně textové) hodnoty. Tyto hodnoty pak lze v programu přečíst a chápat je jako další možný vstup.

Pro čtení obsahu proměnné prostředí slouží funkce `getenv(název)`. Název proměnné prostředí je parametrem funkce. Výsledkem je řetězcová hodnota. Pokud proměnná neexistuje, dodává funkce prázdný řetězec.

Příklad: Předpokládejme, že existuje proměnná `JMENO` a chceme přečíst její hodnotu. Potřebujeme pracovat s řetězcí typu znakového pole (viz příslušná kapitola), zapíšeme:

```
339 | char Hodnota[100]; //znakové pole = řetězec
340 | Hodnota = getenv("JMENO");
```

```

341 |     if (strlen(Hodnota)!=0) //proměnná existuje, lze zpracovat
342 |     else cerr << "Proměnná JMENO nebyla nalezena.";

```

9.7 Rekurze

Technický princip **rekurze** je docela jednoduchý: podprogram ve svém těle obsahuje volání sama sebe. Kromě tohoto případu tzv. přímé rekurze existuje ještě i **nepřímá rekurze**: podprogram A obsahuje volání podprogramu B a ten obsahuje volání podprogramu A.

9.7.1 Analýza rekurze

Úloha vyžadující opakování (například součet N členů zadané řady) má zadání, které se podobá tomuto:

$$S_N = a_1 + a_2 + \dots + a_N,$$

kde tři tečky naznačují použití cyklu.

```

343 | int a, s=0;      //deklarace a inicializace
344 | while (cin>>a)  //čtení
345 |     s = s + a;  //součet
346 | cout << s;     //výpis výsledku

```

Chceme-li tutéz úlohu řešit rekurzivním způsobem, je potřebné zadání poněkud přeformulovat:

$$\begin{aligned} S_N &= S_{N-1} + s_1 \\ S_0 &= 0 \end{aligned}$$

Součet je formulován také pomocí součtu – to je princip rekurzivního zadání. Pro úplnost je potřebné vždy kromě rekurzivního pravidla doplnit i pravidlo nerekurzivní – ve druhém řádku je řečeno, že součet nulového počtu členů je roven nule. Tímto pravidlem je umožněno rekurzi ukončit.

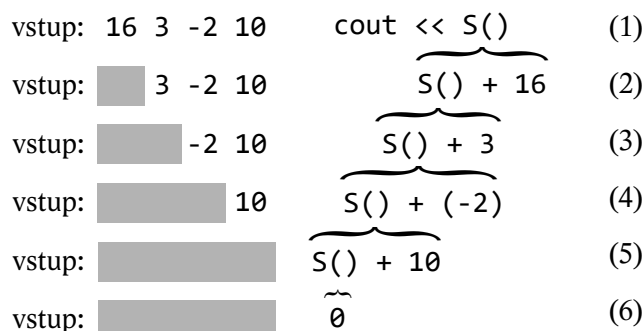
```

347 | int S(){
348 |     int a;      //deklarace proměnné
349 |     if (cin>>a)  //čtení
350 |         return S()+a; //součet
351 |     else return 0;  //inicializace
352 | }
353 | cout << S();    //výpis výsledku

```

Pro objasnění, jak probíhá výpočet, znázorníme jednotlivé fáze na obr. 13. První příkaz, který se začne provádět, je výpis výsledku pomocí `cout`. V něm je ovšem obsaženo volání funkce `S()` – to se

provede. Protože v této chvíli není konec vstupu, přečte se vstupní hodnota 16, uloží se do lokální proměnné a provede se příkaz `return S() + a;`. Tam je ovšem napřed potřebné vyčíslit výraz a v něm je opět volání funkce `S()`. Dojde k dalšímu volání, další deklaraci lokální proměnné, dalšímu čtení atd. V okamžiku, kdy je takto „rozpracováno“ pět volání funkce `S()`, dojde opět k jejímu vyvolání, tentokrát ovšem na vstupu už nic není, využije se tedy větev `else`, funkce skončí s hodnotou nula a vše se vrací k rozpracovanému výrazu v řádku (5). Tam může nyní dojít k součtu ($0 + 10$), dokončí se tato funkce a vydá výslednou hodnotu, která se dosadí do výrazu na řádku (4) atd. Proces postupného volání se často nazývá *zanořování* rekurze, proces postupného ukončování volaných podprogramů se pak nazývá *vynořování* rekurze.



Obrázek 13: Postup rekurzivního výpočtu

9.7.2 Srovnání rekurze a iterace

Pojmem **iterace** označujeme obecně nějaké opakování, v užším smyslu však myslíme opakování řešení pomocí cyklu. Protože rekurze je také jistý druh opakování, budeme se nyní zabývat porovnáním těchto dvou možností.

V příkladu se součtem hodnot si můžeme všimnout na první pohled, že rekurzivní zápis je poněkud složitější. Při bližším zkoumání zjistíme, že navíc při každém zavolání rekurzivní funkce dojde k deklaraci nové lokální proměnné `a` a ještě také víme, každé volání podprogramu způsobí uložení návratové adresy do systémového zásobníku. To se v případě iterativního řešení neděje, veškeré manipulace s daty se odehrávají ve dvou proměnných.

Zdá se, že rekurzivní řešení neposkytuje žádnou výhodu – ano, v tomto případě je to pravda. Zápis algoritmu je složitější a fakt, že všechny vstupní hodnoty jsou umístěny do paměti, je sice zajímavý, ale tuto vlastnost u součtu hodnot nepotřebujeme.

Existují ovšem úlohy, u nichž je situace jiná. Mějme například vstupní řadu čísel vypsát na výstup v obráceném pořadí, tj. například z posloupnosti 16 35 -2 10 máme získat 10 -2 35 16. Pokud bychom neuvažovali rekurzi, musíme napřed všechny hodnoty uložit do paměti (do pole – viz příslušná kapitola) a pak je v požadovaném pořadí vypisovat. Ovšem rekurzivní řešení by mohlo vypadat například takto:

```
354 void Obrat(){
355     float c;
```

```
356     if (cin>>c) {  
357         Obrat();  
358         cout << c << " ";  
359     }  
360 }  
361  
362 Obrat();
```

Při zanořování rekurze se vždy alokuje jedna číselná proměnná, do ní se přečte vstupní hodnota – pokud ještě nějaká na vstupu je – a znovu se zavolá procedura `Obrat()`. V okamžiku, kdy je celý vstup přečten a podmínka příkazu `if` nebude platná, dojde k vynořování rekurze: postupně se vypisují obsahy proměnných v pořadí vynořování, které je ovšem opačné než zanořování, což vede k výpisu hodnot v opačném pořadí než při čtení.

Srovnáme-li iterativní a rekurzivní řešení, dojdeme k závěru, že každé je vhodné na určitý typ úloh. U rekurze je výhodou, pokud úloha vyžaduje úschovu zpracovávaných údajů v paměti, což u iterace automaticky není. Stručně můžeme pak říct, že rekurze je opakování s využitím systémového zásobníku, jinými slovy, iterace je chudší (štíhlejší) sestra rekurze.

10 Strukturované datové typy – pole

Dosud jsme ve všech algoritmech vystačili s proměnnými, které nazýváme **jednoduché** – v jednom okamžiku obsahují jednu hodnotu. Je ale nepochybné, že potřebujeme zpracovávat a uchovávat i velká množství hodnot (stovky, tisíce i řádově více) a na první pohled je zřejmé, že pro tato množství nemůžeme deklarovat stovky, tisíce nebo i větší množství jednotlivých proměnných. Nejenže by samotná deklarace byla nesmírně rozsáhlá, těžkopádná a nepřehledná, ale i jakákoliv další (hromadná) práce s množstvím jednoduchých proměnných není prakticky možná.

Proto existují datové typy reprezentující celou skupinu hodnot tvořících jeden celek. Jde o tzv. **strukturované datové typy** a strukturované proměnné.

10.1 Koncept pole

Nejrozšířenější koncept strukturované proměnné představuje **pole**. Jde o strukturu složenou z určitého počtu prvků stejného typu, například 10 velikostí šroubů s půlkulatou hlavou, 587 zaměstnanců podniku, 20 milionů pixelů obrazu v pravých barvách atd.

Pole uplatníme v případě, že potřebujeme v paměti počítače uchovávat více hodnot *současně*. U algoritmů, které jsme zmiňovali v dřívějších kapitolách, jsme také zpracovávali více hodnot, ale nepotřebovali jsme je všechny uchovávat. Pro zpracování stačila jedna právě přečtená (případně dvě po sobě jdoucí apod.), k tomu je použití pole zcela zbytečné (až kontraproduktivní).

Pole najdeme téměř v každém programovacím jazyce od historických padesátých let 20. století do dneška. Pole je totiž svým způsobem přímý obraz operační paměti, jeho složky jsou „naskládány“ za sebou jako jednotlivá paměťová místa, a také se s nimi podobným způsobem pracuje. Chceme-li se dostat k obsahu nějaké buňky v paměti, zadáme její adresu (což je vlastně pořadové číslo té buňky od začátku paměti). Úplně stejně jsou očíslovány i jednotlivé prvky pole. Místo adresy se v případě pole ovšem používá pojem **index** = pořadové číslo prvku pole.

Deklarace proměnné typu pole je velmi jednoduchá: k „obyčejné“ deklaraci se pouze do hranatých závorek dopíše, kolik prvků (složek) má pole mít, např. `float srouby[10];`

Přístup ke složce pole spočívá v přepočtu hodnoty indexu na paměťovou adresu, na níž se daná složka nachází. Na obr. 14 je vidět schéma pole, které je deklarováno zmíněným příkazem. Identifikátor pole představuje adresu začátku pole v paměti, tj. zároveň i adresu prvního prvku pole.

Přístup ke složce se zapisuje jako hodnota v hranatých závorkách (index) u identifikátoru pole. Jako index může být zapsán jakýkoliv celočíselný výraz. Na obrázku je ukázána vazba mezi indexem a adresou daného prvku v paměti. Při deklaraci se uvádí datový typ složky, takže překladač může pak určit podle velikosti složky, jak se budou indexy přepočítávat. K tomu se používá jednoduchý vztah:

$$A = PA + (I - I_0) \cdot V,$$

kde A je požadovaná adresa složky, PA je počáteční adresa pole, I je hodnota indexu požadované složky, I_0 hodnota prvního indexu a V velikost složky v bajtech.

indexy	srouby	adresy v paměti	
0	20.34	289168	<pre>float srouby[10]; ... srouby[0] = 20.34; cin >> srouby[1]; srouby[2] = (srouby[0]+srouby[4])/2;</pre>
1	28.56	289172	
2	18.81	289176	
3	51.22	289180	
4	17.28	289184	
...	...	289188	
		...	

Obrázek 14: Koncept pole – deklarace, umístění v paměti, indexy, práce se složkami

Pole v jazyce C++ mají vždy počáteční index $I_0 = 0$, v uvedeném vztahu se v tom případě neprovádí žádný rozdíl a zadaný index se pouze násobí velikostí složky.

Další možnou optimalizací výpočtu adresy může být vhodná velikost složky. Je-li velikost vyjádřitelná jako některá mocnina dvojky (například 4, 8, 128 apod.), nemusí se provádět násobení, ale nahradí se pouze bitovým posuvem vlevo o hodnotu této mocniny.

Při výpočtu adresy se rovněž kontroluje, zda hodnota indexu není mimo povolený rozsah, tj. mimo interval $\langle 0, N - 1 \rangle$ pro pole o N složkách. Má-li kompilátor tuto variantu generování kódu zapnutou a pokusíme se o indexaci mimo rozsah pole, je ohlášena běhová chyba.

Přístup ke složkám pole je velmi efektivní a těžko lze najít strukturu, která by pracovala jednodušeji a rychleji. Pro tyto vlastnosti bylo (a doposud je) pole široce využíváno.

Určitou nevýhodou konceptu pole však je, že musíme v okamžiku deklarace určit, kolik složek má pole mít. Pokud je množství dat velmi variabilní a nemůžeme je kvalifikovaně odhadnout, musíme zvolit dostatečně velký počet složek proto, aby se nestalo, že zpracovávaná data do pole nevejdou. To může vést k určitému plýtvání paměti, protože většinou obsadíme jen malou část a extrémní množství nastane vzácně. Existuje i koncept tzv. **dynamického pole**, tj. pole, jehož počet složek lze v průběhu používání změnit (navýšit, ale i snížit), a tím optimalizovat spotřebu paměti. Je ovšem potřeba poznamenat, že takové možnosti jsou mnohdy velmi draze zaplacené podstatně méně efektivním přístupem ke složkám, takže se spíše vyplatí zabrat více paměti než plýtvat časem, který si na rozdíl od větší paměti nemůžeme nikde koupit.

10.2 Algoritmy využívající pole

Ukážeme některé typické manipulace s polem. Jako první bude algoritmus, který naplní pole hodnotami získanými ze standardního vstupu.

10.2.1 Naplnění a zpracování hodnot pole

Přesnější zadání: vstup je tvořen řadou desetinných čísel. Neznáme jejich počet a ani nevíme, která hodnota je poslední. Úkolem je vložit všechna vstupní čísla do pole.

Způsob řešení: Po skončení tohoto algoritmu očekáváme, že bude-li na vstupu K hodnot, budou v poli naplněny složky s indexy 0 až $K - 1$. Na začátku deklarujeme pole, u něhož musíme zvolit počet složek. Protože neznáme množství vstupních dat, jde o volbu orientační – volíme *dostatečný počet* N . V některých případech je nutné ošetřit i situaci, že množství vstupních dat převyšuje kapacitu pole. Pak po skončení algoritmu předpokládáme, že pole je naplněno hodnotami ve složkách s indexy 0 až $\min(K - 1, N - 1)$.

```

363 const unsigned int N = 10000; //kapacita pole
364 typedef float TypPole[N]; //datový typ pole
365 unsigned int Pocet=0; //počet naplněných složek pole
366 TypPole P;
367 while (Pocet<N and cin >> P[Pocet])
368     Pocet++;

```

Proměnná `Pocet` udává po skončení cyklu skutečný počet naplněných složek pole; poslední platný index má hodnotu `Pocet-1`. Na tuto skutečnost je potřebné vždy při následné práci s polem myslet, často totiž vzniká chyba záměnou významů: počet složek `NENÍ` totéž, co poslední platný index.

Druhým algoritmem může být výpis obsahu pole: vypíšeme všechny naplněné složky na standardní výstup, jednou to bude v pořadí, v jakém byly hodnoty přečteny, podruhé to bude v obráceném pořadí. S výhodou využijeme cyklus `for`, protože víme, kolik složek budeme zpracovávat. Předpokládáme předchozí definice a deklarace:

```

369 for (unsigned int i=0; i<Pocet; i++)
370     //i<Pocet znamená, že poslední průchod je pro i=Pocet-1
371     cout << P[i] << " "; //výpis v pořadí čtení
372 cout << endl;
373 for (unsigned int i=Pocet-1; i>=0; i--)
374     cout << P[i] << " "; //výpis v obráceném pořadí
375 cout << endl;

```

Uvedené algoritmy můžeme poněkud zobecnit: jsou analogicky použitelné v každém případě, kdy potřebujeme s každým prvkem pole udělat nějakou operaci – například vynásobit konstantou (ska-

lární součin vektoru), přepočítat na nějakou relativní hodnotu vzhledem ke zvolené vztažné hodnotě, provést součet všech prvků atd.

10.2.2 Dvoupřůchodové algoritmy

Pole potřebujeme u tzv. dvoupřůchodových algoritmů, to jsou algoritmy, které potřebují pro získání výsledku dvakrát zpracovat vstupní data. Protože vstup nelze číst vícekrát než jednou, jde vždy v prvním kroku o úschovu vstupních dat do pole, v dalším průchodu pak o zpracování uložených dat. Jedním z nejjednodušších příkladů dvoupřůchodového algoritmu je zjištění počtu nadprůměrných hodnot ve vstupní řadě. Pro jednoduchost nyní předpokládejme, že kapacita pole je vždy zcela dostačující a na vstupu nikdy víc hodnot nebude.

```

376 const unsigned int N = 10000; //kapacita pole
377 typedef float TypPole[N]; //datový typ pole
378 unsigned int Pocet=0, //počet naplněných složek pole
379     PocetNad=0; //počet nadprůměrných hodnot
380 TypPole P;
381 float Soucet=0, Prumer;
382 while (cin >> P[Pocet]){ //čtení a úschova
383     Soucet += P[Pocet]; //získání součtu
384     Pocet++;
385 }
386 Prumer = Soucet/Pocet; //po skončení cyklu lze spočítat průměr
387 for (unsigned int i=0; i<Pocet; i++) //průchod polem
388     if (P[i]>Prumer) PocetNad++; //složka s nadprůměrnou hodnotou
389 cout << PocetNad << endl; //výpis výsledku

```

10.2.3 Vyhledávání a řazení

Algoritmy těchto skupin patří mezi nejfrekventovanější a také nejpropracovanější. Budeme se na tomto místě zabývat jen těmi nejjednoduššími variantami, důkladnější rozbor je uveden později v příslušných kapitolách.

Úloha vyhledání je obvykle zadána takto: Existuje pole P naplněné N hodnotami a je zadána hodnota C – výsledkem vyhledání je logická hodnota určující, zda se C nachází v poli P .

Nejjednodušší vyhledávací algoritmus – **sekvenční hledání** – postupuje tak, že prochází pole od začátku tak dlouho, dokud se nevyčerpají všechny složky nebo dokud není nalezena hledaná hodnota. Zápis algoritmu předpokládá již naplněné pole N hodnotami (indexy od nuly do $N - 1$):

```

390 bool Nalezeno;
391 unsigned int Kde=0;
392 while (Kde<N and P[Kde]!=C) Kde++;
393 Nalezeno = Kde<N;

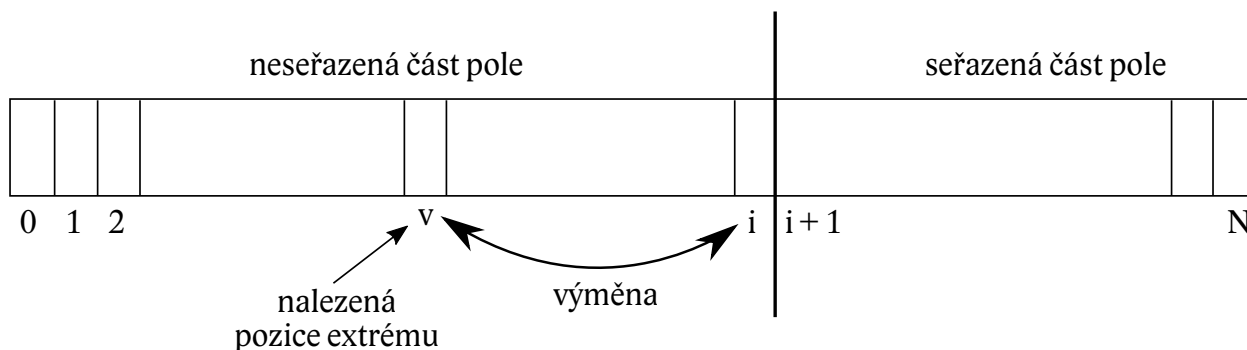
```

Řazení je proces, při němž se uspořádají hodnoty v poli podle velikosti.⁶ Pojem velikost zde znamená jakoukoliv charakteristiku dané položky (číselná velikost, délka řetězce, hodnota první číslice za desetinnou čárkou a mnoho dalších), kterou lze porovnávat.

Z mnoha algoritmů pro řazení hodnot uvedeme dva, které sice nevynikají žádnými vlastnostmi, ale jsou jednoduché a pro ilustraci práce s polem přijatelné.

Prvním z nich je algoritmus nazvaný **přímý výběr**. Můžeme jej popsat slovně takto: Mějme například hromádku karet rozhozenou na stole (to jsou počáteční data). Z této hromádky vybereme nejvyšší kartu a položíme ji stranou na místo, kde očekáváme výsledek řazení. Pak znovu z hromádky vybereme nejvyšší kartu a položíme ji opět stranou vedle té již odložené. Takto budeme postupovat tak dlouho, dokud jsou v hromádce nějaké karty. Postupně se na straně, kde odkládáme vybrané karty, tvoří uspořádaná posloupnost.

V počítačové implementaci budeme pracovat s polem hodnot (viz obr. 15). Na počátku bude celé pole naplněno neuspořádanými hodnotami. Vybereme nejvyšší hodnotu, uložíme ji na konec (do poslední složky) a v dalším kroku už s touto složkou nebudeme pracovat. Původní hodnotu z poslední složky umístíme na místo nejvyšší vybrané hodnoty. Dále vybereme nejvyšší hodnotu z části mezi indexy 0 a $N - 1$ a opět ji vyměníme s hodnotou na indexu $N - 1$. Na obr. 15 vidíme situaci, kdy zbývá ještě rozsah od nuly do i -tého indexu neseřazený a zbytek pole obsahuje již seřazené hodnoty.



Obrázek 15: Řazení metodou přímého výběru – schéma činnosti

Předpokládejme, že na vstupu máme desetinná čísla a chceme je seřadit. Celý algoritmus můžeme zapsat například takto:

```

394 typedef unsigned int Indexy; //datový typ pro indexy v poli
395 const Indexy Max = 1000; //rozměr pole
396 typedef float Pole[Max]; //pole desetinných čísel
397
398 void Zamena(Pole P, Indexy A, Indexy B){
399 //vzájemná záměna hodnot složek v poli P na indexech A, B
400     float Pom;
401     if (A!=B) { //pokud se indexy nerovnají, vyměníme
402         Pom=P[A];

```

⁶ Často bývá řazení nesprávně nazýváno **třídění** – to je ovšem rozdělování do skupin (tj. klasifikace). Toto nesprávné pojmenování vzniklo nepřesným překladem z anglického pojmu „sort“ a ten vznikl pravděpodobně jako zkrácený popis jednoho z prvních algoritmů řazení, u něhož se používá třídění (rozdělení).

```

403     P[A]=P[B];
404     P[B]=Pom;
405 }      //pro shodné indexy se nedělá nic
406 }
407 Indexy Extrem(Pole P, Indexy A, Indexy B){
408 //nalezení pozice nejvyšší hodnoty v poli P v rozmezí indexů A a B
409     Indexy V = A; //index V uchovává pozici extrému, začínáme 1. indexem
410     for (Indexy I=A+1; I<=B; I++) //projdeme celý úsek od 2. indexu
411         if (P[I]>P[V]) V=I; //index nalezeného extrému uložíme do V
412     return V;
413 }
414 int main(){
415     Pole P; Indexy Pocet=0;
416     //naplnění pole ze vstupu:
417     while (cin>>P[Pocet]) Pocet++;
418     //vlastní řazení:
419     for (Indexy I=Pocet-1; I>=1; I--)
420         Zamena(P, Extrem(P, 0, I), I);
421     //výpis výsledku:
422     for (Indexy I=0; I<Pocet; I++)
423         cout << P[I] << " ";
424     return 0;
425 }

```

Druhou řadící metodou, kterou zmíníme, je velmi populární **bublinové řazení** (angl. bubble sort). Populární je zejména proto, že jeho princip je snadno pochopitelný a implementace velmi jednoduchá. Ukážeme si ale také jeho variantu vykazující v některých případech zajímavé vlastnosti a umožňující její aplikaci vedle metod jiných, daleko propracovanějších a komplikovanějších.

Základní princip: procházíme pole (nebo jinou lineární strukturu) a zjišťujeme, zda mají dva po sobě jdoucí prvky požadované pořadí. Chceme-li řadit vzestupně, zjišťujeme, zda prvek následující je skutečně větší než jeho předchůdce. Pokud tomu tak není, prvky přehodíme. Takto procházíme strukturu tak dlouho, dokud nejsou všechny prvky v odpovídajícím pořadí.

Počet průchodů polem s N prvky potřebných pro to, aby bylo seřazeno, je nejvýše $N - 1$. Nejjednodušší implementace má následující tvar:

```

426 //Předpokládejme opět datový typ Indexy, deklarované Pole
427 //a definovanou proceduru Zamena z předchozího algoritmu.
428 for (Indexy I=0; I<N-1; I++)
429     for (Indexy J=0; J<N-I-1; J++)
430         if (P[J]>P[J+1]) Zamena(P, J, J+1);

```

Jednodušší zápis řadící metody si snad ani nelze přát ... Tato varianta se však jeví jako nejhorší z hlediska použití. Aniž bychom nezbytně potřebovali, neustále se prochází pole a testuje pořadí prvků,

i když už možná je všechno dávno seřazeno a veškeré další manipulace jsou zbytečné. Proto ukážeme variantu, která je schopna zjistit, že je už hotovo a že už má skončit.

Podstatou této úpravy je test, že po provedeném průchodu již nebyla provedena žádná záměna. Použijeme logickou proměnnou, kterou nastavíme na **false**, necháme provést průchod polem a v případě, že bude potřeba některé prvky zaměnit, proměnnou nastavíme na **true**. Hodnotu této proměnné použijeme pro řízení hlavního řadicího cyklu (nahradíme tím cyklus s řídicí proměnnou **I**). Upravená verze má tvar:

```

431 bool jeste=true;
432 Indexy I=0;
433 while (jeste) {
434     jeste=false; //předpoklad: už je seřazeno
435     for (Indexy J=0; j<N-I-1; J++)
436         if (P[J]>P[J+1]) {
437             Zamena(P, J, J+1);
438             jeste = true; //ještě musíme pokračovat
439         }
440     I++;
441 }
```

Skutečnost, že seřazené pole se pozná dříve, předurčuje tuto metodu k použití například tam, kde většinou už máme hodnoty seřazené a občas tam přibude nějaká ojedinělá nová hodnota nebo ojedinělá změna existujících hodnot. Rovněž zajímavé na této metodě je, že pracuje výhradně se sousedními prvky, můžeme ji proto použít i tam, kde z nějakého důvodu nemůžeme libovolně sáhnout do kteréhokoliv místa struktury, ale máme možnost pouze sekvenčního přístupu (soubory, seznamy).

10.3 Znaková pole

Zvláštním případem pole je **znakové pole**, tedy pole, jehož složkou je datový typ **char**. Takové pole reprezentuje **znakový řetězec**. Je možné říct, že v každém jazyce se k problému znakových řetězců vyskytuje nějaké řešení, možných přístupů je více a téměř vždy jde o nějaké usnadnění práce s tímto typem dat. Důvod je jasný: ve své podstatě jde o strukturovaná data (posloupnost znaků), ale zároveň jde o tak frekventovanou a potřebnou strukturu, že je nezbytné její zpracování co nejvíce zjednodušit.

I v jazyce C++ můžeme vidět dva přístupy – jeden je převzat z původního jazyka C, druhý je pak typický pro C++, neboť toto řešení je objektové.

Znakový řetězec je v principu posloupnost znakových hodnot, takže tou nejpřirozenější implementací možností je pole znaků. Současně ale potřebujeme, aby se znakový řetězec choval jako jednoduchá proměnná, zejména aby se mohl stejně jako jednoduchá proměnná přečíst ze standardního (nebo obecně z textového) vstupu a vypsát do textového výstupu, aby bylo možné provést nějaké operace (porovnání, zřetězení) a ještě navíc přistupovat k jednotlivým znakům.

Budeme se nejdříve zabývat implementací řetězce převzaté z jazyka C (budeme tyto řetězce pracovně označovat jako C-řetězce). Koncepce těchto řetězců je vlivem masivního rozšíření jazyka C používána v obrovské šíři programového vybavení – prakticky pro jakoukoliv komunikaci prostřednictvím řetězců je tento typ ve hře.

Pokud je nutné s řetězcem pracovat jako s celkem, potřebujeme mít k dispozici informaci o aktuálním počtu znaků, musíme se nějak dovědět, kde řetězec končí. Koncepce C-řetězců z tohoto důvodu předepisuje, že posledním znakem musí být bajt s hodnotou nula. Často se setkáme i s označením „null-terminated string“ (řetězec zakončený nulou). Každá operace, která s řetězcem manipuluje, vždy zajistí, že tam ten nulový bajt bude. Musíme s jeho existencí počítat také při deklaraci – je nutné rezervovat o jeden bajt víc, než je požadovaná maximální délka řetězce.

Podívejme se na konkrétní jazykové řešení: Deklarace řetězce se od deklarace obyčejného jiného pole neliší, např.:

```
442 | int Cisla[40]; //40 složek pro celá čísla, indexy 0 až 39
443 | char Znaky[40]; //39 složek pro znaky, další složkou bude 0
```

Velkou odlišností jsou ovšem možnosti zpracování. Zatímco s polem `Cisla` jako s celkem nemůžeme udělat vůbec nic (lze manipulovat pouze s jednotlivými složkami), s polem `Znaky` můžeme:

- číst ze vstupu: `cin >> Znaky;`
- číst ze vstupu metodou `getline` s nařízeným počtem znaků nebo s udaným oddělovačem, např. `cin.getline(Znaky, 30)` nebo `cin.getline(Znaky, 40, ';')`
- vypisovat na výstup: `cout << Znaky;`
- přiřadit při deklaraci počáteční hodnotu formou řetězcového literálu:
`char Znaky[40] = "efekt hodnoty";`
- využít řadu funkcí z knihovny `cstring` určené pro manipulaci s řetězci:
 - zjištění délky `strlen(Znaky)`,
 - porovnání `strcmp(Znaky, "xxx")`,
 - kopie řetězce `strcpy(Znaky, "Nový obsah")`,
 - zřetězení řetězců `strcat(Znaky, " rozšířen")`,
 - nalezení podřetězce `strstr(Znaky, "obs")` atd.

Koncepce řetězců ve formě objektu typu `string` ještě více zjednodušuje práci s proměnnými jako celkem. Po deklaraci `string s;` lze například použít:

- čtení ze vstupu `cin >> s;` (to je stejné jako u C-řetězců),
- čtení ze vstupu pomocí procedury `getline` (pozor – to je něco jiného než metoda objektu `cin.getline`), např. `getline(s, 25)` nebo `getline(s, 30, ':')`,

- výpis na výstup `cout << s;` (opět stejné jako u C-řetězců),
- přiřazení hodnoty `s = "cokoliv";` – to lze použít nejen v okamžiku deklarace, ale kdykoliv později,
- zřetězení operátorem `+`, např. `s + " další";`, přičemž lze také využít složeného přiřazení `s += "přídavek";`,
- porovnání běžnými relačními operátory, např. `s <= "slovo"`,
- zjištění délky (počtu bajtů, na kterých jsou zobrazeny všechny znaky) metodou `s.length()`,
- získání datové složky objektu v podobě C-řetězce `s.c_str()`,
- výběr libovolného znaku na indexu `i`, např. `s.at(i)`, totéž lze udělat i použitím hranatých závorek `s[i]`,
- získání podřetězce `s.substr(odkud, počet)` atd.

10.4 Pole jako parametr podprogramu

Pole je v jazyce C/C++ poněkud nekonvenčním datovým typem ještě i z jiného úhlu pohledu: na rozdíl od všech ostatních typů je *vždy* umístěno v tzv. dynamické paměti a ve skutečnosti manipulujeme pouze s adresou této paměti (podrobnosti jsou vysvětleny v kapitole 12 zabývající se dynamickými datovými strukturami). Z tohoto faktu plyne řada zvláštností a syntaktických berliček, aby se uživatel nemusel zabývat prací s adresami, ale zároveň bylo systematicky zabráněno možnosti vytvářet rozsáhlé datové struktury v části paměti, která k tomu není primárně určena.

S manipulací pomocí adresy jsme se již setkali v případě parametrů volaných odkazem (viz 2). Princip předávání parametrů typu pole do podprogramu je úplně stejný – protože pole představuje vlastně jen ukazatel, předává se každé pole (bohužel nezměnitelně) *vždy odkazem*.

Uvnitř podprogramu musíme pak vždy počítat s tím, že každá změna v obsahu položek pole se projeví i ve skutečném parametru. Chceme-li tomuto stavu zabránit, lze to zařídit „ruční“ simulací volání hodnotou: deklarujeme novou proměnnou, do které zkopírujeme obsah pole a kterou následně předáme do podprogramu.

Kapacita pole předávaného jako skutečný parametr nemusí být stejná jako kapacita pole deklarovaného jako formální parametr – to je logické: předáváme přece pouze adresu pole, nikoliv pole jako takové. Tuto situaci můžeme zobecnit tak, že při deklaraci formálního parametru nedáváme do hranaté závorky žádnou hodnotu. Překladač tak dostane informaci o tom, že tímto parametrem je pole, u něhož je znám datový typ položek, ale kolik složek bude mít skutečný parametr, bude určeno až v okamžiku volání.

Při předávání pole do podprogramu je nutné vždy pamatovat také na to, aby uvnitř podprogramu existovala dostatečná informace o tom, jaké složky pole jsou obsazeny a s nimiž se dá pracovat.

Předáme-li v podstatě pouze ukazatel (adresu) nějakého paměťového úseku, neexistuje žádná spolehlivá metoda, jak zjistit, co se na tomto paměťovém úseku nachází.

Typicky se s každým předávaným polem proto předává i celočíselná hodnota, která indikuje buď celkový počet obsazených (použitých) položek pole, nebo indikuje poslední obsazený index (pozor na záměnu těchto dvou významů – víme, že jde o *rozdílné hodnoty*). Předání znakového pole tuto informaci nepotřebuje, protože konec je rozpoznán oním speciálním bajtem s hodnotou nula.

Výjimkou z uvedeného pravidla jsou situace, kdy počet zpracovávaných položek známe odjinud nebo jej znát nepotřebujeme (to záleží na algoritmu zpracování, například víme, že pole je zcela naplněno ve všech složkách).

Pro předávání parametrů typu pole je velmi vhodné definovat potřebný datový typ pole, s nímž máme k dispozici informaci o datovém typu složek i o kapacitě pole, a tím výrazně zjednodušíme a zpřehledníme zápis programu.

10.4.1 Příklady

Mějme vektor nezáporných desetinných čísel přečtených ze standardního vstupu. Úkolem bude napsat podprogram, který dokáže vyjádřit (přepočítat) všechny hodnoty v daném vektoru v relativní podobě v procentech vzhledem k maximální vstupní hodnotě. Takže například pro vstupní data

0.9 1.6 2.0 0.53 1.1

bychom měli získat výsledný vektor s hodnotami

45.0 80.0 100.0 26.5 55.0.

Kompletní program může mít následující tvar:

```
444 #include <iostream>
445 using namespace std;
446
447 typedef unsigned int Index; //datový typ pro indexaci a počet položek
448 const Index Rozmer = 100; //kapacita pole
449 typedef float TypVektor[Rozmer]; //datový typ pole
450
451 void Norma(TypVektor V, Index Obsaz){ //procedura pro přepočet vektoru
452 //využíváme fakt, že pole je předáváno v podstatě odkazem
453 //druhým parametrem předáváme počet obsazených složek pole
454     float Max=0; //pomocná proměnná pro maximální složku
455     for (Index i=0; i<Obsaz; i++)
456         if (V[i]>Max) Max=V[i]; //nalezení maxima
457     for (Index i=0; i<Obsaz; i++)
458         V[i]=V[i]/Max*100; //vlastní přepočet hodnot
459     //nové hodnoty se objevují i ve složkách skutečného parametru
460 }
461
```

```

462 int main(){
463     TypVektor A;      //pracovní pole
464     Index Pocet=0;    //počet obsazených složek
465     while (Pocet<Rozmer and cin>>A[Pocet]) Pocet++;
466                     //přečteno nejvýše tolik čísel, kolik je kapacita pole
467                     //první v podmínce musí být test na platnost indexu
468     Norma(A, Pocet); //přepočet hodnot na procenta
469     for (Index i=0; i<Pocet; i++)
470         cout << A[i] << " "; //výpis výsledku
471     cout << endl;
472     return 0;
473 }

```

Ve druhém příkladu budeme řešit následující úlohu: Předpokládejme, že na standardním vstupu je připravena řada desetinných čísel. Máme vytvořit funkci, která určí jejich medián, tj. hodnotu, která rozděluje seřazená data na poloviny.

Jde o poněkud komplexnější řešení, na němž můžeme ukázat oba způsoby předávání pole jako parametr podprogramů. Samotný výpočet je jednoduchý a pro jeho vysvětlení bezpochyby postačí komentáře ve zdrojovém textu.

```

474 #include <iostream>
475 using namespace std;
476
477 typedef unsigned int Index; //odvození datového typu pole
478 const Index MaxPocet=1000;
479 typedef float TypCisla[MaxPocet];
480
481 Index Maximum(TypCisla A, Index Kam){
482     //funkce zjistí pozici maxima v poli A mezi indexy 0 a Kam
483     //pole A se nemodifikuje, nemusíme řešit způsob předání
484     Index Max=0; //první složka je na začátku považována za maximum
485     for (Index i=1; i<=Kam; i++)
486         if (A[i]>A[Max]) Max=i;
487     return Max;
488 }
489
490 void Zamena(TypCisla P, Index A, Index B){
491     //záměna prvků v poli P na pozicích A a B
492     //pole P se má modifikovat, předání odkazem to umožňuje
493     float Pom=P[A];
494     if (A!=B) { //indexy nejsou stejné -- proběhne záměna
495         P[A]=P[B];
496         P[B]=Pom;

```

```
497     }
498 }
499
500 void Serad(TypCisla Data, Index Obsaz){
501     //seřazení metodou přímého výběru
502     //pole Data se má modifikovat -- opět chtěný stav, předání je v pořádku
503     for (Index i=0; i<Obsaz-1; i++){
504         Zamena(Data, Maximum(Data, Obsaz-i-1), Obsaz-i-1);
505     }
506 }
507
508 float Median(TypCisla Data, Index Obsaz){
509     //pole Data nemá být změněno. Zde potřebujeme simulovat předání hodnotou.
510     //Proto se provede jeho kopie:
511     TypCisla V;    //pomocné lokální pole
512     //Simulace parametru předaného hodnotou:
513     for (Index i=0; i<Obsaz; i++) V[i]=Data[i]; //kopie dat
514     Serad(V, Obsaz);    //seřazení lokálního pole
515     return V[Obsaz/2]; //zjištění výsledku
516     //pole Data bylo uchráněno před poškozením, skutečný parametr bez změny
517 }
518
519 int main(){
520     TypCisla Vstup;
521     Index Pocet=0;
522     //přečtení vstupních dat:
523     while (Pocet<MaxPocet and cin>>Vstup[Pocet]) Pocet++;
524     //výpis výsledku:
525     cout << "Medián vstupních dat je " << Median(Vstup, Pocet) << endl;
526     return 0;
527 }
```

10.5 Vícerozměrná pole

V jazyce C/C++ existují pouze jednorozměrná pole, ale to není žádnou překážkou pro tvorbu polí, která potřebujeme v různých aplikacích a která mají dva nebo více rozměrů. Těto vlastnosti je dosaženo jednoduchým způsobem: složkou pole je opět pole. Pro řadu různých matematických výpočtů máme pak k dispozici matice (dvourozměrná pole) nebo i složitější struktury.

Z jazykového hlediska se nejedná o nic zvláštního nebo jiného oproti již zmíněným prostředkům. K poli, které je složkou jiného pole, přistupujeme opět pomocí indexu v hranatých závorkách, a pro přístup ke složkám tohoto vnořeného pole opět použijeme index v hranatých závorkách.

Jako příklad můžeme uvést jednoduchou matematickou úlohu: Na standardním vstupu je připraveno 30 celých čísel představujících hodnoty dvou matic o rozměrech 3 řádky a 5 sloupců. Vypočtete součet těchto matic a vypište přehledně výslednou matici na standardní výstup.

Řešení předpokládá znalost principu součtu matic – prvek výsledné matice na pozici i,j je roven součtu prvků na těchto pozicích v obou vstupních maticích. Celý program s potřebnými komentáři má následující tvar:

```

528 #include <iostream>
529 #include <iomanip>
530 using namespace std;
531
532 typedef unsigned int Index;
533 const Index MaxRadku=3; //rozměry matice
534 const Index MaxSloupcu=5;
535 typedef int TypRadek[MaxSloupcu]; //jeden řádek matice
536 typedef TypRadek Matice[MaxRadku]; //pole složené z řádků = matice
537
538 void Precti(Matice X){
539     //nejjednodušší varianta čtení: známe počet hodnot
540     for (Index i=0; i<MaxRadku; i++) //pro všechny řádky
541         for (Index j=0; j<MaxSloupcu; j++) //čtení jednoho řádku
542             cin >> X[i][j]; //přístup ke složce pole uvnitř jiného pole
543 }
544
545 void Secti(Matice X, Matice Y, Matice Z){
546     for (Index i=0; i<MaxRadku; i++)
547         for (Index j=0; j<MaxSloupcu; j++)
548             Z[i][j] = X[i][j] + Y[i][j]; //součet všech stejnohlých prvků
549 }
550
551 void Vypis(Matice A){
552     for (Index i=0; i<MaxRadku; i++) { //výpis všech řádků
553         for (Index j=0; j<MaxSloupcu; j++) //výpis jednoho řádku
554             cout << setw(10)<< right << A[i][j]; //číslo na 10 pozic vpravo
555         cout << endl; //po výpisu jednoho řádku je přechod na nový řádek
556     }
557 }
558
559 int main(){
560     Matice A, B, C;
561     Precti(A); Precti(B);
562     Vypis(A);
563     Vypis(B);

```

```

564     Secti(A, B, C);
565     Vypis(C);
566     return 0;
567 }

```

Z uvedeného příkladu je patrné, že chceme-li pracovat se všemi prvky matice, použijeme dva do sebe vnořené počítané cykly, které vystřídají postupně všechny indexy. Ne vždycky však potřebujeme pracovat takto jednoduše se všemi prvky, jak je patrné v následujícím příkladu:

Na vstupu je připravena řada desetinných čísel, jejichž počet není předem znám. Vložte tato čísla po řádcích do matice o rozměrech 5×7 . Pokud bude vstupních čísel méně, než je potřeba, doplňte chybějící pozice v matici nulami.

Při řešení této úlohy lze postupovat dvěma cestami:

V prvním způsobu řešení napřed naplníme celou matici nulami, následně budeme číst data ze vstupu. Zde již nemůžeme použít dva vnořené počítané cykly, protože počet hodnot není znám. Musíme využít podmíněných cyklů a zjišťovat, zda ještě jsou na vstupu nějaká data. Důležitou informací, která z tohoto algoritmu ještě vystupuje, je pozice posledního načteného prvku – je na indexu `[i, j-1]`. Pokud však byla naplněna celá matice, hodnota `i` už není platným indexem.

```

568 //předpokládejme definici typu Matice o rozměrech 5 x 7 prvků
569 //konstanta MaxRadku je rovna 5, MaxSloupcu je rovna 7
570
571 void Preci(Matice M){
572     Index i, j;
573     for (i=0; i<MaxRadku; i++) //pro všechny řádky
574         for (j=0; j<MaxSloupcu; j++) //zpracování jednoho řádku
575             M[i][j] = 0; //vynulování celé matice
576     i = 0; j = 0;
577     while (i<MaxRadku and cin>>M[i][j]) //dokud je volno a je co číst
578     {j++; //připravíme index na následující volnou pozici
579         if (j==MaxSloupcu) { //řádek je zaplněn
580             j=0; //první pozice následujícího řádku
581             i++; //řádková souřadnice o jedničku vyšší
582         }
583     }
584 }

```

Druhou cestou je možnost využít sice dvou vnořených počítaných cyklů, ale místo obyčejného čtení musíme testovat, zda jsou ještě nějaká data na vstupu. Podle toho se pak rozhodneme buď pro čtení další hodnoty, nebo pro přiřazení nuly.

```

585 void Preci(Matice M){
586     bool Cteni=true; //ještě jsou na vstupu data
587     for (Index i=0; i<MaxRadku; i++)

```

```

588     for (Index j=0; j<MaxSloupcu; j++) {
589         if (Cteni) { //na vstupu jsou data
590             Cteni=cin>>M[i][j]; //přečtení a nastavení informace
591             if (not Cteni) M[i][j]=0; //čtení se právě nepovedlo
592         } else M[i][j]=0;
593     }
594 } //matice naplněna, nevíme však, kde je poslední načtený prvek

```

V dalším příkladu předpokládejme naplněnou celočíselnou matici o rozměrech R řádků a S sloupců. Máme napsat funkci, která zjistí, kolik prvků na hlavní diagonále má nulovou hodnotu.

Při řešení této úlohy je potřebné si uvědomit, že souřadnice prvků na hlavní diagonále mají stejnou řádkovou i sloupcovou hodnotu. Dále vezmeme v úvahu, že počet prvků hlavní diagonály je roven menší hodnotě z počtu řádků a počtu sloupců. Samotný algoritmus je jedním z elementárních algoritmů určující počet vybraných hodnot (v tomto případě nulových):

```

595 Index min(Index A, Index B){ //zjištění minima z hodnot A, B
596     if (A<B) return A;
597     else return B;
598 }
599
600 Index Kolik(Matice X, Index R, Index S){
601     Index Pocet = 0;
602     for (Index i=0; i<min(R, S); i++) //projdeme hlavní diagonálu
603         if (X[i][i]==0) Pocet++; //počet vybraných hodnot
604     return Pocet;
605 }

```

Následující příklad kombinuje další možnosti práce s maticí: Mějme naplněnou čtvercovou matici řádu N . Upravme tuto matici tak, aby její maximální prvek byl na pozici 0, 0, ale aby se nezměnila hodnota matice.

Pro řešení potřebujeme vědět, že hodnota matice je počet lineárně nezávislých řádků. Úpravou, která nezmění hodnotu matice, může být vzájemná záměna řádků nebo sloupců. Najdeme-li tedy pozici maximálního prvku x, y , pak výměnou řádků 0 a x a výměnou sloupců 0 a y dosáhneme požadovaného tvaru.

Zápis řešení provedeme formou tří podprogramů – nalezení souřadnic maximálního prvku, výměna řádků, výměna sloupců. Předpokládejme, že řád matice je stanoven konstantou `Rad`.

```

606 void Max(Matice M, Index &x, Index &y){
607     x=0; y=0; //počáteční pozice extrému
608     for (Index i=0; i<Rad; i++)
609         for (Index j=0; j<Rad; j++)
610             if (M[i][j]>M[x][y]) { //nalezeno větší číslo
611                 x=i; y=j; //uschováme jeho pozici

```

```
612     }
613 }
614
615 void VymenRadky(Matice M, Index A, Index B){
616     if (A!=B) { //bude se vyměňovat
617         float Pom; //pomocná proměnná pro výměnu dvou prvků
618         for (Index i=0; i<Rad; i++) { //projdeme všechny prvky obou řádků
619             Pom=M[A][i];
620             M[A][i]=M[B][i];
621             M[B][i]=Pom;
622         }
623     }
624 }
625
626 void VymenSloupce(Matice M, Index A, Index B){
627     if (A!=B) { //bude se vyměňovat
628         float Pom; //pomocná proměnná pro výměnu dvou prvků
629         for (Index i=0; i<Rad; i++) { //projdeme všechny prvky obou sloupců
630             Pom=M[i][A];
631             M[i][A]=M[i][B];
632             M[i][B]=Pom;
633         }
634     }
635 }
```

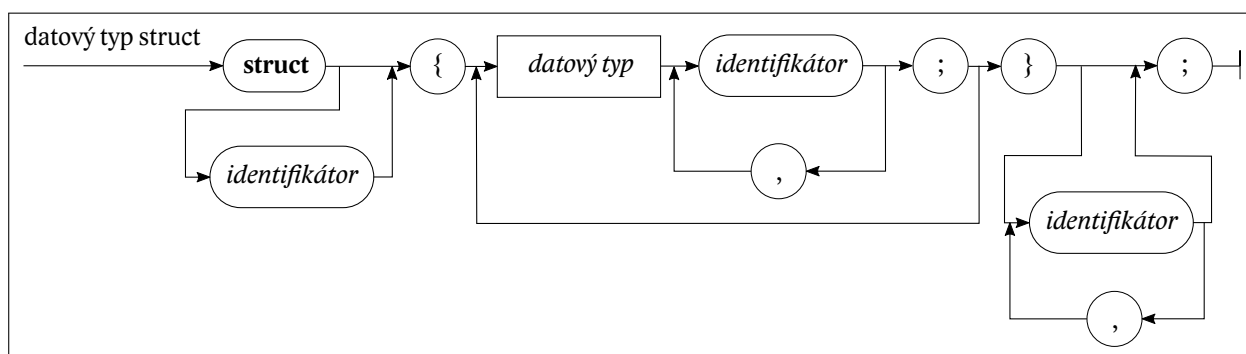
11 Záznamy

11.1 Datový typ záznam

Datový typ pole představuje strukturu, která je složena z položek stejného typu. Někdy však potřebujeme dát do jednoho celku položky rozdílných typů. K tomuto účelu slouží datový typ **struct**. Tento typ budeme pracovníčně nazývat pojmem **záznam**. Záznam použijeme tehdy, mají-li složky spolu nějakou logickou souvislost – například jde o atributy nějaké osoby (jméno, příjmení, výška v metrech, hmotnost v kilogramech), věci (název, počet kusů, cena za kus), procesu (pojmenování, počáteční čas, koncový čas, spotřebovaná energie) apod.

Hned na začátku je potřeba upozornit na rozdílnou implementaci tohoto typu v jazyce C a v jazyce C++. Zatímco v jazyce C se jedná o čistě datovou záležitost s posloupností datových složek, v jazyce C++ jde o téměř identický prvek, jakým je třída (**class**). Protože třídy a objekty jsou obsahem jiného předmětu, budeme se v následujícím textu zabývat pouze pohledem neobjektovým – původním konceptem pocházejícím z jazyka C.

Podívejme se na způsob zápisu datového typu **struct**:



Zápis začíná klíčovým slovem **struct**. Jádrem struktury je posloupnost položek uzavřená do složených závorek. Jednotlivé položky mají podobu deklarací – začínají datovým typem a za ním identifikátorem položky. Pokud je více položek za sebou stejného datového typu, může být za sebou uvedeno několik identifikátorů oddělených čárkami. Každá položka je zakončena středníkem.

Celý zápis může sloužit jako definice nového datového typu nebo jako deklarace proměnných tohoto typu (nebo obojí současně). Identifikátor nového typu se uvádí *před* seznamem položek, identifikátor proměnné (nebo identifikátory více proměnných) je *za* seznamem položek. Logika tohoto zápisu je shodná s definicemi pomocí klíčového slova **enum**.

Příklad definice typu záznam a deklarace proměnných:

```

636 struct Osoba { //datový typ Osoba
637     string Jmeno;
638     int Plat;
639     float Hmotnost;
640 };
  
```

```
641 | Osoba Franta, Lojza; //deklarace dvou proměnných typu Osoba
642 | struct { //datový typ nemá jméno
643 |     int Kusu;
644 |     float Cena;
645 | } Jablko, Mandarinka; //deklarace dvou proměnných typu záznam
```

Potřebujeme-li pracovat s jednotlivými složkami proměnné typu záznam, zapíšeme jméno proměnné, tečku a jméno složky, například `Franta.Jmeno` nebo `Mandarinka.Cena`.

Velmi užitečnou vlastností je, že záznamy stejného typu můžeme navzájem přiřazovat. To je i jeden ze zásadních důvodů, proč vůbec záznam používat – vystupuje jako jeden celek. Lze například zapsat:

```
646 | Lojza.Jmeno = "Alois Švihálek";
647 | Lojza.Plat = 27800;
648 | Lojza.Hmotnost = 82.5;
649 | Franta = Lojza; //Franta má nyní všechny položky stejné jako Lojza
```

Zajímavostí je, že přiřazení celých záznamů se může provést i tehdy, je-li jejich složkou libovolné pole, přestože samotná pole se přiřadit navzájem nemohou, i když jsou stejného typu. Reprezentaci pole můžeme poněkud zobecnit – pole bude s výhodou vytvořeno záznamem, jehož jednou složkou budou vlastní data pole a druhou složkou počet obsazených prvků. Takovou strukturu můžeme předávat jako parametr podprogramů a navzájem přiřazovat. Předávání záznamu jako parametr podprogramu nemá žádné zvláštnosti, jako je vidíme u pole.

Příklad datového typu představujícího zobecněné pole:

```
650 | typedef unsigned int TypIndex;
651 | const TypIndex Rozmer = 1000;
652 | typedef float TypData[Rozmer];
653 | struct TypPole {
654 |     TypData P;
655 |     TypIndex Obsaz;
656 | };
```

Předáním parametru typu `TypPole` do podprogramu hodnotou skutečně vede ke kopii složek, takže odpadají problémy s ochranou skutečného parametru proti nechtěné modifikaci.

Záznam se často používá jako složka pole. Získáváme tak typickou strukturu chápanou například jako databázovou tabulku. Protože záznamy lze navzájem přiřazovat, můžeme například snadno pole seřadit – zaměňovat celé složky pole.

Složkou záznamu může být jiný záznam. Záznamy tak lze do sebe vnořovat. Přístup ke složce vnořeného záznamu je analogický – použije se opět tečka a název složky.

Jednoduchým příkladem používajícím pole záznamů může být následující úloha: Na standardním vstupu jsou trojice údajů o skladových položkách: skladové číslo (řetězec), počet kusů a cena za kus.

Na standardní výstup je potřebné vypsat skladová čísla zboží s nadprůměrným počtem kusů na skladě a počet položek, jejichž oceněná zásoba (cena za kus násobena počtem kusů) nepřevyšuje 100 Kč.

Řešení spočívá ve dvou krocích: Nejprve přečteme vstupní data a uložíme do pole záznamů o výrobcích. Při tom můžeme spočítat průměrný počet kusů na skladě a zjistit počet výrobků, jejichž oceněná zásoba nepřevyšuje 100 Kč. Ve druhém kroku projdeme pole a u každého záznamu zjistíme, zda jeho počet kusů převyšuje vypočtený průměr. Pokud ano, vypisujeme na standardní výstup skladové číslo tohoto výrobku.

Kompletní program řešící tuto úlohu doplníme potřebnými komentáři:

```

657 #include <iostream>
658 using namespace std;
659
660 int main(){
661     struct TypZbozi { //datový typ záznam o zboží
662         string SklCis;
663         int PocKusu;
664         float CenaKus;
665     };
666
667     typedef unsigned int TypIndex;
668     const TypIndex MaxZbozi=1000;
669     typedef TypZbozi TypSklad[MaxZbozi]; //typ pole záznamů o zboží
670
671     TypSklad Sklad; //pole záznamů o zboží
672     TypIndex Pocet=0, PocMalo=0;
673     int CelKusu=0;
674     while (Pocet<MaxZbozi and //čtení, dokud není zaplněno pole...
675         cin >> Sklad[Pocet].SklCis >> Sklad[Pocet].PocKusu
676         >> Sklad[Pocet].CenaKus) { //...nebo dokud je co číst
677         CelKusu+=Sklad[Pocet].PocKusu;
678         if (Sklad[Pocet].PocKusu*Sklad[Pocet].CenaKus<100) PocMalo++;
679         Pocet++; //zvýšení počítadla musí být až na konci těla cyklu
680     }
681     float Prumer=float(CelKusu)/Pocet;
682     for (TypIndex i=0; i<Pocet; i++) //průchod polem záznamů o zboží
683         if (Sklad[i].PocKusu>Prumer) cout << Sklad[i].SklCis << endl;
684     cout << "Počet položek s oceněnou zásobou menší než 100 Kč je "
685         << PocMalo << endl;
686     return 0;
687 }
```

11.2 Datový typ variantní záznam

Pod pojmem **variantní záznam** budeme chápat strukturu, která má položky definované podobně jako u již zmíněného „obyčejného“ záznamu, ale její smysl, implementace a chování jsou zcela odlišné. Deklarované položky totiž na rozdíl od běžného záznamu *sdílejí stejný paměťový prostor*. Z toho pak vycházejí možnosti použití. Původním záměrem takového nástroje byla úspora paměťového prostoru na zobrazení položek, které se logicky vylučují (nemohou být použity současně). Chceme například uchovávat informace o materiálu – v případě hřebíků nás bude zajímat jejich délka, v případě oleje jeho teplotní odolnost, v případě sádry její hmotnost v daném sáčku apod. Je jasné, že údaj o délce nemá v případě oleje a sádry žádný smysl, podobně teplotní odolnost u hřebíků atd. Tyto položky se v různých variantách logicky vylučují. Aby nepoužitelné položky nezabíraly zbytečně místo, jsou všechny varianty uloženy na jedné adrese – sdílejí stejnou paměť. Celková velikost záznamu je pak dána velikostí největší varianty. Z tohoto smyslu této struktury pochází i zmíněný pojem „variantní záznam“.

Definice typu variantního záznamu vypadá úplně stejně jako běžného záznamu, jen místo klíčového slova **struct** je použito klíčové slovo **union**. Podobně jako obyčejné záznamy mohou být i variantní záznamy vnořovány různě do sebe. Toho se často využívá při konstrukci struktury, která obsahuje jak pevné, tak i variantní složky.

Ukažme si definici složky popisující uvedené materiály ve skladu:

```
688 union TypPopis { //variantní část záznamu o materiálu
689     float Delka;    //použije se u hřebíků
690     int Odolnost;   //použije se u oleje
691     float Hmotnost; //použije se u sádry
692 };
693
694 struct TypMater { //záznam o materiálu
695     string Nazev;   //název je u všech druhů materiálu
696     TypPopis Popis; //u popisu se použije vždy jen jedna položka
697 };
698 //příklad přiřazení hodnoty:
699 TypMater Zbozi;
700 Zbozi.Nazev = "Olej motorový Aral 10W-40";
701 Zbozi.Popis.Odolnost = -35;
```

Při přístupu ke složkám vnořeného záznamu musíme použít jméno tohoto záznamu a tečku. V některých případech je však takový strukturovaný způsob přístupu zbytečně náročný, můžeme v tom případě využít tzv. **anonymního záznamu**, kdy nepotřebujeme pojmenovat vnitřní záznam a jeho položky (přestože jsou v jeho působnosti) budou postaveny na roveň položkám ostatním. Uvedený příklad můžeme přepsat následujícím způsobem:

```
702
703 struct TypMater { //záznam o materiálu
```

```

704     string Nazev;      //název je u všech druhů materiálu
705     union {            //anonymní variantní záznam
706         float Delka;    //použije se u hřebíků
707         int Odolnost;    //použije se u oleje
708         float Hmotnost; //použije se u sádry
709     };
710 };
711 //příklad přiřazení hodnoty:
712 TypMater Zbozi;
713 Zbozi.Nazev = "Olej motorový Aral 10W-40";
714 Zbozi.Odolnost = -35; //zjednodušený přístup k variantním složkám

```

Pokud nějaké položky sdílejí v paměti stejný prostor, lze toho využít ještě k jiné věci: na jednu a tutéž hodnotu v paměti lze pohlížet různým způsobem podle toho, o jaký datový typ jde. Máme-li například čtyřbajtový prostor, můžeme s ním pracovat jako s proměnnou typu **long int**, ale také jako s proměnnou typu **float** nebo také **char[4]**.

Příklad: Na vstupu je celočíselná hodnota. Vypišme hodnoty jednotlivých bajtů této hodnoty, jak je zobrazena v paměti počítače.

```

715 union TypPrevod{
716     long int CeleCislo; //4 bajty, zpracovávané jako celek
717     char Bajty[4];      //také 4 bajty, ale dostupné po jednom
718 };
719 TypPrevod P;           //proměnná P bude sloužit pro převod
720 cin >> P.CeleCislo;    //přečteme vstupní hodnotu do položky CeleCislo
721 for (int i=0; i<4; i++) //a nyní budeme pracovat s jejími bajty
722     cout << int(P.Bajty[i]) << " "; //chceme číselný výpis
723 cout << endl;

```

12 Dynamické datové struktury

12.1 Statické, nebo dynamické?

Pojmem **statické**⁷ budeme pracovně označovat proměnné datových typů, se kterými jsme až dosud pracovali. Připomeňme si jejich vlastnosti:

- V okamžiku deklarace proměnné se v paměti vyhradí úsek, jehož velikost odpovídá datovému typu.
- Takto deklarované proměnné se uchovávají v úseku paměti pro ně vyhrazeném (v systémovém zásobníku).
- Proměnná existuje až do ukončení programového bloku, v němž byla deklarována.
- Velikost proměnné je pevně dána, závisí na datovém typu proměnné (u proměnných strukturovaných datových typů se odvíjí od složek, ze kterých se tento typ skládá).
- Proměnné jsou uloženy na adrese symbolicky vyjádřené jejich identifikátorem.

Některé z uvedených vlastností jsou nevýhodné – zejména nemožnost vytvořit proměnnou pouze v okamžiku, kdy ji potřebujeme, a opět uvolnit paměť, když už proměnná není potřeba, a dále nemožnost detailně ovlivňovat velikost proměnné. Z toho důvodu se používají pro ukládání dat proměnné obsluhované poněkud jiným způsobem. Tyto odlišné proměnné se nazývají **dynamické** a vyznačují se následujícími vlastnostmi:

- Proměnná vzniká až v okamžiku použití speciálního příkazu pro vytvoření proměnné, teprve tehdy je místo v paměti obsazeno.
- Proměnnou lze speciálním příkazem kdykoliv zrušit a paměť opět uvolnit.
- Velikost proměnné lze určit až v okamžiku vytvoření.
- Používá se jiná podstatně větší část paměti (tzv. hromada, halda, angl. heap) než u statických proměnných.

Dynamické proměnné se konstruují za použití datového typu ukazatel a datového typu podprogram. Nemají vlastní identifikátor a přistupuje se k nim přes adresy uložené v ukazatelích.

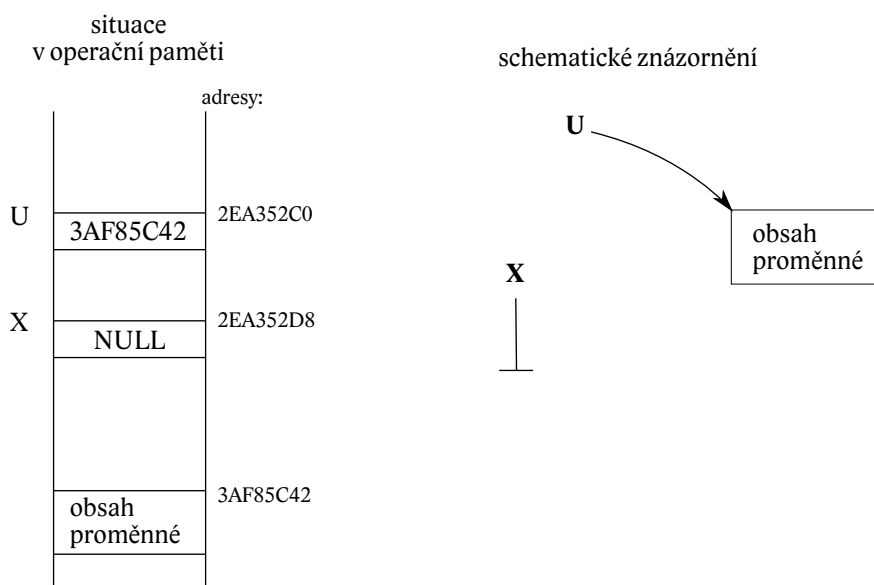
Kdy použít statickou a kdy dynamickou proměnnou? To záleží na tom, kterou vlastnost v daném případě považujeme za nejpodstatnější. Rozhodnutí je někdy klíčové pro efektivní práci celé aplikace.

⁷ Tento pojem je rovněž používán v objektově orientované terminologii ve zcela jiném významu.

12.2 Princip dynamické proměnné

Jak už víme, adresa statické proměnné je přímo dostupná pomocí jejího identifikátoru. Pokud například máme deklaraci `int A;`, uvedením identifikátoru `A` přistoupíme k obsahu proměnné. Pokud se použije dynamická proměnná, je potřeba někde uložit informaci o tom, na které adrese se nachází. Tato adresa je uložena v nějaké jiné proměnné. Každá proměnná obsahující adresu se nazývá **ukazatel**.

Pro lepší pochopení si můžeme ukazatel na dynamickou proměnnou přirovnat k odkazu na webovou stránku nebo ke směrovníku, šipce. Ukazatel pouze oznamuje, na které adrese jsou data. Samotná data jsou obsahem dynamické proměnné a nacházejí se v paměti na místě s danou adresou. Pro získání představy o těchto skutečnostech je velmi důležité umět celou situaci zakreslit. V nákresech se ukazatel znázorňuje šipkou, která vede směrem k dynamické proměnné – viz obr. 16.



Obrázek 16: Schematické znázornění vazby ukazatele na dynamickou proměnnou

V obrázku je na levé straně vidět skutečná situace v operační paměti, na pravé straně pak schematické vyjádření. Adresa v proměnné `U` odkazuje na dynamickou proměnnou ležící v jiném místě paměti. Adresa v proměnné `X` je speciální hodnota představující prázdný ukazatel, který neukazuje nikam (bývá implementována jako nula), schematicky se znázorňuje „nárazníkem“.

Je potřeba si uvědomit, že získání hodnoty dynamické proměnné vždy vyžaduje minimálně dvojí přístup do paměti: v prvním kroku se získá hodnota adresy, v druhém kroku se tato adresa využije pro přístup k hodnotě dynamické proměnné.

Velikost ukazatele je 4 B nebo 8 B, záleží na velikosti adresy v paměťovém modelu v dané počítačové architektuře. Téměř jakkoli velká dynamická proměnná nebo celá struktura provázaná navzájem ukazateli tak může být umístěna v hromadě, v systémovém zásobníku přitom zabírá pouze velikost adresy. Takto lze pracovat i s velkým množstvím dat a obejít omezení kladená na paměť. Nevýhodou tohoto řešení naopak může být určitá režie s ukazateli potřebnými ke svázání jednotlivých částí celé struktury.

12.3 Ukazatele v C++

V jazyce C++ je určení skutečnosti, že nějaká proměnná má obsahovat ukazatel (adresu), velmi jednoduché: při deklaraci použijeme znak `*` mezi názvem datového typu a identifikátorem proměnné. Například deklarace `int *A, B;` říká, že proměnná `A` bude obsahovat adresu, na níž se může nacházet hodnota typu `int`, zatímco proměnná `B` bude obyčejná celočíselná proměnná. Proměnnou `A` budeme dále nazývat **určitý ukazatel**, protože proměnná, na kterou ukazuje, *musí být* datového typu `int` a při jejím použití tuto skutečnost překladač kontroluje.

Speciálním případem je tzv. **obecný ukazatel** deklarovaný pomocí prázdného typu: `void *X;`. Předpokládá se, že při jakémkoliv přístupu k dynamické proměnné bude použito přetypování na potřebný datový typ, což umožňuje použít takový ukazatel podle potřeby pro jakákoliv data.

Jak víme, po deklaraci jakékoliv proměnné je její hodnota nedefinována. Totéž platí i pro ukazatele – po deklaraci nemůžeme ukazatel použít pro přístup k dynamické proměnné, ani se nemůžeme dotázat, zda je v ukazateli platná adresa či nikoliv. Adresu do ukazatele musíme nějak vložit – k tomu slouží tyto možnosti:

1. Vložení konstanty **NULL** – jde o jedinou možnou konstantu představující prázdný ukazatel. Bývá vnitřně implementována jako hodnota 0. Na tuto konstantu se můžeme dotázat – lze tedy jednoduše zjistit, zda obsahem ukazatele je prázdná adresa.
2. Vložení adresy, kterou už překladač zná – víme, že po deklaraci proměnné představuje její identifikátor adresu v paměti. Tuto adresu lze získat operátorem `&` a přiřadit ji do ukazatele. Například v uvedené deklaraci proměnných `A` a `B` lze přiřadit `A = &B;`. V případě určitého ukazatele se kontroluje datový typ proměnné, jejíž adresu přiřazujeme do daného ukazatele. Máme-li dva ukazatele stejného typu, můžeme je rovněž mezi sebou přiřazovat. Pomocí přetypování můžeme přiřazovat i různé ukazatele, čehož se využívá zejména v případě obecného ukazatele.
3. Vložení adresy, která vznikne při vytvoření dynamické proměnné a při alokaci potřebné paměti – to je ten hlavní způsob, kvůli kterému dynamické proměnné využíváme. Alokaci paměti a její adresu zajistí operátor **new**⁸, jehož operandem je datový typ dynamické proměnné. Platí-li uvedená deklarace, pak můžeme psát `A = new int;`. V paměti se vytvoří prostor o velikosti proměnné typu `int` a jeho adresa se uloží do proměnné `A`. V případě alokace pole se ještě za datový typ dynamické proměnné uvede v hranatých závorkách požadovaný počet složek, např. `A = new int[55];` (o poli jako o zvláštní struktuře se ještě zmíníme).

Máme-li v proměnné typu ukazatel platnou adresu, potřebujeme přistoupit k hodnotě dynamické proměnné. Tuto tzv. **dereferenci** provedeme operátorem hvězdička zapsaným *před* proměnnou typu ukazatel, například `*A`. Platí-li uvedená deklarace, pak můžeme například psát `*A = 17;` nebo `*A++;` či `cout << *A - 10;`.

⁸ V jazyce C podobnou službu zajišťuje funkce `malloc`, ale tímto mechanismem se zde nebudeme zabývat. Systém alokace paměti pomocí `new` a původní systém v jazyce C nelze používat současně.

Musíme respektovat skutečnost, že hvězdička představující dereferenci je operátor s určitou prioritou a v případě kombinace s jinými operátory ve výrazu je občas potřeba prioritu upravit závorkami. Hvězdička jako operátor násobení má například nižší prioritu, takže výraz `*A * *A` bude vyhodnocen očekávaným způsobem (napřed dereference a pak násobení), zatímco operátor tečka (přístup ke složce záznamu) má prioritu vyšší než dereference, máme-li tedy například dynamickou proměnnou `Z` typu záznam a chceme zapsat přístup k její složce `cislo`, je nutné psát `(*Z).cislo`. Zápis `*Z.cislo` by znamenal, že složka `cislo` je ukazatel, jehož adresu chceme použít. Protože velmi často máme dynamickou proměnnou typu struktura (nebo třída – **class**), bylo by poměrně těžkopádné při každém přístupu ke složce dynamické proměnné měnit závorkami prioritu dereferenčního operátoru. Proto existuje syntaktická náhrada: místo `(*Z).cislo` můžeme použít operátor `Z->cislo`.

Uvolnění paměti zabírané dynamickou proměnnou provedeme operátorem **delete**⁹, jehož operandem je ukazatel na tuto proměnnou. Například uvolnění paměti alokované pro dynamickou proměnnou, na kterou ukazuje ukazatel `A`, provedeme příkazem **delete A**; . Uvolňujeme-li pole, přidáváme k operátoru **delete** ještě prázdné hranaté závorky. Inteligence tohoto operátoru spočívá v tom, že ví, kolik paměti (kolik složek pole) má uvolnit.

V následujícím příkladu shrneme všechny uvedené prvky:

```

724 int *UkInt, cislo;
725 UkInt = &cislo; //adresa statické proměnné se uloží do ukazatele
726 *UkInt = 25; //přístup k proměnné, jejíž adresa je v UkInt
727 cout << cislo << endl; //zobrazí se 25 -> princip práce s odkazem
728 UkInt = new int; //původní adresa se nahradí novou, alokuje se paměť
729 *UkInt = 38;
730 *UkInt *= 2;
731 cout << *UkInt << endl; //zobrazí se 76
732 delete UkInt; //paměť se uvolní
733
734 struct TypOsoba {
735     string Jmeno;
736     int Plat;
737 };
738 TypOsoba *Lojza, *Franta; //ukazatele na strukturu
739 Lojza = new TypOsoba; //v paměti se vytvoří dynamická struktura
740 (*Lojza).Jmeno = "Alois"; //dvě alternativy přístupu ke složkám
741 Lojza->Plat = 25000;
742 Franta = new TypOsoba;
743 *Franta = *Lojza; //přiřazení záznamů -- nyní máme dvě kopie dat
744 Lojza->Plat = 30000;
745 cout << Franta->Plat << endl; //zobrazí se 25000
746 Franta = Lojza; //přiřazení adres -- máme jednu adresu (Lojzovu)

```

⁹ Podobně jako u alokace paměti i zde existuje funkce staršího systému C s názvem **free**, jejímž parametrem je ukazatel na uvolňovanou proměnnou. A stejně jako u alokace i zde nemůžeme navzájem mluvit **delete** a **free**.

```
747 //ukazatel na Frantův záznam se přepíše, už s ním nemůžeme pracovat
748 Lojza->Plat = 35000;
749 cout << Franta->Plat << endl; //zobrazí se 35000
750 delete Lojza; //uvolnění paměti
751 cout << Franta->Plat << endl; //POZOR! Chyba -- práce s neplatnou adresou
```

12.4 Pole v C++

V části 10.4 na s. 96 už bylo zmíněno, že pole je poněkud zvláštní struktura, protože je (aniž to uživatel může jakkoliv ovlivnit) *vždy dynamická*. Deklarací pole v podstatě deklarujeme ukazatel na první složku pole a alokujeme potřebnou paměť v hromadě. Je to jednoduchá obrana proti alokaci velkého paměťového bloku v systémovém zásobníku. Už bylo na různých místech rovněž zmíněno, že z toho vyplývají různé odlišnosti – pole nelze navzájem přiřadit, pole nelze předat do podprogramu hodnotou, velikost pole lze určit při deklaraci výrazem vyčísleným až v době běhu programu (nikoliv konstantou) atd.

Ukážeme si nyní práci s polem dvojí cestou – již známou metodou „statické“ struktury a metodou ukazatelovou.

```
752 long int *pole; //ukazatel na proměnnou typu long int
753 unsigned int pocet;
754
755 pole = new long int[1000]; //pole má velikost 1000 krát velikost long int
756 delete [] pole; //zrušení pole a uvolnění paměti
```

Nyní je i jasnější, proč počet obsazených prvků pole je tak důležitou hodnotou, kterou nutně potřebujeme téměř při jakémkoliv zpracování. Jinými slovy, musíme mít v každém okamžiku jasno v tom, které indexy jsou platné. To je nezbytné při předávání pole do podprogramu, protože uvnitř podprogramu už není možné nic takového zjistit.

12.4.1 Ukazatelová aritmetika

Ukazatel je implementován jako celočíselná hodnota. Logicky se nabízí tuto celočíselnou hodnotu zpracovávat jako skutečné celé číslo – přičítat nebo odečítat hodnotu, případně odečítat mezi sebou dvě adresy. Těmto možnostem se říká **ukazatelová aritmetika**.

Jako programátoři nebo uživatelé vyššího programovacího jazyka bychom však měli být dostatečně izolováni od přímých implementačních detailů a také chráněni před vyloženě nesprávnými a nebezpečnými operacemi. Proto je představa, že volně zacházíme s adresami jako s obyčejnými celými čísly, poněkud přísněji vymezena. Operace, které zapisujeme stejně jako u obyčejných celých čísel, mají významně jiné chování podpořené informacemi, které překladač z kontextu již zná. Ukazatelová aritmetika má smysl, pokud je jednoznačně definováno uspořádání příslušných objektů v paměti.

Prakticky je to zaručeně splněno, pokud jde o ukazatele na prvky téhož pole. S ukazateli lze provádět dvě následující operace (**U** je ukazatel, **C** je celočíselná hodnota):

Operace	Význam
U+C , U-C	posuv v poli o C prvků
U2-U1	zjištění počtu prvků pole mezi adresami dvou prvků

Z toho vyplývá, že máme-li například pole `double *p;`, pak `p` představuje adresu prvního prvku a `p+1` neukazuje o 1 bajt dále, ale o jednu složku dále. Adresa se změní nikoliv o jedničku, ale o `sizeof(double)`. Jde v podstatě o jiný zápis indexace, neboť `p+C` je totéž jako `&p[C]`.

Ukažme si na příkladu, jak lze pracovat s polem alternativně pomocí ukazatelové aritmetiky místo „obyčejné“ indexace:

```

757 long int *p = new long int[10]; //pole 10 složek long int
758 //ekvivalentní zápis deklarace: long int p[10];
759 for (int i=0; i<10; i++) *(p+i)=2*i; //naplnění složek
760 for (int i=0; i<10; i++) cout<<*(p+i)<<' '; //výpis
761 long int *z = &p[9]; //ukazatel z dostává adresu posledního prvku
762 cout<<"Počet prvků pole: " << z-p << endl; //rozdíl dvou adres

```

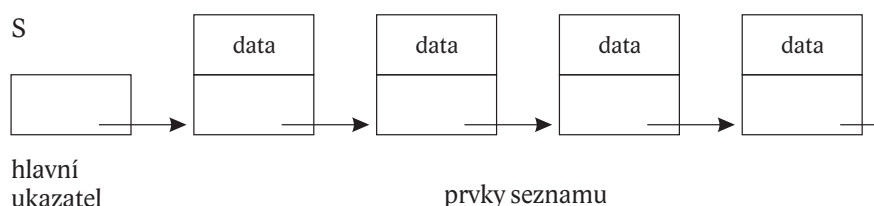
12.5 Dynamické datové struktury

Použití dynamických proměnných odvozených od jednoduchých datových typů je sice možné, ale málo efektivní. Proměnné strukturovaných datových typů mají využití například při očekávaném velkém množství dat. Zásadní výhoda využití dynamických proměnných ale spočívá v možnosti jejich spojování do rozsáhlých dynamických struktur. Struktura se může skládat z prvků stejného i různého typu. Princip je jednoduchý – prvek struktury obsahuje kromě dat také (alespoň jeden) ukazatel, který ukazuje na další prvek. Potřebujeme tedy kombinovat různé datové typy pro data a k nim navíc ukazatele, proto se dynamické datové struktury často realizují s použitím záznamů (**struct**).

12.5.1 Lineární seznamy

Lineární jednosměrný seznam

Nejjednodušším případem dynamické struktury je **lineární jednosměrný seznam**. Prvkem tohoto seznamu je záznam s nejméně dvěma složkami: datovou a ukazatelovou. Datová složka může být složena z více dalších složek, může to také být ukazatel na data alokovaná v jiné části paměti. Druhá složka, ukazatelová, pak nese adresu následníka daného prvku. Poslední prvek seznamu má v této složce prázdný ukazatel **NULL**. První prvek seznamu je dostupný jedním ukazatelem. Situaci znázorňuje obr. 17.

Obrázek 17: *Lineární jednosměrný dynamický seznam*

Typické operace s lineárním seznamem

Na jednoduchém příkladu seznamu, v jehož prvcích jsou celočíselné hodnoty, ukážeme základní způsoby práce se seznamem. Mějme následující definice:

```

763 struct Prvek { //jeden prvek seznamu
764     int data;    //datová složka (např. celé číslo)
765     Prvek *dalsi; //vazba na další prvek stejného typu
766 };
767 typedef Prvek *UkPrvek; //datový typ ukazatele na prvek seznamu
768 UkPrvek Start; //Start je ukazatel na první prvek (hlavní ukazatel)

```

Proměnná `Start` ukazující na první prvek reprezentuje celý seznam. Její adresu nesmíme za žádných okolností ztratit – přišli bychom o celý seznam. Při různých operacích proto dost často použijeme ještě další pomocné ukazatele.

Před započítím práce je nutné seznam inicializovat, tj. získat nový prázdný seznam – seznam s nulovým počtem prvků. Toho dosáhneme jednoduše – hlavní ukazatel bude obsahovat prázdnou hodnotu `NULL`. Použijeme k tomu následující proceduru, i když obsahuje pouze jediný příkaz, budeme ale podobný systém používat i v případě, že seznam bude poněkud složitější.

```

769 void Init(UkPrvek &S){
770     S = NULL;
771 }

```

Zjištění prázdnosti – obsahuje-li hlavní ukazatel hodnotu `NULL`, je seznam prázdný. Ke zjištění tohoto stavu lze použít funkci:

```

772 bool JePrazdny(UkPrvek S){
773     return S==NULL;
774 }

```

Vložení prvku do seznamu – rozšíření seznamu o nový prvek s potřebnými daty. Operace se liší podle toho, kam nový prvek vkládáme: na začátek, na konec nebo do libovolného jiného místa seznamu. Nejjednodušší je vložení prvku na začátek seznamu, použijeme k tomu následující proceduru:

```

775 void VlozStart(UkPrvek &S, int D){ //seznam a vkládaná data
776     UkPrvek Pom = new Prvek; //vytvoření nového prázdného prvku
777     Pom->data = D; //do datové složky vložíme data z parametru
778     Pom->dalsi = S; //nový prvek bude první, navážeme na současný start
779     S = Pom; //nový prvek se stává prvním v seznamu
780 }

```

Procedura funguje správně jak při vkládání do prázdného, tak i do neprázdného seznamu. Je-li seznam prázdný, **S** má hodnotu **NULL**, pak na řádku 778 vkládáme do ukazatele na další prvek tuto hodnotu **NULL**, prvek je v tom případě posledním prvkem seznamu. Pokud seznam prázdný není, nový prvek na stejném řádku dostává ukazatel na současný první prvek, vzniká vazba nového prvku na celý zbytek současného seznamu, čímž se nový prvek stává jeho součástí. Následně se do hlavního ukazatele vloží ukazatel na nový prvek – nový start seznamu.

Vložení prvku na konec seznamu se provádí poněkud odlišně. Pro tento účel je velmi vhodné mít vždy ukazatel na současný poslední prvek, za nějž nový prvek budeme připojovat. Předpokládejme poněkud jinou definici celého seznamu a jeho inicializaci:

```

781 struct TypSeznam {
782     UkPrvek Start, Konec;
783 };
784 void Init(TypSeznam &S){
785     S.Start = NULL; //reprezentace prázdného seznamu
786     S.Konec = NULL;
787 }

```

Procedura pro přidání nového prvku na konec seznamu pracuje odlišně, je-li seznam prázdný, nebo je-li seznam neprázdný:

```

788 void VlozKonec(TypSeznam &S, int D){ //seznam a vkládaná data
789     if (S.Start==NULL) { //seznam je prázdný, tvoříme první prvek
790         S.Start = new Prvek; //ukazatel na nový prvek jde do hlavního
791         S.Konec = S.Start; //i do koncového ukazatele
792     } else { //seznam je neprázdný
793         S.Konec->dalsi = new Prvek; //nový prvek je navázán na posledního
794         S.Konec=S.Konec->dalsi; //aktualizujeme ukazatel na konec
795     }
796     S.Konec->data = D; //do datové složky vložíme data z parametru
797     S.Konec->dalsi = NULL; //nový prvek bude poslední
798 }

```

Nejkomplikovanější je vložení nového prvku někam doprostřed již existujícího seznamu. Obvykle totiž musíme nějak najít odpovídající místo, kam nový prvek chceme, rozlišit všechny případy, které mohou nastat (prázdný/neprázdný seznam, nový prvek jde na začátek, na konec nebo někam do prostřed seznamu) a pak teprve můžeme aplikovat odpovídající postup.

Předpokládejme, že jsme tyto přípravné akce již provedli a následující procedura dostane jako první parametr ukazatel na prvek, za nějž se má vkládat. Dále předpokládejme, že se skutečně jedná už jen o situaci, kdy seznam je neprázdný a nový prvek není ani první, ani poslední (nenastává ani jeden z dříve uvedených případů):

```

799 void VlozStred(UkPrvek Kam, int D){ //ukazatel na prvek a vkládaná data
800     UkPrvek Pom = new Prvek; //vytvoření nového prázdného prvku
801     Pom->data = D; //do datové složky vložíme data z parametru
802     Pom->dalsi = Kam->dalsi; //nový prvek bude ukazovat na následníka prvku Kam
803     Kam->dalsi = Pom; //nový prvek se stává prvním následníkem prvku Kam
804 }

```

Průchod seznamem – často je třeba projít seznam prvek po prvku od začátku (lineární jednosměrný seznam ostatně ani jiný přístup neumožňuje). Začínáme prvkem, na nějž ukazuje hlavní ukazatel; průchod končí u posledního prvku (jeho ukazatel na další prvek má hodnotu **NULL**).

Při průchodu seznamem se vždy používá pomocný ukazatel, hlavním ukazatelem zásadně nemaniplujeme, abychom seznam neztratili. Pomocný ukazatel postupně dostává adresy dalších a dalších prvků, dokud nenabude hodnoty **NULL**:

```

805 void Projdi(UkPrvek S){ //hlavní ukazatel na seznam
806     UkPrvek Pom = S; //pomocný ukazatel na začátek
807     while (Pom!=NULL) {
808         cout << Pom->data << endl; //zpracování daného prvku
809         Pom = Pom->dalsi; //přesun na další prvek
810     }
811 }

```

Velmi podobně vypadá algoritmus sekvenčního hledání v lineárním seznamu. Jde o průchod, který však může být ukončen dříve při nalezení hodnoty. Funkce může jako výsledek vrátit logickou hodnotu (nalezen/nenalezen) nebo ukazatel na prvek s nalezenými daty (nebo **NULL** při neúspěšném hledání):

```

812 bool Najdi(UkPrvek S, int Co){ //hlavní ukazatel na seznam a hledaná hodnota
813     UkPrvek Pom = S; //pomocný ukazatel na začátek
814     while (Pom!=NULL and Pom->data!=Co)
815         Pom = Pom->dalsi; //přesun na další prvek
816     return Pom!=NULL; //výsledek logickou hodnotou
817 //pokud chceme vrátit ukazatel, zapíšeme: return Pom; a typ funkce je UkPrvek
818 }

```

Odstranění prvku – podobně jako u přidávání je potřebné rozlišit několik případů: odstranění prvního prvku, posledního prvku, jiného prvku, prázdný seznam, neprázdný seznam. Vždy při odstranění prvku je nezbytné správně navázat ukazatele, aby se neporušila struktura seznamu, zpracovat data z odstraňovaného prvku (např. vypsát na výstup, vložit do proměnné nebo do parametru), a odstra-

nit prvek z paměti (**delete**). Při manipulaci s prvním nebo posledním prvkem seznamu aktualizovat hlavní ukazatel a ukazatel na konec. Ukažme případ odstranění prvku někde uprostřed seznamu – jako vstup máme neprázdný ukazatel na prvek, za nímž má být odstraňovaný prvek. Výsledkem funkce je datová složka odstraňovaného prvku:

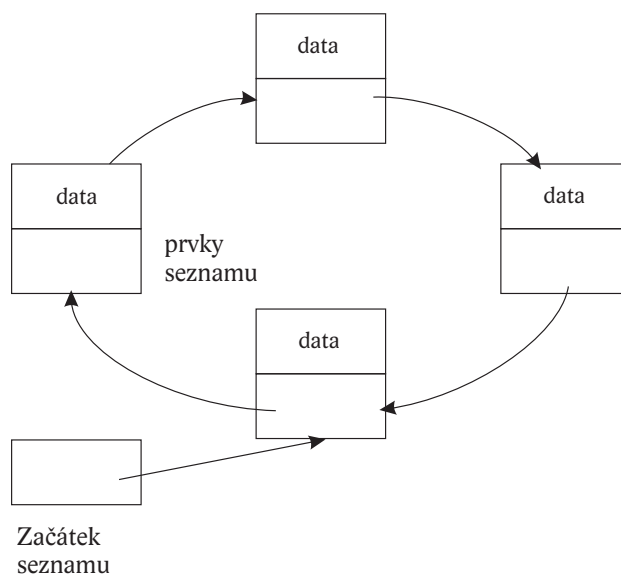
```

819 int OdstranStred(TypSeznam &S; UkPrvek Kde){ //seznam a místo odstranění
820     UkPrvek Pom = Kde->dalsi; //pomocný ukazatel na rušený prvek
821     Kde->dalsi = Pom->dalsi; //obejití rušeného prvku
822     int D = Pom->data; //úschova dat rušeného prvku
823     delete Pom; //odstranění rušeného prvku z paměti
824     return D; //výstup hodnoty rušeného prvku
825 }

```

Lineární kruhový seznam

Princip spočívá v tom, že ukazatel na následníka „posledního“ prvku ukazuje na první prvek (viz obr. 18).



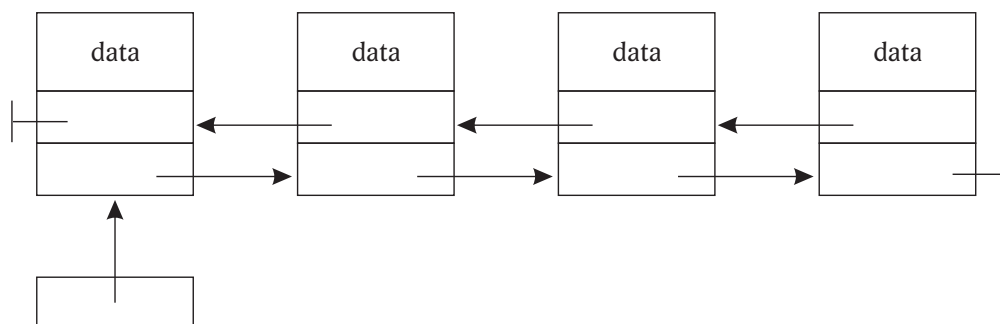
Obrázek 18: *Lineární kruhový seznam*

Odlišnost práce spočívá zejména v tom, že v seznamu není ukazatel s hodnotou **NULL**, který by označoval konec seznamu. Udržíme proto jeden (hlavní) ukazatel na seznam, se kterým se nehýbe. Tento ukazatel pak slouží i pro orientaci (například při hledání můžeme testovat, zda jsme už „obešli“ celý kruh).

Při všech vkládacích a odebíracích operacích musíme zajistit, aby kruh zůstal zachován.

Lineární obousměrný seznam

Dalším speciálním případem seznamu s mírně odlišným způsobem implementace je obousměrný seznam, v němž má každý prvek seznamu dva ukazatele a ukazuje jak na svého následníka, tak na svého předchůdce (ovšem kromě prvního a posledního prvku, ty v krajním ukazateli obsahují hodnotu **NULL**) – viz obr. 19.



Obrázek 19: Lineární obousměrný seznam

Seznam s aktivním prvkem

V jednoduchých případech je seznam využíván jako zásobník nebo fronta, takže se typicky manipuluje pouze s prvky na začátku nebo na konci. V obecnějším případě můžeme manipulovat s libovolnými prvky kdekoliv jinde, proto je vhodné udržovat ještě třetí ukazatel (kromě ukazatele na začátek a na konec). Tímto ukazatelem zpřístupňujeme obvykle prvek seznamu, s nímž hodláme provádět nějakou akci, nebo s nímž byla prováděna předchozí operace. Takový prvek seznamu pracovním nazýváme *aktivní prvek*, pokud takový prvek existuje, seznam pak označujeme jako *aktivní seznam*. Aktivitu lze přesouvat na následující prvek (v případě obousměrného seznamu i na předchozí prvek), nastavovat na začátek (konec), zcela zrušit, testovat. Operace vkládání, mazání nebo zobrazování obsahu se vážou na aktivní prvek, v neaktivním seznamu neprovádějí nic.

12.5.2 Stromy

Pojmem **strom** označujeme v teorii grafů libovolný acyklický graf. Pro aplikace v oblasti programování a algoritmizace se hodí některé případy stromů, na něž se v následujícím textu podíváme.

Stromy jsou obvykle definovány jako *rekurzivní struktury* – určitou vlastnost, která platí v celém stromu, snadno znázorníme na úseku složeném z otcovského uzlu a jeho bezprostředních následníků. Logicky se pak na rekurzivně definované struktury uplatňují rekurzivně zapsané operace.

Pro implementaci stromových struktur se přímo nabízejí dynamické struktury. Uzlem stromu je záznam s datovou složkou (budeme předpokládat existenci typu `TypData`) a ukazatelovou částí, v níž počet ukazatelů rozhoduje o možném počtu následníků uzlu.

Obecné stromy

Jedná se o stromy, jejichž každý uzel může mít libovolný (i nulový) počet následníků. Obecným stromem můžeme například znázornit obsah disku – jednotlivé adresáře tvoří uzly stromu, z nichž vycházejí jako následníci uložené soubory (listy stromu) nebo další adresáře.

Implementace obecného stromu je založena na vhodné interpretaci libovolného počtu následníků daného uzlu. To může být řešeno jako lineární seznam ukazatelů na uzly stromu, například:

```

826 struct TypUzel; //odložená definice záznamu o uzlu stromu
827 typedef TypUzel *UkUzel; //ukazatel na uzel
828
829 struct TSeznamDalsich { //seznam ukazatelů na uzly
830     UkUzel Uzel;
831     TSeznamDalsich *DalsiUzel;
832 };
833
834 typedef TSeznamDalsich *UkSeznamDalsich;
835
836 struct TypUzel { //doplněná definice - reprezentace uzlu stromu
837     TypData Data;
838     UkSeznamDalsich Naslednici;
839 };

```

Všimněte si, že v záznamu `TypUzel` potřebujeme ukazatel na seznam následníků a v seznamu následníků potřebujeme ukazatel na `TypUzel`. Ať zvolíme jakékoliv pořadí definic, vždy se dostaneme do situace, že potřebujeme identifikátor typu, který ještě nebyl definován. Abychom překladači mohli sdělit ve správném okamžiku, o jaký typ jde, provedeme odloženou definici záznamu o uzlu na řádce 826. Pak můžeme takto vytvořený identifikátor `TypUzel` nadále využívat jako známý typ, i když ještě nebylo jeho tělo uvedeno. To vyřešíme později (od řádku 836).

Vytváření obecného stromu je různé podle toho, co ve stromu je a jak získáme informaci o pozici uzlu, který do stromu chceme vkládat. Pro ilustraci práce s obecným stromem ukážeme průchod stromem a výpis všech jeho datových složek.

Předpokládejme, že máme strom, v němž je v každém uzlu jméno člověka a následnické uzly obsahují jména všech jeho dětí. Chceme vypsát jména všech lidí celého rodu. Navážeme na předchozí definice:

```

840 typedef string TypData; //datovou složkou uzlů budou řetězce
841 void VypisRod(UkUzel U){
842     if (U!=NULL) { //jde o platný uzel
843         cout << U->Data << endl; //výpis osoby
844         UkSeznamDalsich Pom = U->Naslednici;
845         while (Pom!=NULL) {
846             VypisRod(Pom->Uzel); //rekurzivní zpracování podstromu
847             Pom = Pom->DalsiUzel; //přechod na dalšího následníka

```

```
848     }  
849   }  
850 }  
851  
852 int main(){  
853     UkUzel Rod;      //ukazatel na kořen stromu -- zakladatele rodu  
854     ... //strom je naplněn daty  
855     VypisRod(Rod);  //výpis celého rodu  
856     ...  
857 }
```

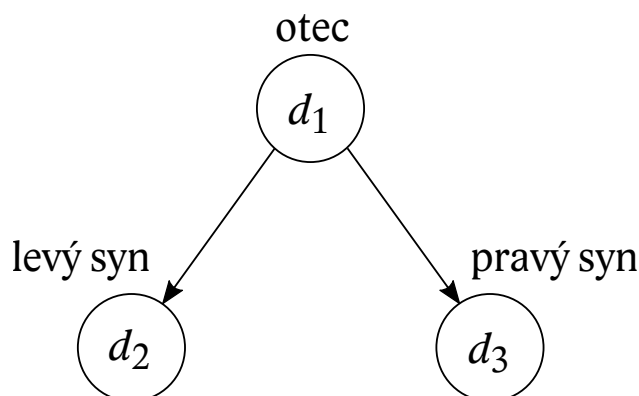
Pravidelné stromy

Pravidelnými stromy rozumíme takové stromy, jejichž počet následníků je omezen na určitý počet. Nejčastěji se můžeme setkat se stromy binárními (počet následníků je nejvýše dva), existují i stromy ternární (s nejvýše třemi následníky) atd.

Pravidelné stromy mohou být *uspořádané*, to znamená, že mezi datovými složkami uzlů existují předem určené relační vazby.

Uspořádaný binární strom – binární vyhledávací strom

Jednou z nejčastějších variant pravidelných uspořádaných stromů je **binární vyhledávací strom** (BVS), jehož element je znázorněn na obr. 20.



Obrázek 20: Trojice uzlů (*element*) binárního vyhledávacího stromu

V elementu BVS platí uspořádání v datech, nejčastěji $d_2 \leq d_1 < d_3$, tj. levý syn (a s ním i celý podstrom) obsahuje hodnoty menší než otec, pravý syn pak hodnoty větší než otec. Protože musíme pamatovat i na stejně velká data, do levého (nebo někdy do pravého) podstromu vkládáme i stejně velké hodnoty.

Implementaci pomocí dynamické struktury provedeme podobně jako u obecného stromu, ovšem vzhledem ke skutečnosti, že následníci jsou nejvýše dva, nemusíme je řešit seznamem, ale stačí dvě složky:

```

858 struct TypUzel { //uzel binárního stromu
859     TypData Data;
860     TypUzel *Vlevo, *Vpravo;
861 };
862 typedef TypUzel *UkUzel; //ukazatel na uzel

```

Celý strom lze reprezentovat jediným ukazatelem na kořen. Typické operace nad binárním uspořádaným stromem jsou: inicializace, vložení nového uzlu, vyhledání uzlu s určitou hodnotou, výpis celého stromu. Opět jako u obecného stromu využijeme skutečnosti, že se jedná o rekurzivně definovanou strukturu, některé operace s výhodou zapíšeme v rekurzivní podobě. Využijeme předchozích definic a zapíšeme možné tvary jednotlivých operací:

```

863 void Init(UkUzel &S){ //inicializace
864     S = NULL;
865 }
866 void BInsert(UkUzel &S, TypData D){ //vložení údaje D do stromu
867     if (S==NULL){ //prázdný strom nebo místo pro připojení listu
868         S = new TypUzel; //vytvoření nového uzlu + vazba na existující uzel
869         S->Data = D; //nový uzel je listem
870         S->Vlevo = NULL;
871         S->Vpravo = NULL;
872     } else
873         if (D < S->Data) //vkládaná hodnota je menší, jdeme vlevo
874             BInsert(S->Vlevo, D);
875         else BInsert(S->Vpravo, D); //jinak rekurzivně vpravo
876 }
877 bool BSearch(UkUzel S, TypData D){ //zjištění přítomnosti D ve stromu
878     UkUzel Pom = S; //pomocný ukazatel pro vyhledávání, začínáme v kořeni
879     while (Pom!=NULL and D!=Pom->Data) //je kde hledat a nebylo nalezeno
880         if (D<Pom->Data) Pom=Pom->Vlevo;
881         else Pom=Pom->Vpravo; //jdeme do nižší hladiny
882     return Pom!=NULL; //výsledek hledání
883 }
884 void Inorder(UkUzel S){
885     if (S!=NULL){ //existující uzel -> zpracujeme
886         Inorder(S->Vlevo); //napřed levý podstrom
887         cout << S->Data; //pak kořen
888         Inorder(S->Vpravo); //nakonec pravý podstrom
889     }
890 }

```

13 Soubory

Pojem **soubor** představuje posloupnost údajů typicky uloženou na nějakém záznamovém médiu (disku). Každý programovací jazyk nezbytně potřebuje nějakou výbavu pro práci se soubory, protože bez toho by nebylo možné efektivně zpracovávat data (veškeré vstupy a výstupy dat by se odehrávaly pouze editací z klávesnice a sledováním výsledků na zobrazovacím zařízení). Soubory jsou spojeny s jejich fyzickým uložením mimo operační paměť, proto je k jejich ovládnutí nutná asistence operačního systému – v každé architektuře (případně i verzi) operačního systému se fyzicky soubory mohou obsluhovat jinak, přičemž technické detaily uložení a manipulace se soubory nás jako programátory nemusí vůbec zajímat. Potřebujeme vytvářet programy, které jsou co nejvíce přenositelné mezi různými platformami.

Z toho důvodu je veškerá fyzická manipulace se soubory ukryta do podprogramů, které jsou sice různě implementovány, ale používají se stejně.

Operační systém rozpoznává dva hlavní typy souborů – **textové** a **netextové** (binární). Hlavní výhodou a hlavním smyslem textových souborů je přímá čitelnost člověkem – veškeré informace jsou ukládány ve formě posloupnosti znaků. Z toho samozřejmě vyplývá, že (téměř) veškeré informace se při čtení takového souboru konvertují na binární podobu, aby je bylo možné zpracovat (například sečítat čísla) a naopak při výpisu se obsah operační paměti zase konvertuje na posloupnost znaků vyjadřující člověkem čitelnou zobrazenou hodnotu.

Naproti tomu netextové soubory mají základní výhodu v efektivitě komunikace – přenosy mezi operační pamětí a diskem probíhají bez jakýchkoliv konverzí a mnohdy i bez jakékoliv kontroly datových typů.

13.1 Standardní soubory

Práci se soubory už do jisté míry známe – už od prvních programů jsme využívali tzv. **standardních** souborů (standardní vstup, standardní výstup, standardní chybový výstup). Standardní soubory jsou logicky textové, neboť vstup může být tvořen posloupností znaků zadávanou z klávesnice, standardní výstupy pak tvoří informaci, kterou uživatel čte na monitoru.

Pro práci se standardními soubory využíváme knihovnu `iostream`, v níž jsou k dispozici objekty `cin`, `cout` a `cerr` (resp. `clog`), veškeré manipulace probíhají podle potřeby s konverzemi (proudové operátory) i bez konverzí (např. čtení po jednotlivých bajtech, po jednotlivých řádcích). Řadu z těchto manipulací můžeme využít i pro nestandardní, tj. **uživatelské** soubory.

Hlavní výhodou standardních souborů je jejich poloautomatická obsluha: automaticky se otevírají, kontrolují a uzavírají. Specifika detailní obsluhy jsou skryta, proto se velmi dobře hodí pro většinu aplikací, které jsme doposud řešili. Navíc má operační systém i další praktické vymoženosti, které umožňují významně rozšířit aplikaci standardních souborů: přesměrování a kolonu v příkazovém řádku.

13.2 Uživatelské soubory

V některých případech však potřebujeme pracovat se soubory, které jsou jiné než standardní: například máme více různých vstupních proudů dat, potřebujeme zpracovávat binární data atd. V těchto situacích je nutné převzít některé operace a soubory obhospodařovat ve vlastní režii. Opět jako u standardních souborů je k dispozici knihovna, která se tentokrát nazývá `fstream` a implementuje veškerou potřebnou podporu.

13.2.1 Základní manipulace se soubory

Při práci s uživatelskými soubory obecně musíme:

1. *Deklarovat proměnnou typu soubor.*

V knihovně `fstream` máme k dispozici tyto datové typy:

- `ifstream` – proměnná představující soubor určený ke čtení (předpona `i` je zkratkou slova input).
- `ofstream` – proměnná představující soubor určený k zápisu (předpona `o` je zkratkou slova output).
- `fstream` – proměnná představující soubor pro čtení i pro zápis.

Příklady:

```
891 | ifstream Vstupy; //soubor určený pro čtení
892 | ifstream Pomocny; //soubor k obecné manipulaci
```

2. *Propojit vnitřní proměnnou s fyzickým souborem na disku.*

Před započítím jakékoliv manipulace musíme sdělit operačnímu systému, se kterým skutečným souborem máme v úmyslu pracovat. Fyzické jméno souboru je dáno konvencemi příslušného operačního systému (například `e:\data\projekty\ucto.dat`), někde záleží na velkých a malých písmenech, někde nikoliv atd.

Propojení proměnné a fyzického jména souboru můžeme provést při deklaraci, nebo po deklaraci metodou `open`, např.

```
893 | ifstream Vstupy;
894 | Vstupy.open("../data/zdroje.txt");
895 | ofstream Pomocny("/tmp/aux.txt");
```

3. *Otevřít soubor pro zamýšlenou operaci.*

Otevřením souboru se provede počáteční zjištění, zda je soubor k dispozici a zda je fyzicky schopen zamýšlené operace (například soubor nemusí mít oprávnění k zápisu). Pokud použijeme k deklaraci typ signalizující režim práce (`ifstream` nebo `ofstream`), už nemusíme specifikovat nic dalšího, soubor se otevře pro čtení (zápis) a předpokládá se, že je textový.

Pokud použijeme typ `fstream` nebo chceme nastavit některý specifický režim práce, uvedeme při otevření ještě druhý parametr, který může být složen z různých dalších specifikací, např.

```
896 | ifstream Vstupy; //soubor je určen pro čtení
897 | Vstupy.open("../data/zdroje.txt", ios::binary); //soubor je binární
898 | fstream Pomocny("/tmp/aux.txt", ios::out | ios::binary);
899 |         //soubor je otevřen pro zápis a současně je binární
```

Režimy práce mohou být:

- `app`, nastaví pozici zápisu na konec;
- `ate`, nastaví ukazatel do souboru na konec;
- `binary`, soubor je binární;
- `in`, soubor je otevřen pro čtení;
- `out`, soubor je otevřen pro zápis;
- `trunc`, obsah souboru je smazán a bude přepsán novými daty.

4. Zkontrolovat použitelnost souboru.

K tomu můžeme využít logické metody `Vstupy.is_open()` – hodnota `true` signalizuje, že je všechno v pořádku. V opačném případě nastal problém, soubor nelze zpracovávat.

5. Provést potřebné čtení nebo zápis dat.

Operace čtení a zápisu se liší podle typu souboru (textový/netextový), viz dále.

6. Uzavřít soubor.

Uzavření souboru je operace nezbytná při jakékoliv aktualizaci, protože teprve uzavřením souboru se všechna data dočasně ukládaná ve vyrovnávací operační paměti fyzicky přenesou na disk. Není-li soubor korektně uzavřen, může se stát, že nebude po skončení programu obsahovat potřebná data, nebo dokonce se může jevit jako poškozený.

Uzavření se provede metodou `close()`, např. `Pomocny.close()`.

13.2.2 Zpracování textových souborů

U textových souborů je manipulace stejná jako u standardních souborů, provedeme tedy jen stručnou rekapitulaci:

- Předpokládejme deklarace:

```
900 | ifstream Vstupy("isoubor"); ofstream Vystupy("osoubor");
901 | char c, p[N]; string s;
```

a předpokládejme, že soubory jsou zpracovatelné.

- Čtení s konverzí: `Vstupy >> proměnná`

- Čtení znak po znaku: `Vstupy.get(c);`
- Čtení řádku: `Vstupy.getline(p, M);` nebo `Vstupy.getline(p, M, 0);`, kde `M` je požadovaný počet čtených znaků a `0` je požadovaný oddělovač
- Čtení řádku: `getline(Vstupy, s);` nebo `getline(Vstupy, s, 0);`
- Zápis s konverzí: `Vystupy << výraz;`
- Zápis znak po znaku: `Vystupy.put(c);`
- Zápis konce řádku: `Vystupy << endl;`

Příklad: V jistém konfiguračním souboru s názvem „datarc“ je na každém řádku dvojice údajů: název parametru a jeho hodnota. Dvojice je oddělena rovnítkem. Přečtete tento soubor a zjistíte, jakou hodnotu má parametr „MaxRowCount“. Příklad datového souboru:

```

902 neco=78
903 param2=retezec
904 MaxRowCount=32
905 dalsipar=19
906 MaxRowCount=17
907 konec=10

```

Řešení:

```

908 #include <iostream>
909 #include <fstream>
910 #include <cstring>
911 using namespace std;
912
913 int main(){
914     ifstream konf ("datarc");
915     if (konf.is_open()) {
916         //kontrola použitelnosti souboru
917         char par[30]; //proměnné pro čtení
918         string s;
919         int num;
920         while (not konf.eof()) {
921             konf.getline(par, 30, '=');
922             if (strcmp(par,"MaxRowCount")==0)
923                 konf>>num;
924             else getline(konf, s);
925         }
926         cout << "Hodnota je " << num << endl;

```

```
927     } else
928         cerr << "Soubor datarc nelze otevřít."
929         << endl;
930     return 0;
931 }
```

13.2.3 Zpracování netextových souborů

Výměna dat v netextových souborech probíhá bez konverze a také bez kontroly datových typů. Netextový soubor se jeví jako posloupnost bajtů, takže při jakékoliv výměně dat se předpokládá, že v operační paměti máme taky jen posloupnost bajtů. Jeden bajt v jazyce C++ je reprezentován datovým typem **char**, takže z tohoto pohledu lze každé znakové pole chápat jako posloupnost bajtů. Této skutečnosti se plně využívá při práci s netextovými soubory.

Pro čtení a zápis předpokládejme tuto deklaraci: `fstream soubor; char pole[N];`

Pro *čtení* existuje metoda `soubor.read(pole, M);`, kde `M` je požadovaný počet přenesených bajtů. Tato celočíselná hodnota není nijak kontrolována vzhledem k deklarované velikosti `N` příslušného bajtového pole. Lze číst jakákoliv data, je pouze potřebné přetypovat vše na typ **char***. Metoda `soubor.gcount()` udává skutečný počet přečtených bajtů, čehož můžeme využít například v případě, že nařídíme čtení většího počtu bajtů než zbývá do konce souboru. Příklad:

```
932 struct Data {
933     int klic;
934     double hodnota;
935 };
936 Data *x;
937 soubor.read((char*)x, sizeof(Data));
```

Velmi podobně probíhá *zápis*. Podobně jako u čtení existuje metoda `soubor.write(pole, M);`. Pokud vypisujeme jakoukoliv jinou proměnnou než bajtové pole, opět musíme provést přetypování na typ **char***. Musíme si přitom uvědomit, že měníme *adresu*, takže nemáme-li ukazatel na příslušnou proměnnou, musíme adresu získat referencí – viz řádek 945.

Příklad:

```
938 struct Data {
939     int klic;
940     double hodnota;
941 };
942 Data *x;
943 int Pocet;
944 soubor.write((char*)x, sizeof(Data));
945 soubor.write((char*)&Pocet, sizeof(int));
```

Metody `read` a `write` představují natolik obecný přesun dat, že je lze využívat i u textových souborů.

Příklad: Předpokládejme, že v paměti je naplněno pole záznamů o zboží. Chceme obsah celého pole uložit do binárního souboru s názvem „data.bin“. Vypisovat do souboru ovšem budeme pouze naplněné indexy, jejichž počet je v proměnné `Naplнено`. Definice datového typu pole záznamů:

```

946 const int MaxPole = 300;
947 struct TypZbozi {
948     string Nazev;
949     int PocKusu;
950     float CenaKus;
951 };
952 typedef TypZbozi TypSklad[MaxPole];
953 TypSklad Sklad;
954 int Naplнено; //počet naplněných složek pole

```

Předpokládejme, že pole je již naplněno a vypíšeme je do souboru:

```

955 int *UkNaplнено=&Naplнено;
956     //ukazatel vhodný k přetypování
957 fstream vystup;
958 vystup.open("data", ios::binary|ios::out);
959     //binární soubor otevřený pro zápis
960     //nejprve vypíšeme počet záznamů:
961 vystup.write((char*)UkNaplнено, sizeof(int));
962     //totéž by bylo: vystup.write((char*)&Naplнено, sizeof(int));
963     //a následně celé pole najednou:
964 vystup.write((char*)Sklad, Naplнено*sizeof(TypZbozi));
965     //soubor je pak nutné uzavřít:
966 vystup.close();

```

13.2.4 Ovládání pozice v souboru

Čtení a zápis v souboru se typicky chovají jako *sekvenční* operace, tj. při každém vyvolání provedou přenos dat a automaticky posunou aktuální pozici. Na binární soubor lze ovšem pohlížet i jako na pole bajtů, kde každý bajt má svůj index (číslováno od nuly jako každé jiné pole v C++). Otevřený soubor udržuje informaci o aktuální pozici čtení (bývá označována jako tzv. **get pointer**) a aktuální pozici zápisu (**put pointer**). Obě pozice jsou na sobě navzájem nezávislé a jsou datového typu `long int`. Obě pozice lze zjišťovat, k tomu jsou k dispozici metody `soubor.tellg()` a `soubor.tellp()`. Obě pozice lze také nastavit metodami `soubor.seekg(index)` a `soubor.seekp(index)`.

Jednoduchý příklad: Předpokládejme, že v paměti existuje naplněné pole celočíselných hodnot o `VelPole` prvcích. Toto pole vypíšeme do netextového souboru. Později se rozhodneme, že někte-

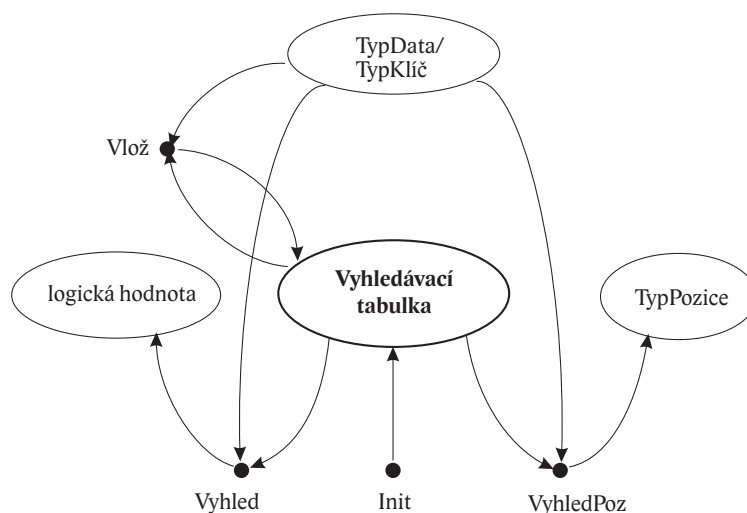
rý údaj v souboru potřebujeme za určitých podmínek aktualizovat – vytvoříme program, do něhož zadáme dvě hodnoty: první hodnota se má v souboru najít a pak nahradit tou druhou. Řešení:

```
967 int main() {
968     const int VelPole = 100;
969     int pole[VelPole];
970     for (int i=0; i<VelPole; i++) pole[i]=2*i;
971     //pole je něčím naplněno...
972     int Hledat, Nahradit, ZPole;
973     //co hledáme, čím nahrazujeme, kam čteme
974     fstream bindata ("data.bin", ios::binary | ios::out);
975     //soubor otevřeme pro zápis
976     bindata.write((char*)pole, VelPole*sizeof(int));
977     //vypíšeme pole do souboru
978     bindata.close(); //soubor uzavřeme...
979     bindata.open("data.bin", ios::binary | ios::out | ios::in);
980     //... a otevřeme pro čtení i zápis
981
982     //Nyní budeme aktualizovat:
983     cin >> Hledat >> Nahradit;
984     //vstupní data
985     for (int i=0; i<VelPole; i++) {
986         bindata.seekg(i*sizeof(int));
987         //nastavíme pozici v souboru
988         bindata.read((char*)&ZPole, sizeof(int));
989         //přečteme, pozice se automaticky posune
990         if (ZPole==Hledat) { //našli jsme, chceme aktualizovat
991             bindata.seekp(i*sizeof(int)); //návrat pozice
992             bindata.write((char*)&Nahradit, sizeof(int));
993             //přepíšeme hodnotu v souboru
994         }
995     }
996     bindata.close(); //ukončení práce se souborem
997     return 0;
998 }
```

14 Algoritmy vyhledávání

Vyhledávání je bezesporu jedna z nejzákladnějších funkcí, kterou všechny aplikace v počítači řeší.

Mějme hodnotu **C** a datovou strukturu **S**. Základní vyhledávací úloha implementuje zjištění, zda **C** je obsaženo v **S**. Výsledkem je tedy logická hodnota. Často se však za výsledek považuje místo (index, ukazatel), kde byla hodnota nalezena. Struktura určená pro vyhledávání je implementací abstraktního datového typu Vyhledávací tabulka. Slovo „tabulka“ v názvu typu je spíše symbolické, s žádnými konkrétními tabulkami nesouvisí. Typické operace – viz diagram signatury na obr. 21.



Obrázek 21: Diagram signatury abstraktního typu Vyhledávací tabulka

14.1 Sekvenční vyhledávání

Sekvenční vyhledávání (nazývané také lineární vyhledávání) funguje na principu procházení všech prvků zadané struktury – každý z prvků seznamu se postupně porovnává s hledanou hodnotou.

Sekvenční vyhledávání není tak efektivní jako jiné metody, jeho výhodou však je možnost použití i v neseřazených strukturách. Používá se také pro lineární dynamické struktury.

Sekvenční vyhledávání má lineární časovou složitost $O(N)$. V případě náhodného rozložení je průměrně potřeba $N/2$ porovnání. Nejlepší případ nastane, pokud se hledaná hodnota nachází na prvním místě v seznamu, nejhorší případ naopak nastane tehdy, když se hodnota v seznamu nenachází. Máme-li v případě zcela náhodného uspořádání a náhodného hledání průměrně 50 % úspěšných vyhledání, na úspěšné vyhledání potřebujeme průměrně $N/2$ porovnání. Na neúspěšné musíme projít celou strukturu, takže N porovnání. Sloučíme-li oba případy dohromady, vychází průměrný počet testů na $\frac{3}{4}N$, což je lineární funkce.

V dalším textu budeme předpokládat, že máme pole **P**, v němž jsou umístěny hledané hodnoty, přičemž obsazeny jsou indexy od nuly do **N-1**. Dále budeme předpokládat, že kapacita pole je větší než **N**, pole není zcela zaplněno. Výsledkem hledání bude hodnota v logické proměnné **Nalezeno**.

Sekvenční hledání – přímá metoda:

```
999 | int i=0; //nastavení na první prvek
1000 | while (i<N /*je kde hledat*/ and P[i]!=C /*dosud nenalezeno*/)
1001 |     i++; //přecházíme k dalšímu prvku
1002 | Nalezeno = i<N; //úspěšné hledání: nevyčerpali jsme všechny prvky
```

14.2 Sekvenční vyhledávání se zarážkou

Modifikací základní metody je **sekvenční hledání se zarážkou**. Za poslední prvek se umístí hledaná hodnota, která již logicky nepatří do původního seznamu. Hodnota je tak vždy nalezena, jde o to, ve kterém místě. Výhoda spočívá ve zkrácení testu v cyklu, protože se nemusíme dotazovat na konec seznamu (konec se pozná podle uměle vložené hledané hodnoty). Jistým omezením je, že pole musí mít ještě alespoň jeden volný index, na který se zarážka umístí.

```
1003 | P[N] = C; //umístění zarážky
1004 | int i=0; //nastavení na první prvek
1005 | while (P[i]!=C) //testujeme jen nalezení
1006 |     i++; //přecházíme k dalšímu prvku
1007 | Nalezeno = i<N; //úspěšné hledání: index menší než N
```

I tato modifikace má časovou složitost lineární, mírně se jen mění implementační konstanta.

14.3 Sekvenční vyhledávání v uspořádaném poli

Další modifikací oproti základní metodě je vyhledávání v uspořádaném poli, kde informace o nenalezení nevyžaduje průchod celým polem. Je-li hledaná hodnota menší než aktuální ve vzestupně uspořádaném poli, máme jistotu, že dále už nemá smysl pole procházet.

```
1008 | int i=0;
1009 | while (i<N and P[i]<C) //hledaná hodnota ještě může být dále
1010 |     i++;
1011 | Nalezeno = i<N and P[i]==C; //úspěch: nejsme na konci a bylo nalezeno
```

I tato modifikace má stejnou lineární časovou složitost. Pokud by bylo řazení provedeno jen kvůli hledání, jde o velmi neefektivní postup.

14.4 Vyhledávání půlením intervalu

Pokud vyhledáváme v uspořádaném poli, je mnohem efektivnější vyhledávání využívající plně informací plynoucích z uspořádání hodnot. Nejznámější algoritmus zvaný často **půlení intervalu** se orientuje podle hodnoty uprostřed prohledávaného úseku seřazeného pole.

Hledání začíná v polovině uspořádané struktury a podle hodnoty, která se zde nachází, se můžeme rozhodnout, zda budeme dále hledat v dolní (první, levé) polovině, nebo v horní (druhé, pravé) polovině.

Každý průchod cyklem tedy zmenší počet prvků, mezi nimiž se hledá, na polovinu. Díky tomu metoda dosahuje složitosti $O(N) = K \cdot \log_2 N$, a je tak nesrovnatelně rychlejší než sekvenční vyhledávání v libovolné variantě. Hledání končí, když už není co půlit (daný interval obsahuje jediný prvek pole), nebo byla nalezena hodnota **C**.

```

1012 int levy = 0; //interval hledání
1013 int pravy = N-1;
1014 int index = (levy+pravy)/2;
1015 while (levy<pravy and P[index]!=C){
1016     if (C < P[index]) pravy=index-1; //jdeme vlevo
1017     else levy=index+1; //jdeme vpravo
1018     index=(levy+pravy)/2;
1019 }
1020 Nalezeno = P[index]==C;

```

14.5 Stromové vyhledávání

Hlavní využití binárního vyhledávacího stromu lze spatřovat v implementaci vyhledávání s logaritmickou časovou složitostí a v implementaci řazení s lineárně logaritmickou časovou složitostí.

Operace vyhledávání je zapsána jako sekvenční hledání, ale při přechodu k dalšímu prvku se rozhodujeme, zda půjdeme do levého, nebo do pravého podstromu, čímž vždy jeden z podstromů zcela vynecháváme. Počet kroků je tedy závislý nikoliv na celkovém počtu uzlů N , ale na počtu hladin h daného stromu. Za předpokladu, že strom je vyvážený (má přibližně stejné počty uzlů ve všech sousedních podstromech), pak je počet hladin dán $h = \log_2 N$.

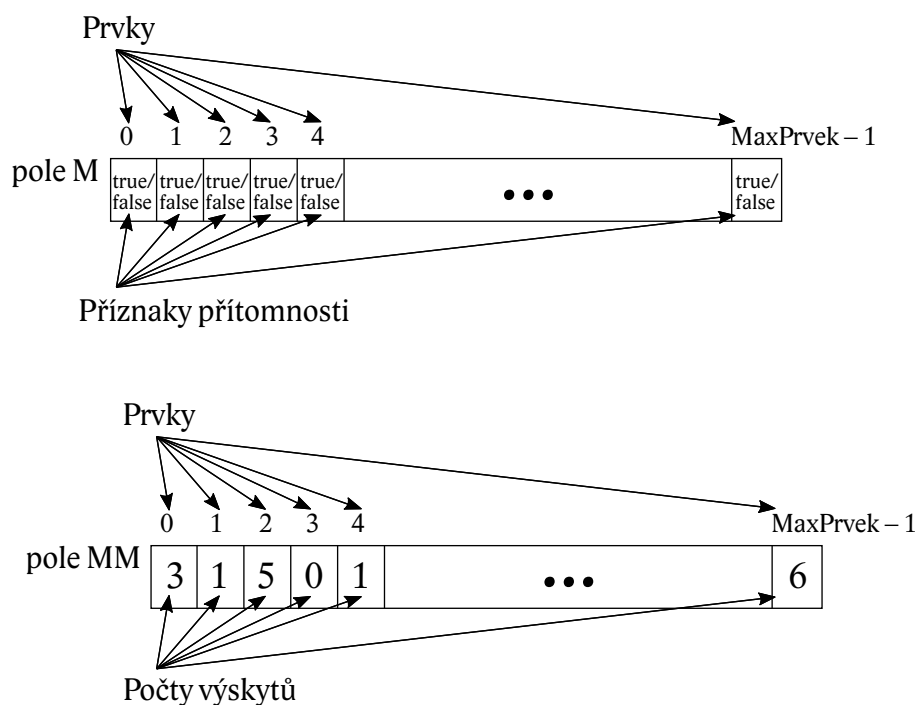
14.6 Vyhledávání indexací

Mezi algoritmy vyhledávání existují i případy s konstantní časovou složitostí. Nelze je bohužel aplikovat úplně obecně, protože do značné míry závisejí na datech – přesto však mohou zcela zásadním způsobem změnit chování programového díla, ve kterém jsou nasazeny.

Základní myšlenka spočívá v implementaci množiny, jak ukážeme v následující části.

14.6.1 Implementace množiny a multimnožiny

Matematickou množinu můžeme chápat jako skupinu prvků, o nichž můžeme jednotlivě prohlásit, zda do té skupiny patří, či nikoliv (zda jsou obsaženy). Pokusíme-li se implementovat takovou vlastnost, potřebujeme u každého prvku atribut (vlastnost) typu **bool** udávající, zda daný prvek je, či není v množině obsažen.



Obrázek 22: Implementace množiny a multimnožiny

Efektivní implementaci můžeme udělat tak, že prvky množiny budou používány jako *indexy* v poli logických hodnot (viz horní část obr. 22), například:

```

1021 typedef unsigned int TypData;
1022 const TypData MaxPrvek = ... //konstanta určující možný rozsah hodnot
1023 typedef bool Mnozina[MaxPrvek]; //pro prvky s hodnotami 0 až MaxPrvek-1

```

Matematickou multimnožinu, tj. množinu, jejíž prvky se mohou vyskytovat vícenásobně, lze jednoduše implementovat polem s celočíselnými hodnotami. Opět platí, že prvky jsou *indexy* tohoto pole, počty výskytů jsou pak umístěny v jednotlivých složkách pole (dolní část obr. 22):

```

1024 typedef unsigned int TypData;
1025 const TypData MaxPrvek = ... //konstanta určující možný rozsah hodnot
1026 typedef unsigned int MultiMnozina[MaxPrvek]; //prvky 0 až MaxPrvek-1

```

Z implementace přímo vyplývá zásadní omezení, o jaké prvky se může jednat: prvky tvoří indexy pole, tj. celočíselný interval od nuly do hodnoty **MaxPrvek-1**. Samozřejmě je možné takový interval nahradit i jiným, při indexaci se napřed hodnota prvku přepočítá (lineárně posune) a pak se teprve

použije jako index. Mějme například data v intervalu $\langle -700; 1500 \rangle$. Můžeme vytvořit následující pomůcku:

```

1027 const int DolniMez = -700;
1028 const int HorniMez = 1500;
1029 unsigned int Prepocet(int Data){
1030     return Data-DolniMez;
1031 }
```

Lze si představit v některých speciálních případech i komplikovanější přepočet – například jsou-li prvky pouze sudá čísla, nemusíme liché indexy ponechávat neobsazené a můžeme využít každou položku pole, pokud prvek vydělíme dvěma.

S rozsahem hodnot prvků v množině souvisí velikost pole. Je zřejmé, že bude-li rozsah hodnot velký, bude také vzrůstat prostorový nárok na alokaci pole. Většinou však platí, že nárok na velký prostor není překážkou a za tu značnou úsporu času tuto platbu rádi vynaložíme.

14.6.2 Vyhledávání v množině

Samotné vyhledání prvku C v množině (zjištění přítomnosti) spočívá v pouhé indexaci $M[C]$. Indexový výraz se vypočítá ve stejném čase pro jakýkoliv počet vstupních dat.

Množinu je před použitím nutné vyprázdnit a pak ji naplnit daty. Následně lze vyhledávat. Mějme například vstupní hodnoty v intervalu $\langle -5000; 10000 \rangle$. Příprava množiny pro vyhledávání pak může vypadat takto:

```

1032 const int DolniMez = -5000;
1033 const int HorniMez = 10000;
1034 typedef int TypData;
1035 const TypData MaxPrvek = HorniMez-DolniMez+1;
1036 typedef bool Mnozina[MaxPrvek];
1037 unsigned int Prepocet(int Data){
1038     return Data-DolniMez;
1039 }
1040 Mnozina M;           //deklarace množiny
1041 TypData vstup;       //proměnná pro čtení vstupních dat
1042 //Vyprázdnění množiny:
1043 for (TypData I=DolniMez; I<=HorniMez; I++)
1044     M[I]=false;
1045 //Vložení vstupních dat do množiny:
1046 while (cin>>vstup) M[vstup]=true;
```

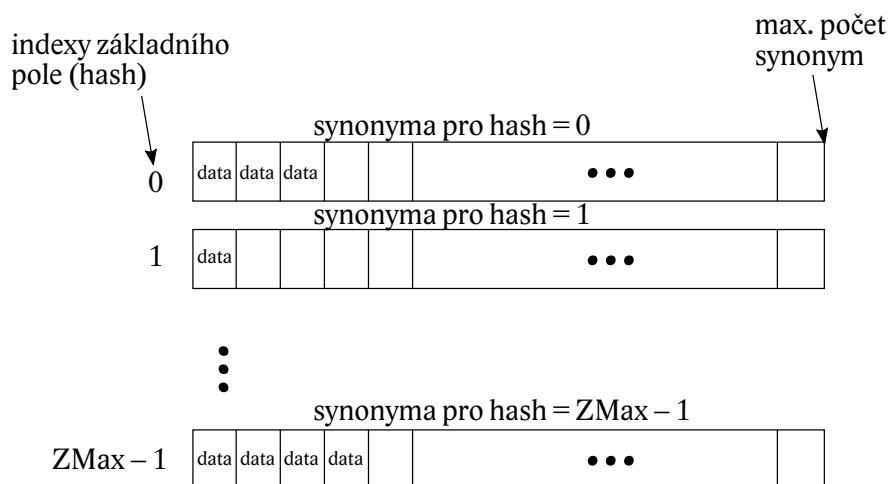
14.7 Vyhledávání hašováním

Z implementace množiny vyplývá zásadní omezení pro možné typy a hodnoty, které lze takto vyhledávat. Nabízí se přirozeně otázka, zda by se i další datové typy mohly po určité úpravě využít pro indexaci. Alespoň částečně lze řešit i jiné množiny dat pomocí hašování. Pojem **hash** (v počestěné podobě také haš nebo heš) lze přeložit jako rozptýlení, rozmělnění, rozemletí – jde o speciální funkci, která přepočítá (rozemele, rozptýlí) jakoukoliv vstupní hodnotu na celočíselnou hodnotu, která se následně využije pro různé účely, mimo jiné i pro indexaci v množině.

Máme-li jakákoliv data zvoleného typu (například řetězce), musíme k nim najít odpovídající hašovací funkci. Mělo by to fungovat tak, že při zadání dvou rozdílných řetězců dostaneme vždy rozdílné výstupní celočíselné hodnoty. V praxi ovšem tohoto ideálního stavu dosáhneme jen velmi obtížně. Může se často stát, že funkce nepředstavuje bijektivní zobrazení původních dat na celočíselnou množinu, tudíž ze dvou rozdílných vstupů vyrobí stejnou výstupní hodnotu. Pak ovšem nemůžeme jednoznačně výstupním indexem vyjádřit původní data a musíme uložit všechny jejich varianty. Datové hodnoty, které po hašování produkují stejnou hodnotu, se v této souvislosti nazývají **synonyma**. Ukládání synonym je klíčovou úlohou reprezentace dat v takové vyhledávací struktuře.

Logickým krokem v této souvislosti je, že prvkem „množinového“ pole budou samotná data, přičemž indexem bude hodnota vzniklá hašováním. Protože ovšem na jednom indexu může být více synonym, uložíme je rovněž do pole. Jeho velikost musíme odhadnout podle předpokládaného maximálního počtu synonym.

Vzniklá struktura je nazývána **hašovací tabulka** a je znázorněna na obr. 23.



Obrázek 23: Princip konstrukce hašovací tabulky

Místo polí pro ukládání synonym lze využít i dynamické struktury.

14.7.1 Práce s hašovací tabulkou

Pro práci s tabulkou potřebujeme umět vkládat data a zjišťovat jejich přítomnost. Před započítím práce je nutné tabulku inicializovat. Typové definice a příslušné operace mohou mít následující tvar:

```

1047 typedef unsigned int Indexy; //datový typ pro indexy
1048 const Indexy VelikostZP = 2581; //velikost základního pole
1049 const Indexy MaxSynonym = 1000; //odhad maximálního počtu synonym
1050 typedef TypData PoleSynonym[MaxSynonym]; //pole synonym
1051 struct Synonyma{ //pole synonym + jejich aktuální počet
1052     PoleSynonym Syn;
1053     Indexy Zaplneno;
1054 };
1055 typedef Synonyma HashTab[VelikostZP]; //datový typ hašovací tabulky
1056 Indexy Hash(TypData D){ //hašovací funkce
1057     //výpočet indexu základního pole ze zadaných dat
1058     return vysledek;
1059 }
1060 void TabInit(HashTab H){
1061     //počty synonym se nastaví na všude na nulu
1062     for (Indexy i=0; i<VelikostZP; i++)
1063         H[i].Zaplneno=0;
1064 }
1065 void TabVloz(HashTab H, TypData D){
1066     //vkládání dat probíhá v konstantním čase
1067     Indexy I = Hash(D); //zjistíme, na který index data patří
1068     H[I].Syn[H[I].Zaplneno]=D; //přidáme na konec pole synonym
1069     H[I].Zaplneno++; //upravíme počet obsazených prvků
1070 }
1071 bool TabHledej(HashTab H, TypData D){
1072     //hledání probíhá v lineárním čase, ale pouze v seznamu synonym
1073     Indexy I = Hash(D), J; //zjistíme, na kterém indexu se bude hledat
1074     J=0; //sekvenční hledání, ale v malém počtu hodnot
1075     while (J<H[I].Zaplneno and H[I].Syn[J]!=D) J++;
1076     return J<H[I].Zaplneno;
1077 }

```

Ukládání dat se provádí v konstantním čase, vyhledávání je sice v lineárním čase, ale za předpokladu dobrého rozptýlení dat se počty synonym minimalizují a hledání je velmi krátké. Nezávisí tedy na celkovém počtu vstupních dat, ale pouze na maximálním počtu synonym. Někdy se tento způsob vyhledávání také nazývá **index-sekvenční** – napřed se indexuje (konstantní složitost) a pak se dohledá sekvenčním algoritmem v malém počtu hodnot.

14.7.2 Hašování

Je zřejmé, že klíčem k efektivní práci hašovací tabulky je vlastní rozptýlení (hašování) dat na jednotlivé indexy základního pole. Hašovací funkce by měla splňovat dva (obvykle protichůdné) požadavky: vytvářet rovnoměrně indexy po celém rozsahu základního pole a současně být rychlá. To, že funkce

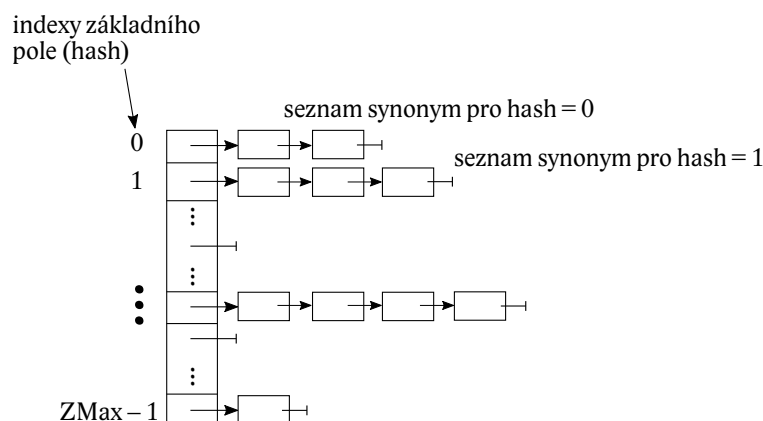
pracuje v konstantním čase, je naprostou samozřejmostí. Můžeme si ukázat několik příkladů, jak hašovací funkci navrhnout. V podstatě jde o jeden až dva kroky: nejprve se získá nějaká celočíselná hodnota ze zadaných dat (pokud data přímo nejsou celočíselná – viz první příklad) a pak se upraví na interval indexů základního pole:

```
1078 //celočíselná data:
1079 Indexy Hash(int data){
1080     return abs(data) % VelikostZP;
1081     //jde o přepočítání na interval 0 až VelikostZP-1
1082 }
1083 //řetězcová data:
1084 Indexy Hash(string data){
1085     int pom=0;
1086     for (int i=0; i<data.length(); i++)
1087         pom += int(data.at(i)); //sečteme všechna ordinální čísla
1088     return pom % VelikostZP;    //a přepočítáme na vhodný interval
1089 }
1090 //desetinná čísla jako data
1091 Indexy Hash(double data){
1092     return int(data*1000) % VelikostZP;
1093 }
```

Principům hašování obecných hodnot (o nichž nemáme žádné konkrétní informace) se věnuje řada vědeckých studií, protože se jedná zároveň i o oblast zabezpečení. Hloubavé čtenáře můžeme odkázat zejména na různé internetové zdroje.

14.7.3 Hašovací tabulka s dynamickými seznamy synonym

Jak bylo již dříve ukázáno, hašovací tabulka potřebuje ukládat synonyma. Pro tento účel bylo použito pole, u něhož se musí dopředu ohlásit, kolik prvků bude mít. To ovšem pro obecná data nemůžeme tušit, a z toho důvodu volíme dostatečně velkou dimenzi (plýtvání pamětí). Zde se velmi hodí, pokud synonyma neukládáme do pole, ale do dynamického seznamu. Počet jeho položek může být zcela libovolný a obvykle je přídatná režie s ukládáním ukazatelů zcela zanedbatelná proti případu alokace polí synonym. Tvar hašovací tabulky s lineárními seznamy přináší obr. 24.



Obrázek 24: Hašovací tabulka se seznamy synonym

V této podobě se mírně mění tvar datového typu a příslušných operací s tabulkou. Základní pole obsahuje ukazatele na lineární seznamy. Při inicializaci se na všechny indexy základního pole vloží prázdný ukazatel **NULL**. Synonyma se vkládají jako prvky těchto seznamů. Protože nezáleží na pořadí, přidává se nový prvek vždy na začátek seznamu synonym, takže vložení hodnoty má konstantní časovou složitost. Vyhledání se pak provede sekvenčním algoritmem v seznamu synonym. Implementace může mít následující tvar:

```

1094 struct Clen {
1095     TypData Data; //typ dat podle potřeby
1096     Clen *Dalsi;
1097 };
1098 typedef Clen *UkClen;
1099 typedef unsigned int Indexy;
1100 const Indexy Max = 8951; //délka základního pole
1101 typedef UkClen ZaklPole[Max];
1102 typedef Indexy (*HashFce)(TypData); //hašovací funkce
1103 struct HashTab {
1104     ZaklPole ZP;
1105     HashFce H;
1106 };
1107
1108 void HTInit(HashTab &HT, HashFce X){
1109     for (Indexy i=0; i<Max; i++) HT.ZP[i]=NULL;
1110     HT.H = X;
1111 }
1112 void HTVloz(HashTab &HT, TypData D){
1113     Indexy I = HT.H(D); //hašování
1114     UkClen Pom = new Clen; //přidání nového prvku na začátek seznamu
1115     Pom->Data = D;
1116     Pom->Dalsi = HT.ZP[I];
1117     HT.ZP[I] = Pom;

```

```
1118 }  
1119  
1120 bool HTNajdi(HashTab HT, TypData D){  
1121     UkClen Pom = HT.ZP[HT.H(D)];    //hašování a start seznamu synonym  
1122     while (Pom!=NULL and Pom->Data!=D)  
1123         Pom = Pom->Dalsi;  
1124     return Pom != NULL;  
1125 }
```

15 Algoritmy řazení

Řazení je uspořádání datových položek podle specifikovaného pořadí. Předpokládá se, že nad daty existuje relace (úplného) uspořádání. Čísla se obvykle řadí podle velikosti, řetězce abecedně, jsou ale samozřejmě možné i jiné způsoby. Smysl řazení může být **sestupný** (od největšího prvku po nejmenší) nebo **vzestupný** (od nejmenšího po největší).

Někdy je řazení *nesprávně* označováno jako třídění, tříděním však rozumíme rozdělování dat do skupin (klasifikace) a v této souvislosti potřebujeme, aby nad daty existovala jiná relace – ekvivalence. K tomuto zmatení pojmů došlo v minulosti, kdy byla jedna z řadících metod používající roztřídění dat do skupin nazývána „řazení tříděním“, krátce „třídění“. Nedbalým překladem pojmu „sort“ z angličtiny se masivně tento omyl přenesl na všechny ostatní řadící metody.

15.1 Vlastnosti řadících metod

K řazení je možné použít řadící metody (algoritmy) různých vlastností, které se také hodí pro různé účely. Než se budeme věnovat jednotlivým metodám, zaměříme se na obecné vlastnosti řadících metod.

Sekvenčnost – řadící metoda přistupuje k řazeným datům sekvenčně, je vhodná na aplikaci v případě, kdy například data jsou v seznamu, nikoliv v poli. Nesekvenční metoda přistupuje přímo, potřebuje tedy umět v jakémkoliv okamžiku přistoupit k jakémukoliv prvku řazených dat.

Přirozenost – přirozená řadící metoda je schopna využít seřazenost vstupních dat, seřazenou množinu tak zpracovává rychleji než neseřazenou.

Stabilita – stabilní řadící metoda nemění pořadí shodných položek.

Práce in situ (v místě samém) – metoda nepotřebuje podstatně více místa, než zabírají vstupní data, tato dodatečná paměť má relativně malou (a zpravidla na množství dat nezávislou) velikost. Algoritmy, které nepracují in situ, potřebují více dodatečné paměti pro zpracování, obvykle se jedná o případ, kdy se zdrojová data při řazení kopírují do druhé paměťové oblasti (pole, souboru) nebo se při řazení používá rekurze.

Vícekritériální řazení je řazení podle více klíčů (kritérií). Prvky jsou složeny z více složek, z nichž některé slouží jako řadící klíče. Řazení probíhá podle jednoho z klíčů; jsou-li hodnoty tohoto klíče u několika položek stejné, rozhoduje o pořadí hodnota další složky. Typický příklad: Seznam osob řazený podle příjmení, u stejných příjmení podle jména, u stejných příjmení i jmen řazení podle věku atd.

Někdy se vícekritériální řazení provádí vícenásobným řazením postupně podle různých klíčů. Základním požadavkem na použitou řadící metodu je pak její stabilita.

Nepřímé řazení (řazení bez přesunu položek) – vytvoření pomocného souboru, v němž jsou uspořádaně uvedeny pouze hodnoty klíčů s odkazem (indexem) do původního souboru. Při samotném řazení není třeba přesunovat velké záznamy, přesunují se pouze klíče (velmi často se používá v databázových systémech – tzv. indexování).

Dále lze algoritmy rozdělit podle *způsobu řazení*. Základní dělení je na metody *rozdělovací* (výběr, rozdělení) a metody *sdužovací* (vkládání, slučování). Základní princip realizuje tzv. *přímá metoda*. Modifikace základního algoritmu nazýváme nepřímé (odvozené, zdokonalené) metody.

Výběrové metody – v datech se najde extrém (maximum, minimum) a umístí se na konec nově vznikající seřazené posloupnosti dat. Mohou se použít dvě pole dat (zdrojové, cílové). Zdrojová část se tedy zpracovává nesequenčně, cílová část narůstá sekvenčně.

Vkládací metody – z dat (neseřazených) se postupně zpracovává jedna položka po druhé a vkládá se na správné místo v seřazených datech. Zdrojová část se tedy zpracovává sekvenčně, cílová nesequenčně.

Rozdělovací metody – vstupní data se rozdělují na části s prvky odpovídajícími určitému kritériu (například velikost vůči jinému vybranému prvku). Takto připravené úseky se pak dále dělí a zpracovávají podobným způsobem. Dělení končí u dvouprvkových (někdy jednoprvkových) úseků.

Slučovací metody – vstupní data se rozdělí na úseky, které se pak procházejí a slučují, při slučování se však prvky porovnávají a upravují do posloupnosti, která vykazuje seřazené části.

Uživatele zpravidla nezajímá, jak metoda pracuje, ale jak je pro něj výhodná. Velmi zajímavé je v tom případě hodnocení algoritmů podle *časové složitosti*. Řadící metody lze rozdělit na *kvadratické*, *lineárně logaritmické* a *lineární* (teoretické minimum). Podle tohoto kritéria budeme dále uvádět příklady konkrétních metod.

15.2 Přípravné definice

V následujících příkladech budeme vždy předpokládat řazení v poli o určitém počtu složek. Ve zdrojových textech jednotlivých metod využijeme následující definice:

```

1126 typedef unsigned int Indexy; //indexace pole nezápornými čísly
1127 typedef ... Slozka; //prvek pole, předpokládáme zde relaci uspořádání
1128 typedef Slozka Pole[N]; //řazené pole, N složek, indexy 0 až N-1
1129
1130 //obecná pomůcka: záměna prvků v poli
1131 void Zamena(Pole P, Indexy A, Indexy B){
1132     if (A!=B) { //má smysl zaměňovat
1133         Slozka Pom=P[A];
1134         P[A]=P[B];
1135         P[B]=Pom;
1136     }
1137 }
```

15.3 Kvadratické metody

Jde o kategorii, kterou můžeme nazvat jako doménu začátečníků, laiků nebo sebevědomých neználků – metody jsou primitivní, snadno pochopitelné a snadno zapsatelné. Většinou jsou však prakticky nepoužitelné pro rozsáhlejší data, kdy i při použití výkonného hardwaru trvá řazení dlouhé desítky sekund, minuty i déle. Jako reprezentanty uvedeme přímý výběr, bublinové řazení a přímé vkládání.

15.3.1 Přímý výběr

Další názvy: Straight selection; Select(ion) sort

Metoda přímého výběru pracuje nad jedním polem, jehož jedna část (např. počáteční) je seřazena, koncová je v původním uspořádání. Každým krokem metody se zvětšuje seřazená část a zmenšuje neseřazená. Pozice hranice mezi oběma částmi je uschována v indexu **I**. Metoda byla již zhruba popsána dříve v části týkající se algoritmů s polem.

Princip:

1. V neseřazené části se najde extrém (prvek s nejmenší/největší hodnotou).
2. Extrém se vymění s prvkem na pozici **I**.
3. Totéž se opakuje pro zbylých $N - 1$ prvků, dokud je $N > 1$. Pro poslední prvek už se nemusí nic dělat, protože on sám je sobě extrémem a už se nebude nikam přesunovat.

```

1138 Indexy Extrem (Pole P, Indexy A, Indexy B){ //nalezení pozice extrému
1139     Indexy V = A, I;
1140     for (I=A+1; I<=B; I++)
1141         if (P[I]<P[V]) V=I; // zde se definuje smysl řazení
1142     return V;
1143 }
1144 for (Indexy I=0; I<N-1; I++) //vlastní řazení
1145     Zmena(P, Extrem(P, I, N-1), I);

```

15.3.2 Bublinové řazení

Další názvy: Bubble sort

Jde o velmi populární metodu (patrně už kvůli názvu). Pochopí ji i velmi málo zdatný jedinec, pro neználky je to univerzální řadicí nástroj. Přímá varianta je nesmírně snadná na zápis, je ovšem značně nevhodná. Varianta byla popsána již dříve v části týkající se algoritmů pro práci s polem. Za zmínku ovšem stojí přirozená varianta – hodí se pro speciální případy, kdy může předčít i mnohem sofistikovanější metody. Jako jedna z mála metod je sekvenční – řazená data prochází postupně od začátku a k samotnému řazení a výměně potřebuje vždy jen dvě po sobě jdoucí složky.

Princip: pole dat se sekvenčně prochází a zjišťuje se, zda dvě po sobě jdoucí položky mají požadované pořadí. V záporném případě se provede vzájemná záměna těchto položek. Toto se opakuje, dokud není celá posloupnost seřazená (tj. v základní variantě po $N - 1$ průchodech, v přirozené variantě tehdy, když při procházení pole nedojde k žádné záměně).

Metoda má mnoho modifikací. Jedna z nejdůležitějších je modifikace zajišťující přirozenost, neboť obsahuje nejrychlejší detekci již seřazených dat.

Základní varianta:

```
1146 for (Indexy I=0; I<N-1; I++) //počet průchodů polem
1147     for (Indexy J=0; J<N-I-1; J++) //jeden průchod
1148         if (P[J]>P[J+1]) //pořadí nevyhovuje
1149             Zamena(P, J, J+1);
```

Přirozená varianta zbytečně neprochází znovu a znovu již seřazené pole. Stav, kdy už není potřeba nic dělat, se zjistí podle toho, že při posledním průchodu nedošlo k žádné záměně složek. K tomu slouží logická proměnná `Jeste`:

```
1150 Indexy J=0;
1151 bool Jeste=true; //vynutíme vstup do cyklu
1152 while (Jeste) {
1153     Jeste=false; //předpokládáme, že je již seřazeno
1154     for (Indexy I=0; I<N-J-1; I++) //provedeme průchod
1155         if (P[I]>P[I+1]) { //předpoklad se nesplnil, není seřazeno
1156             Zamena(P, I, I+1);
1157             Jeste=true;
1158         }
1159     J++; //zkrátíme počet prvků nutných k procházení
1160 }
```

15.3.3 Přímé vkládání

Další názvy: Straight insertion, Insert(ion) sort

Algoritmus přímého vkládání je založen na vkládání hodnot do seřazeného pole. Zdrojová data jsou buď získávána z vnějšku, nebo z jiné části téhož pole. Algoritmus přímého vkládání provádí opakovaně následující kroky:

1. Vyhledání vhodného místa pro hodnotu, kterou chceme vkládat.
2. Posun prvků v poli od vyhledané pozice směrem ke konci o jednu pozici.
3. Vložení nového prvku na uvolněnou pozici.

Algoritmus může mít následující tvar:

```

1161 Indexy vyhledej(Pole p, Indexy odkud, Indexy pokud, Slozka prvek){
1162     Indexy index=odkud;
1163     while (index<pokud and prvek<p[index]) index++;
1164     return index;
1165 }
1166
1167 void odsun(Pole p, Indexy odkud, Indexy kam){
1168     for (Indexy i=kam; i>odkud; i--) p[i]=p[i-1];
1169 }
1170
1171 //vlastní řazení:
1172 Indexy kam;
1173 Slozka novy;
1174 for (Indexy i=0; i<N; i++){
1175     novy=pole[N-1];    //vezmeme poslední prvek pole
1176     kam=vyhledej(pole, 0, i, novy); //vyhledáme místo, kam patří
1177     odsun(pole, kam, N-1); //odsuneme prvky dozadu
1178     pole[kam]=novy;    //vložíme na uvolněné místo
1179 }

```

Metodu můžeme zlepšit tím, že vyhledávání vhodného místa (prováděné v seřazené části) provedeme metodou půlení intervalu. To je metoda s logaritmickou složitostí. V základním cyklu se nachází i odsun, který je v lineární složitosti, má ovšem při implementaci obvykle nízkou implementační konstantu (může být řešen jako jedna instrukce pro přesun bloku paměti).

15.4 Lineárně logaritmické metody

Do této skupiny patří nejpoužívanější a nejznámější „profesionální“ metody: řazení hromadou, Quick sort, stromové řazení, řazení slučováním.

15.4.1 Řazení hromadou

Další názvy: Řazení haldou, Heap sort

Vtip této metody spočívá ve dvou skutečnostech:

1. Složky pole si představíme jako binární stromovou strukturu – kořenem je složka s indexem 1, každý levý potomek má index roven dvojnásobku indexu rodiče a každý pravý potomek má index rovný indexu levého syna zvětšeného o 1. Je-li index rodiče x , index levého syna je $2 \cdot x$ a index pravého syna $2 \cdot x + 1$. Pole naplňujeme od indexu 1, nikoliv od indexu 0. Složka na indexu nula zůstane nevyužita, poslední platný index je N . Pokud tomuto stavu chceme zabránit, musíme při každé indexaci přičítat jedničku (zbytečně nepřehledné a časově ztrátové).

2. Pojem **hromada** (halda, heap) zde označuje stav takového uspořádání, že rodič je vždy větší (menší – záleží na smyslu řazení) než jeho potomci, mezi potomky není žádný vztah definován.

Pokud je v tomto stromu ustavena hromada, je v jeho kořeni extrémní prvek. Dále budeme uvažovat vzestupné řazení, v kořeni stromu bude maximum.

Metoda pracuje v těchto dvou krocích:

1. Ustavení hromady – postupně se ustaví pravidla hromady počínaje posledním otcem a jeho syny a konče kořenem. Poslední otec ve stromu má index $N/2$, pokud je N liché, pak existuje i pravý syn, jinak existuje jen levý. Ustavení pravidel hromady je schopna zařídit v zadané části stromu procedura **Sift** (angl. prosít, vytrítit, přeneseně také zatřepat stromem).

```
1180 | for (Indexy I=N/2; I>=1; I--)
1181 |     Sift(P, I, N);
```

2. Ustavením pravidel hromady se extrémní prvek dostává do kořene stromu (na index 1). Následně se cyklicky tento kořenový prvek neboli vrchol hromady vyměňuje s posledním listem. Když se poslední list výměnou dostane do kořene, poruší se pravidla hromady. Po každé výměně se proto opět volá pomocná procedura **Sift**, která řeší ustavení pravidel hromady, tentokrát v celém zbývajícím poli. Největší prvek se tak dostane na místo, kam patří, a pro další manipulace se již s tímto prvkem nepracuje (zmenší se hodnota posledního indexu). Tím dochází ke zkracování pole, v němž se pracuje (tj. ke zmenšování hromady), protože prvky za koncem hromady jsou už seřazené.

```
1182 | for (Indexy I=N; I>=2; I--){
1183 |     Zamena(P, 1, I);
1184 |     Sift(P, 1, I-1);
1185 | }
```

Klíčová procedura **Sift** má logaritmickou složitost – pracuje napříč „hladinami“ stromu. Protože tato stromová představa navíc reprezentuje perfektně vyvážený strom, je počet hladin roven $\log_2 N$ a to je zároveň i maximální počet kroků, které procedura **Sift** na jakémkoliv zavolání provede. Všimněte si také, jak řeší vzájemné výměny rodičů a potomků: na začátku si uschová nejvyššího otce, se kterým začíná pracovat, pak podle potřeby stěhuje jednotlivé syny vždy do vyšší hladiny, až najde místo, kde přesuny končí, a uloží do něj uschovaného otce.

```
1186 | void Sift(Pole P, Indexy L, Indexy R){
1187 |     Indexy I = L, J = I*2;
1188 |     Slozka Pom = P[I];
1189 |     bool Jeste = J<=R;
1190 |     while (Jeste) {
1191 |         if (J<R and P[J]<P[J+1]) J++;
1192 |         if (P[I]<P[J]) {
1193 |             P[I]=P[J];
```

```

1194         I=J; J=I*2;
1195         Jeste = J<=R;
1196     } else Jeste=false;
1197     P[I]=Pom;
1198 }
1199 }
```

Metoda je nepřírozená, nestabilní, in-situ. Jedná se o jednu z nejrychlejších obecných řadicích metod s minimální prostorovou složitostí. V rychlosti někdy dokonce může předčit i metodu Quick sort, protože není závislá na datech a nemá takovou prostorovou složitost vznikající rekurzivním zápisem.

15.4.2 Quick sort

Quick sort (rychlé řazení) je metoda, která odpovídá svému názvu, jde o jeden z nejrychlejších známých obecně použitelných řadicích algoritmů založených na porovnávání.¹⁰

Principem metody je rozdělování řazeného pole na úseky, v nichž se provede přeuspořádání prvků. Z daného úseku pole je vybrán jeden prvek, který pole svou hodnotou určí rozdělení na dvě části (říkáme mu **pivot**).

Daný úsek pole je procházen z obou stran a hledají se prvky, které nevyhovují svou hodnotou dané pozici, tj. v levé části pole jsou větší než pivot a v pravé části pole jsou menší než pivot (předpokládáme-li vzestupné řazení). Je-li nalezena na obou stranách současně taková nevyhovující dvojice, je zaměněna – oba prvky pak vyhovují. Proces končí, jsou-li obě části daného úseku zpracovány. V jedné části se pak nacházejí prvky menší než zvolená hodnota pivota, v druhé části větší prvky.

V každé z obou částí se pak opakuje rozdělení na dva úseky a přeuspořádání hodnot v nich. Dělení (a tím i řazení) končí v okamžiku, kdy se dojde k jednoprvkovému úseku.

Výběr pivota určuje, jak bude přesně probíhat dělení na úseky. Je proto vhodné, pokud jsou obě části pole (přibližně) stejně velké. Ideálním pivotem by byl medián – na obou stranách by bylo zaručeno stejné množství prvků. Protože je ale zjišťování mediánu nepřijatelné, používá se buď nějaká fixní pozice (střed jako v dále uvedené variantě, nebo i první prvek, poslední prvek nebo úplně pseudonáhodná volba. Quick sort je také metoda použitelná spíše pro velká pole, která se dají dobře dělit na poloviny (ideální jsou mocniny dvou).

Při vhodné volbě pivota má metoda složitost $kN \log_2 N$, při nevhodné volbě se složitost blíží až ke kN^2 . To lze ovšem v praxi těžko docílit (pivot by musel být čirou náhodou vždy právě extrémem daného úseku).

Metoda má jednoduchý rekurzivní zápis. Při rekurzi se však zvyšuje prostorová složitost z lineární na lineárně logaritmickou. Rekurzivní algoritmus má následující tvar:

¹⁰ Sir Charles Antony Richard Hoare znám též jako Tony Hoare nebo C. A. R. Hoare (* 11. ledna 1934 Colombo) je britský počítačový vědec, který se proslavil zejména vyvinutím řadicího algoritmu zvaný Quick sort. V roce 1980 získal Turingovu cenu – obdoba Nobelovy ceny, ale v oblasti informatiky.

```

1200 void quick(Pole p, Indexy L, Indexy R){
1201     Indexy i=L, j=R; //algoritmus C. A. R. Hoare
1202     Slozka pivot=p[(L+R)/2]; //pivot je uprostřed
1203     do {
1204         while (p[i]<pivot) i++; //nevhodný prvek zleva
1205         while (p[j]>pivot) j--; //nevhodný prvek zprava
1206         if (i<=j) { //nalezena vhodná dvojice
1207             zamena(p, i, j);
1208             i++; j--; //příprava na další hledání
1209         }
1210     } while (i<=j); //dokud je kde hledat
1211     if (L<j) quick(p, L, j); //má-li levý úsek alespoň 2 prvky, zpracuj ho
1212     if (i<R) quick(p, i, R); //totéž pro pravý úsek
1213 }

```

15.4.3 Stromové řazení

Velmi jednoduchý zápis má algoritmus řazení touto strukturou. Předpokládejme typ dat, který lze jednoduše číst a porovnávat (například celá čísla), pak seřazení posloupnosti hodnot ze vstupu zapíšeme s využitím dříve definovaných datových typů a operací nad stromem:

```

1214 TypData vstup; UkUzel Strom;
1215 Init(Strom);
1216 while (cin>>vstup) BInsert(Strom, vstup);
1217 Inorder(Strom);

```

Jak efektivita vyhledávání, tak i řazení je ovšem závislá na skutečném počtu hladin stromu. Podíváme-li se na operaci vkládání, vidíme, že v sobě neobsahuje žádnou část, která by umisťovala uzly vyváženě. Tvar stromu závisí čistě na datech a jejich posloupnosti. Představíme-li si na vstupu uspořádaná data (například od nejmenší hodnoty po největší), dojde při vkládání do stromu k přidávání uzlů pouze doprava, takže místo stromu dostaneme jednu lineární větev, tj. strom kolabuje do tvaru ekvivalentu lineárního seznamu. V takové struktuře bude mít hledání lineární časovou složitost a řazení pak kvadratickou. Nejsou-li data zcela seřazena, ale existují v nich například určité seřazené posloupnosti, dostáváme nevyvážený strom, jehož vlastnosti se pohybují někde mezi ideální a lineární variantou. Tato silná závislost na datech je nebezpečná a v některých aplikacích znemožňuje efektivní nasazení této struktury.

15.4.4 Řazení slučováním

Další název: Merge sort

Hlavní vlastností této metody je sekvenčnost, která bývávala využívána při řazení velkého množství údajů uložených na magnetických páskách. Prostorová složitost metody je (z hlediska operační pa-

měti) zcela minimální: v paměti se uchovávají vždy nejvýše dvě řazené položky, mezi nimiž se určuje uspořádání. Metoda tedy měla svůj zásadní význam v době, kdy velikosti operačních pamětí počítačů byly podstatně menší než velikost řazených dat.

Metoda má dvě varianty. Základní variantou je práce se třemi proudy dat, dvěma vstupními a jedním výstupním, tzv. třípásková varianta.

Třípásková varianta pracuje v následujících krocích:

1. Data na pásce se rozdělí na poloviny na dvě pracovní pásy.
2. Dvě pracovní pásy se vezmou jako vstup.
3. Hodnoty, které jsou „na řadě“ z obou vstupních pásek, se porovnají a ve správném pořadí vypíší na jednu výstupní pásku. Ta hodnota, která byla použita do výstupu, se nahradí z příslušného souboru další čtenou hodnotou.
4. Postup se opakuje znovu od prvního bodu, ovšem při další manipulaci se berou v úvahu již seřazené úseky (po prvním kroku dvojice, dále čtveřice atd.).

Čtyřpásková varianta pracuje zcela stejně, pouze kroky slučování dat do jedné pásy a následné rozdělování na dvě výstupní pásy se spojují do jednoho – vstupem jsou dvě pásy, výstupem také dvě pásy. V cyklech se tedy zaměňují vstupní dvojice pásek za výstupní.

V dnešní době, kdy velikosti operačních pamětí umožňují většinou všechna řazená data umístit do paměti, se již uvedená varianta metody Merge sort s páskami nepoužívá. Řazení probíhá v poli vždy tak, že se slučují neklesající posloupnosti z obou zdrojových proudů dat, což dává metodě přirozenost (sloučení dvou již seřazených posloupností je provedeno v jednom kroku).

15.5 Lineární metody

Řadicí metody s lineární časovou složitostí nejsou univerzálně použitelné – za jejich teoreticky nepřekonatelně nízkou složitost platíme tím, že je můžeme použít jen v některých případech dat. Nejjednodušší je řazení množinou (nebo multimnožinou) – bylo již popsáno. Dále můžeme množinovou indexaci nahradit hašováním (řazení hašovací tabulkou), v tom případě je nezbytné ovšem mít speciální hešovací funkci, která zachovává uspořádání, a navíc skládat synonyma uspořádaně. Jako dalšího reprezentanta této kategorie můžeme uvést historickou metodu řazení podle základu (Radix sort).

16 Preprocesor, moduly

Podíváme se nyní poněkud podrobněji na proces překladu zdrojového textu. Obvykle je uživateli (programátorovi) tento proces zcela skryt: zavolá se překladač (ať už z příkazového řádku nebo tlačítkem z nějakého vývojového prostředí) a po nějaké chvíli se objeví hotový přeložený a spustitelný program (pokud ve zdrojovém textu pochopitelně nebyly nějaké chyby). Výsledný program však vzniká v několika *fázích* překladu, v každé z nich se řeší poněkud jiný problém. Aníž bychom se pouštěli do přílišných podrobností, je dobré vědět, že první fází je tzv. **preprocessing**, tj. příprava zdrojového textu programu pro hlavní překlad, pak následují obvyklé fáze analýzy (lexikální, syntaktická, sémantická), dále vytvoření vnitřního kódu přeloženého programu, případná optimalizace – odstranění všech nepotřebných částí, vygenerování výsledného kódu a jeho spojení s knihovními nebo vlastními moduly.

V tomto procesu nás budou zajímat dvě z uvedených fází: první z nich bude preprocessing, druhou pak možnost rozdělení programu na menší relativně samostatné celky – moduly – a jejich propojení do jednoho přeloženého díla.

16.1 Preprocesor

Preprocesor je část překladače, která čte zdrojový text programu, hledá v něm speciální konstrukce určené pro tuto fázi překladu a na základě těchto povelů upraví zdrojový text do podoby, v níž je předložen analytické fázi překladu. Je to stručně řečeno jakýsi automatizovaný editor, který upraví text napsaný programátorem do jiné podoby.

Povel pro preprocesor musí být dobře oddělen od běžných konstrukcí jazyka C/C++. Všechny tyto povely jsou zapsány tak, že hned v první pozici řádku se nachází znak `#` a za ním nějaké slovo (název povelu) a případné další parametry. V některé literatuře se preprocesorové povely nazývají direktivy překladu (direktivy překladače). Tento pojem budeme používat také.

Jednu z direktiv už z předchozích kapitol důvěrně známe – jde o vložení knihovny pomocí direktivy `#include`. Z toho vyplývá, že se bez direktiv prakticky neobejdeme, některé další mohou být velmi užitečné. V následujícím přehledu jsou uvedeny a vysvětleny ty nejčastěji používané.

- `#include` – jak již bylo uvedeno, jde o možnost připojit k danému zdrojovému textu jiný soubor. Jméno tohoto souboru se uvádí jako parametr direktivy a může být zapsáno v úhlových závorkách (platí pro systémové soubory) nebo v uvozovkách (platí pro uživatelské soubory). Překladač připojí daný soubor ke zdrojovému textu vlastního programu – složí více zdrojových souborů do jednoho celku.

Příklad:

```
1218 #include <iostream>
1219 //systémový soubor (knihovna)
1220 #include " /moduly/knihovna/mujsoubor.h"
1221 //vlastní soubor -- lze uvést i přístupovou cestu
```

- **#define** – direktiva umožňuje definovat tzv. **makro** (makropříkaz). Název makra a jeho hodnota jsou parametry direktivy. K čemu makro slouží? Představte si například, že chcete ve zdrojovém textu používat zápis **unsigned long int** na více místech. Abyste nemuseli vypisovat stále dokola tutéž typovou specifikaci, můžete definovat a použít makro:

```

1222 #define Mujtyp unsigned long int
1223 //příklady použití:
1224 void Cinnost(Mujtyp &X) ...
1225 Mujtyp A = 0, B;

```

Při každém použití makra udělá preprocesor *obyčejnou textovou náhradu* – za jméno makra nahradí text makra. Neprovádí se v této chvíli žádná kontrola, zda je použito ve správném místě a se správnou syntaxí okolí.

Víme, že něco podobného můžeme zařídit i pomocí **typedef**. Rozdíl ale spočívá především v tom, že použití makra je pouhá textová úprava zdrojového kódu, zatímco příkaz **typedef** je jazyková rekvizita, která se zpracovává až ve fázích překladových analýz, a proto se u ní kontroluje správnost použití.

- **#undef** – direktivu použijeme, chceme-li „oddefinovat“ (tj. zrušit definici) existujícího makra. Pokud navážeme na předchozí příklad a rozhodneme se, že již makro **Mujtyp** nepotřebujeme, nebo potřebujeme dále změnit jeho definici, zapíšeme **#undef Mujtyp**.
- Direktivy umožňující tzv. **podmíněný překlad** – preprocesor umožňuje některé části zdrojového textu při překladu vynechat, taková část se nazývá „podmíněně překládaná“. Stanovení podmínky, kdy se něco má překládat nebo vynechat, a vymezení odpovídajících částí textu umožňují následující direktivy:
 - **#if** – jako parametr direktivy je logický výraz složený z konstant nebo jiných maker. Pokud je tento logický výraz platný, následující úsek zdrojového textu až po direktivu **#else**, **#elif** nebo **#endif** se *bude* překládat.
 - **#else** – vymezuje začátek zdrojového textu, který se bude překládat jen v případě *neplatné podmínky* předchozí direktivy **#if**.
 - **#endif** – vymezuje konec podmíněného překladu direktivy **#if**.
 - **#elif** – zkratka za „**else if**“.
 - **#ifdef** – parametrem direktivy je jméno makra. Pokud je toto makro již definováno, jedná se o platnou podmínku.
 - **#ifndef** – parametrem direktivy je jméno makra. Pokud toto makro není definováno, jedná se o platnou podmínku.
- **#line** – direktiva definuje číslování řádků zdrojového textu v uvedeném souboru (zapisuje se například **#line 10000 "modul.h"**). Při výpisu chybových hlášení, kde se uvádí číslo řádku s chybou, se pak lze snadno orientovat podle zadaných čísel, ve kterém souboru chyba je.
- **#error** – direktiva vynutí chybové hlášení. Je pochopitelně spojena s podmíněným překladem a indikuje situaci, kdy překlad pravděpodobně neprobíhá tak, jak bychom si představovali...

- `#pragma` – direktiva umožňuje předat nějaké informace překladači. Jde o záležitost spojenou s konkrétními možnostmi použitého překladače.

16.2 Programové moduly

Ani v případě jednoduchých programů programátor nepíše úplně celý program, který se následně spouští a zpracovává data. Představte si například situaci, kdy potřebujeme na terminál vypsat nějaký řetězec:

```
1226 int main(){
1227     cout << "Pozdrav běžícího programu!" << std::endl;
1228     return 0;
1229 }
```

Víme dobře, že se takový program nedá přeložit – důvodem je, že proud `cout` není v tomto zápisu definován a jeho definice není vůbec jednoduchá. Nicméně vypsat něco na terminál potřebuje skoro každý program a bylo by velmi nepohodlné, kdyby programátor musel všechno v každém programu neustále psát znovu a znovu. Proto existuje rozsáhlá množina již hotových, odladěných a optimalizovaných rekvizit, které lze použít jednoduše tím, že je do potřebného programu vložíme direktivou `#include`.

Avšak princip skládání programu z více dílčích částí není omezen jen na obecně použitelné knihovny. Větší program je prakticky vždy nezbytné rozdělit na menší celky, které se snadněji zkonstruují a odladí. Dostáváme se tím k pojmu **programový modul**. Důležitou vlastností takového modulu je relativní samostatnost – lze jej přeložit a použít v libovolném jiném celku. Modul vždy představuje nějakou logicky ucelenou část programu nebo přináší prvky, které spolu logicky souvisejí. Princip modulů je implementován v mnoha jazycích, neboť se jedná o nástroj umožňující aplikovat metodu programování shora dolů (a při jejich využívání i zdola nahoru).

Jak vypadá implementace modulu v jazyce C++?

Modul je složen ze dvou částí: v jedné se nacházejí definice všech prvků, které daný modul nabízí jako prvky „veřejné“ – použitelné zvenku (nejčastěji datové typy a hlavičky podprogramů), ve druhé jsou pak „soukromé“ části neviditelné zvenku a těla podprogramů. Každá z obou částí je uložena v samostatném souboru – veřejná část je v tzv. **hlavičkovém souboru** (má rozšíření `.h`) a soukromá část v implementačním souboru s obvyklým rozšířením `.cpp`.

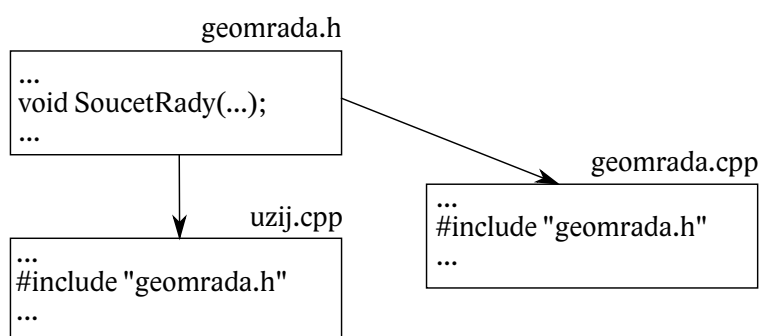
Ukažme si definici a použití modulu na tomto příkladu: Chceme vytvořit jednoduchý modul, jehož součástí je funkce pro výpočet součtu geometrické řady s určitou přesností. Funkce pro svou činnost potřebuje dva parametry charakterizující zadanou řadu: počáteční člen a kvocient. Pro řízení výpočtu se ještě použije přesnost – ta bude v modulu nachystána jako proměnná, do které se v případě potřeby vždy zadá taková přesnost výpočtu, jaká je v daném okamžiku vyžadována.

Jak jsme už uvedli, modul je složen ze dvou částí (každá je v samostatném souboru) – jedna část obsahuje prvky viditelné zvnějšku (veřejná část) a je v hlavičkovém souboru s rozšířením `.h` – zvolíme jméno `geomrada.h`, druhá část doplňuje implementaci a je v souboru s rozšířením `.cpp` – v našem

případě `geomrada.cpp`. Modul budeme chtít použít v nějakém programu, zapíšeme ho do souboru `uzij.cpp`. Samotný překlad obvykle probíhá tak, že všechny soubory umístíme do jednoho adresáře a zavoláme překladač na všechny soubory daného adresáře, například:

```
1230 | g++ *.cpp *.h -o vysled
```

Při překladu se překladač musí v těch souborech nějak vyznat – musíme mu sdělit, co je kdy potřeba. To uděláme pomocí direktivy `#include`. Překladač vytvoří přeložené segmenty, které pak musí linker vhodně pospojovat dohromady tak, aby všechny odkazy na všechny prvky byly řádně uspokojeny. Je v tom ještě jeden drobný háček: hlavičkový soubor nutně musíme vložit do dvou míst: do implementační části modulu a také do hlavního programu. Schematicky celou situaci znázorňuje obr. 25.



Obrázek 25: Vkládání souborů modulu a hlavního programu

Překladač pak bude muset hlavičkový soubor překládat nejméně dvakrát – ale to není žádoucí: myslel by si, že definice (proměnné, typy, hlavičky podprogramů) jsme omylem zapsali dvakrát. Celý obsah hlavičkového souboru v tom případě musíme „obrnit“ vůči vícenásobnému překladu, a to podmíněným překladem.

Chceme zajistit, aby se po prvním překladu už podruhé soubor nemohl překládat. Při prvním překladu provedeme definici nějakého prázdného makra, pokud tato definice již proběhla, poznáme, že překládáme podruhé. Celý obsah hlavičkového souboru „zabalíme“ do následujících direktiv:

```
1231 | #ifndef nejake_makro
1232 | #define nejake_makro
1233 | ... celý obsah souboru .h
1234 | #endif
```

Teď se ještě zamyslíme nad názvem toho fiktivního makra. Představme si, že bychom potřebovali takto ošetřit více modulů. Jaký název zvolit, aby se nikdy nestalo, že jsme použili nějaké makro dvakrát? Jak si jednoduše zapamatovat, které makro jsme v kterém modulu použili? K řešení nám poslouží úplně jednoduchý a funkční mechanismus: název makra odvodíme z názvu hlavičkového souboru. Ten musí být pro každý hlavičkový soubor jiný – nehrozí duplicita. Obvykle se volí název makra tak, že jméno hlavičkového souboru se zapíše velkými písmeny a místo tečky oddělující rozšíření `.h` se použije podtržítko. Konkrétně v našem případě:

```
1235 #ifndef GEOMRADA_H
1236 #define GEOMRADA_H
1237 ... celý zbývající obsah souboru geomrada.h
1238 #endif
```

Nyní se už můžeme zabývat obsahem všech uvedených souborů. Nejprve doplníme hlavičkový soubor:

```
1239 #ifndef GEOMRADA_H
1240 #define GEOMRADA_H
1241 extern float Presnost; //proměnná pro uložení přesnosti výpočtu
1242 float SoucetRady(float a, float q); //hlavička funkce
1243 #endif
```

Klíčové slovo **extern** říká, že někde bude použita proměnná **Presnost** a na toto použití se musí udělat vazba. Ta proměnná bude potřeba jak v implementační části modulu, tak i v hlavním programu.

Implementace spočívá v doplnění těla funkce **SoucetRady**. Funkce může pracovat jen tehdy, pokud součet řady konverguje (tj. členy se postupně přibližují nule). Tuto skutečnost ověříme tím, že zkontrolujeme hodnotu kvocientu – aby se členy zmenšovaly, musí být kvocient v absolutní hodnotě menší než jedna. Pokud řada nekonverguje k nule, vydáme jako hodnotu součtu náhradní číslo nula. Je-li vše v pořádku, funkce vypočítává postupně další a další členy geometrické řady a sčítá je. Tento proces probíhá tak dlouho, dokud jsou vypočítávané členy větší než zadaná přesnost. K výpočtu potřebujeme funkci pro absolutní hodnotu – tuto funkci bychom mohli vzít z nějaké jiné knihovny, ale v této ukázce ji raději doplníme před kód funkce **SoucetRady**. Ukážeme na ní, jak se používají soukromé součástky – funkce je dostupná pouze v implementaci modulu a není vidět zvnějšku. Celý kód souboru **geomrada.cpp** může mít následující tvar:

```
1244 #include "geomrada.h"
1245
1246 float abs(float x){
1247     if (x<0) return -x;
1248     else return x;
1249 }
1250
1251 float SoucetRady(float a, float q){
1252     if (abs(q)<1) { // řada konverguje -> existuje součet
1253         float Soucet=a;
1254         while (abs(a)>Presnost){
1255             a *= q;
1256             Soucet += a;
1257         }
1258         return Soucet;
1259     }
```

```
1259 | } else return 0; // součet neexistuje
1260 | }
```

Program, který využívá definovaný modul, pouze přečte potřebné hodnoty, nastaví přesnost na $1 \cdot 10^{-9}$ a vypíše výsledek:

```
1261 | #include <iostream>
1262 | #include "geomrada.h"
1263 | using namespace std;
1264 |
1265 | int main(){
1266 |     float x, y;
1267 |     cin >> x >> y;
1268 |     Presnost = 1e-9;
1269 |     cout << "Součet řady je "<< SoucetRady(x, y) << endl;
1270 |     return 0;
1271 | }
```

17 Struktury s obecnými daty

Operace se strukturami potřebujeme ve stejné podobě pro různé typy dat. Je velmi neefektivní pro každou aplikaci přepisovat definice odladěných funkcí a procedur jen proto, že musíme měnit data a manipulaci s nimi. Tento problém se řeší několika různými způsoby podpořenými různými syntaktickými nástroji, pro nejjednodušší vysvětlení principu využijeme reprezentaci dat ve formě obecného ukazatele. Mějme například strom, který chceme využít pro řazení nějakých dat (pokaždé jiných). V definici uzlu stromu zrušíme závislost na existenci určitého datového typu `TypData` tím, že jako datovou složku použijeme obecný ukazatel:

```
1272 struct TypUzel { //uzel binárního stromu
1273     void *Data;
1274     TypUzel *Vlevo, *Vpravo;
1275 };
1276 typedef TypUzel *UkUzel; //ukazatel na uzel
```

Operace nad takto definovaným stromem budou muset být ovšem také změněny. Zatímco všechny manipulace se stromem jako takovým zůstávají stejné, všude tam, kde se odvoláváme na hodnotu dat (zjišťujeme menší hodnotu, zjišťujeme shodu hodnot, vypisujeme hodnotu na výstup), musíme použít podobné zobecnění, jehož konkrétní tvar budeme schopni dosadit až v okamžiku, kdy potřebujeme aplikovat strukturu na daná data. Nástrojem umožňujícím toto zobecnění je datový typ ukazatele na podprogram.

17.1 Datový typ podprogram

Podle principu von Neumannovy konstrukce počítače platí, že data i instrukce jsou umístěny ve stejné operační paměti. Ukazatel na data jako reprezentace adresy dat má v tom případě úplně stejnou podobu jako adresa jakékoliv instrukce, můžeme pak úplně stejně mít k dispozici například adresu nějakého podprogramu (přesněji řečeno jeho počáteční instrukce, tzv. vstupního bodu).

Stejně jako máme určité ukazatele (ukazatele na určité datové typy), lze definovat ukazatele na podprogramy. Definice vypadá takto:

```
1277 typedef vystup_typ (*TPodprogram)(parametry);
1278 //například:
1279 typedef float (*TypRealneFunkce)(float X);
```

Vznikne nový identifikátor typu `TPodprogram` představující adresu podprogramu, jehož hlavička odpovídá předpisu z definice: podprogram musí mít odpovídající výstupní typ a musí mít odpovídající parametry. Všimněte si kulatých závorek `(*TypRealneFunkce)` – ty jsou velmi důležité a říkají, že definujeme datový typ *ukazatele na funkci*. Kdybychom je neuvedli, pak by šlo o funkci, jejímž výstupním typem bude ukazatel na `float`, což je něco zcela jiného.

Použijeme-li uvedený příklad reálné funkce jedné reálné proměnné, můžeme psát:

```

1280 TypRealneFunkce Povrch; //deklarace proměnné typu reálná funkce
1281 float MujPovrch(float R){
1282     return 4*R*R*M_PI;
1283 }
1284 Povrch = MujPovrch; //přiřazení adresy funkce do proměnné
1285 float vstup;
1286 cin >> vstup;
1287 cout << "Povrch daného tělesa je " << Povrch(vstup) << endl;
1288 cout << "Adresa funkce je " << Povrch << endl;

```

Na řádcích 1287 a 1288 vidíme syntaktický i sémantický rozdíl v použití: jsou-li uvedeny kulaté závorky, jde o *volání podprogramu*, pokud závorky neuvedeme, jde o *adresu podprogramu*.

Proměnná `Povrch` může pokaždé nabývat jiné adresy, přičemž část kódu, kde se používá (řádek 1287), je stále stejná. Tato část kódu ale vlivem výměny adres podprogramů může pokaždé provádět něco poněkud jiného – jednou to bude povrch koule, jindy povrch válce s výškou 1, někdy jen plocha podstavy nebo povrch krychle. Princip datového typu podprogram je implementačním základem virtuálních metod v objektovém programování: virtuální metodu si lze představit jako složku objektu v podobě adresy podprogramu. Přiřazením jiného objektu se do této složky přiřadí adresa jiné metody a v této chvíli se začne objekt chovat jinak – jde o polymorfní objekty. Souvislost, která je zde ukázána, pouze dokládá, že princip obecných datových struktur lze chápat i jako princip polymorfismu.

17.2 Modifikace operací vyžadujících znalost dat

Vraťme se nyní opět k operacím nad strukturou s obecnými daty. Na příkladu binárního vyhledávacího stromu si ukážeme, jak operace zobecnit pomocí ukazatelů na vhodné podprogramy.

Kde potřebujeme manipulovat s daty? Při vkládání potřebujeme zjistit, zda je vkládaná hodnota *menší než* uložená v uzlu; při vyhledávání potřebujeme zjistit, zda se *hledaná data rovnají* hodnotě v uzlu; při průchodu stromem potřebujeme *vhodně zpracovat* data v daném uzlu.

Představme si, že pro různá porovnání bude existovat funkce se dvěma údaji v parametrech, jejímž výsledkem bude hodnota `-1`, pokud první údaj bude menší než druhý údaj, hodnota `0`, budou-li údaje shodné, a hodnota `1`, bude-li první údaj větší než druhý. Takovou funkci můžeme aplikovat v prvních dvou případech (zjištění menšího údaje, zjištění shodnosti údajů). Kde tuto funkci získáme? Vzhledem k tomu, že tato funkce musí být pro daná data stále stejná (jinak se strom nebude chovat, jak očekáváme), můžeme ji připojit jako součást stromu při inicializaci. Celý strom pak nebudeme chápat jen jako ukazatel na kořen, ale jako dvě složky: ukazatel na kořen a porovnávací funkci:

```

1289 typedef signed char (*TypPorov)(void *X, void *Y); //typ funkce
1290 struct TypStrom{
1291     UkUzel Koren; //reprezentace stromu dvěma složkami

```

```

1292     TypPorov Porov;
1293 };

```

Třetím případem manipulace s daty je zpracování při průchodu **Inorder**. Zde ovšem můžeme jedna a tatáž data zpracovávat různými způsoby, operaci **Inorder** můžeme zavolat několikrát a pokaždé můžeme požadovat něco jiného. Proto je optimální, pokud se zpracovávající procedura dodá jako parametr procedury **Inorder**.

Operace nad stromem přepíšeme do následujícího tvaru:

```

1294 void Init(TypStrom &S, TypPorov R){ //inicializace
1295     S.Koren = NULL; //prázdný strom
1296     S.Porov = R; //přiřazení odpovídajícího porovnání
1297 }
1298 void BInsertVnitrek(UkUzel &S, TypData D, TypPorov R){
1299     //rekurzivní vnitřek vkládání zůstává podobný, změna v porovnání
1300     if (S==NULL){ //prázdný strom nebo místo pro připojení listu
1301         S = new TypUzel; //vytvoření nového uzlu + vazba na existující uzel
1302         S->Data = D; //nový uzel je listem
1303         S->Vlevo = NULL;
1304         S->Vpravo = NULL;
1305     } else
1306         if (R(D,S->Data)<1)
1307             //vkládaná hodnota je menší nebo rovna, jdeme rekurzivně vlevo
1308             BInsertVnitrek(S->Vlevo, D, R);
1309         else BInsertVnitrek(S->Vpravo, D, R); //jinak rekurzivně vpravo
1310 }
1311 void BInsert(TypStrom &S){ //nerekurzivní obálka
1312     BInsertVnitrek(S.Koren, S.Porov);
1313 }
1314
1315 bool BSearch(TypStrom S, TypData D){ //zjištění přítomnosti D ve stromu
1316     UkUzel Pom = S.Koren; //pomocný ukazatel pro vyhledávání, začínáme v kořeni
1317     while (Pom!=NULL and S.Porov(D,Pom->Data)!=0)
1318         //je kde hledat a nebylo nalezeno
1319         if (S.Porov(D,Pom->Data)<1) Pom=Pom->Vlevo;
1320         else Pom=Pom->Vpravo; //jdeme do nižší hladiny
1321     return Pom!=NULL; //výsledek hledání
1322 }
1323 typedef void (*TypZpracuj)(void *A);
1324 void InorderVnitrek(UkUzel S, TypZpracuj Z){ //rekurzivní vnitřek
1325     if (S!=NULL){ //existující uzel -> zpracujeme
1326         InorderVnitrek(S->Vlevo, Z); //napřed levý podstrom
1327         Z(S->Data); //zpracování dat kořene

```

```

1328     InorderVnitrek(S->Vpravo, Z); //nakonec pravý podstrom
1329 }
1330 }
1331 void Inorder(TypStrom S, TypZpracuj Zpracuj){
1332     InorderVnitrek(S.Koren, Zpracuj);
1333 }

```

Všimněte si, že v celé implementaci operací se vůbec neřeší, jaká budou konkrétní těla podprogramů **Porov** nebo **Zpracuj**. To je v této chvíli překladači jedno, protože všechny potřebné informace pro to, aby kód úspěšně přeložil, má k dispozici. Ví, jak vypadají hlavičky, je schopen případně zkontrolovat odpovídající podprogram, který tam vejde jako skutečný parametr, takže se nemůže stát, že by něco někde zhavarovalo. Tímto postupem jsme zcela oddělili implementaci stromu od jeho použití. Je vhodné takovou implementaci umístit do samostatného modulu, aby se v případě použití nemusel nikam kopírovat tento zdrojový kód, ale jen se vložil hlavičkový soubor. Pro implementaci modulu bude potřebné provést ještě drobné editační úpravy, důsledně rozdělit veřejné a soukromé prvky a vytvořit hlavičkový a implementační soubor. Modul nazveme BVS a ve výsledku může mít následující tvar (hlavičkový soubor):

```

1334 #ifndef BVS_H
1335 #define BVS_H
1336 using namespace std;
1337
1338 typedef signed char (*TypPorov)(void *X, void *Y);
1339 typedef void (*TypZpracuj)(void *A);
1340
1341 struct TypUzel { //uzel binárního stromu
1342     void *Data;
1343     TypUzel *Vlevo, *Vpravo;
1344 };
1345 typedef TypUzel *UkUzel; //ukazatel na uzel
1346 struct TypStrom{
1347     UkUzel Koren; //reprezentace stromu dvěma složkami
1348     TypPorov Porov;
1349 };
1350
1351 void Init(TypStrom &S, TypPorov R);
1352 void BInsert(TypStrom &S);
1353 bool BSearch(TypStrom S, TypData D);
1354 void Inorder(TypStrom S, TypZpracuj Zpracuj);
1355 #endif

```

Implementační soubor:

```
1356 #include "BVS.h"
1357 #include <iostream>
1358 using namespace std;
1359
1360 void Init(TypStrom &S, TypPorov R){ //inicializace
1361     S.Koren = NULL; //prázdný strom
1362     S.Porov = R; //přiřazení odpovídajícího porovnání
1363 }
1364 void BInsertVnitrek(UkUzel &S, TypData D, TypPorov R){
1365     //rekurzivní vnitřek vkládání zůstává podobný, změna v porovnání
1366     if (S==NULL){ //prázdný strom nebo místo pro připojení listu
1367         S = new TypUzel; //vytvoření nového uzlu + vazba na existující uzel
1368         S->Data = D; //nový uzel je listem
1369         S->Vlevo = NULL;
1370         S->Vpravo = NULL;
1371     } else
1372         if (R(D,S->Data)<1)
1373             //vkládaná hodnota je menší nebo rovna, jdeme rekurzivně vlevo
1374             BInsertVnitrek(S->Vlevo, D, R);
1375         else BInsertVnitrek(S->Vpravo, D, R); //jinak rekurzivně vpravo
1376 }
1377 void BInsert(TypStrom &S){ //nerekurzivní obálka
1378     BInsertVnitrek(S.Koren, S.Porov);
1379 }
1380
1381 bool BSearch(TypStrom S, TypData D){ //zjištění přítomnosti D ve stromu
1382     UkUzel Pom = S.Koren; //pomocný ukazatel pro vyhledávání, začínáme v kořeni
1383     while (Pom!=NULL and S.Porov(D,Pom->Data)!=0)
1384         //je kde hledat a nebylo nalezeno
1385         if (S.Porov(D,Pom->Data)<1) Pom=Pom->Vlevo;
1386         else Pom=Pom->Vpravo; //jdeme do nižší hladiny
1387     return Pom!=NULL; //výsledek hledání
1388 }
1389 void InorderVnitrek(UkUzel S, TypZpracuj Z){ //rekurzivní vnitřek
1390     if (S!=NULL){ //existující uzel -> zpracujeme
1391         InorderVnitrek(S->Vlevo, Z); //napřed levý podstrom
1392         Z(S->Data); //zpracování dat kořene
1393         InorderVnitrek(S->Vpravo, Z); //nakonec pravý podstrom
1394     }
1395 }
1396 void Inorder(TypStrom S, TypZpracuj Zpracuj){
1397     InorderVnitrek(S.Koren, Zpracuj);
1398 }
```

17.3 Použití obecné datové struktury

Podíváme se nyní na program, který obecnou strukturu využije pro konkrétní data. Předpokládejme, že na vstupu jsou řetězce, které chceme seřadit. Pak budeme muset vytvořit příslušnou porovnávací funkci, která obecné ukazatele přetypuje na ukazatele na řetězce a pak je porovná, stejně tak potřebujeme proceduru, která bude umět zpracovat obecné ukazatele přetypováním na ukazatele na řetězce. Oba podprogramy pak použijeme v příslušných operacích nad modifikovaným stromem:

```

1399 //vlození implementace stromu ve formě modulu
1400 #include "BVS.h"
1401 #include <iostream>
1402 using namespace std;
1403
1404 signed char PorovRetez(void *A, void *B){
1405     string *R1 = (string*)A, *R2 = (string*)B;
1406     if (*R1 < *R2) return -1;
1407     else if (*R1 == *R2) return 0;
1408     else return 1;
1409 }
1410 void VypisRet(void *A){
1411     string *R = (string*)A;
1412     cout << R << endl;
1413 }
1414 int main(){
1415     string *vstup, R;
1416     TypStrom Strom;
1417     Init(Strom, PorovRetez);
1418     while (cin>>R) {
1419         vstup = new string;
1420         *vstup = R;
1421         BInsert(Strom, vstup);
1422     }
1423     Inorder(Strom, VypisRet);
1424     return 0;
1425 }
```

Podobně bychom postupovali v případě jiných struktur. Na příkladu s binárním vyhledávacím stromem jsme měli možnost vidět všechny prvky, které se v této souvislosti dají použít, u některých jiných struktur může být situace někdy jednodušší (nejsou například potřeba nerekurzivní obálky rekurzivních operací, nejsou potřeba varianty vkládání vnějších podprogramů v různých parametrech apod.).

18 Dodatky

18.1 Tabulka priorit operátorů jazyka C++

Priorita	Operátor	Popis	Směr vyhodnocení
1	::	rozlišení prostoru	zleva doprava
2	a++ a-- typ(), typ{ } a() a[] . ->	postinkrement, postdekrement funkcionální přetypování volání funkce indexace přístup ke složce	zleva doprava zleva doprava
3	++a --a +a -a ! ~ (typ) *a &a sizeof new new[] delete delete[]	preinkrement, predekrement unární plus, unární minus logická negace, bitová negace přetypování ve stylu jazyka C dereference získání adresy paměťová velikost alokace paměti uvolnění paměti	zprava doleva
4	.* ->*	ukazatel na složku	zleva doprava
5	a*b a/b a%b	násobení, dělení, zbytek	zleva doprava
6	a+b a-b	sčítání, odčítání	zleva doprava
7	<< >>	bitový posuv	zleva doprava
8	<=>	třícestné porovnání (od C++20)	zleva doprava
9	< <= > >=	porovnání	zleva doprava
10	== !=	rovnost/nerovnost	zleva doprava
11	&	bitový součin	zleva doprava
12	^	bitový výhradní součet	zleva doprava
13		bitový součet	zleva doprava
14	&& and	logický součin	zleva doprava
15	or	logický součet	zleva doprava
16	a?b:c throw = += -= *= /= %= <<= >>= &= ^= =	ternární operátor operátor throw přiřazení složené přiřazení složené přiřazení	zprava doleva
17	,	operátor čárka	zleva doprava

18.2 Vybraná klíčová slova jazyka C++

V následujícím přehledu se můžeme zorientovat v rezervovaných slovech, která nemůžeme použít jako identifikátory vlastních prvků. Množství klíčových slov je bohužel závislé na verzi jazyka, klíčová slova stále přibývají, takže se může stát, že dojde ke kolizi se zvoleným identifikátorem.

and	asm	auto	bitand	bitor
bool	break	case	catch	char
class	const	continue	default	delete
do	double	else	enum	explicit
extern	false	float	for	friend
goto	if	inline	int	long
mutable	namespace	new	not	operator
or	private	protected	public	register
return	short	signed	sizeof	static
struct	switch	template	this	throw
true	try	typedef	typeid	typename
union	unsigned	using	virtual	void
volatile	while	xor		

18.3 Možnosti formátování vstupu a výstupu

V předchozím textu už jsme se na různých místech o některých vlastnostech vstupů a výstupů zmínili, v této části vše soustředíme na jedno místo a doplníme o některé další možnosti.

18.3.1 Ovlivnění vstupu

Objekt `cin` s operátorem `>>` představuje jednoduchý vstup, který vynechává tzv. bílé znaky (meze-ry, tabulátory, konce řádků). Tato metoda je užitečná pro vstup čísel, izolovaných znaků a některých znakových řetězců. Často však potřebujeme pracovat se vstupními hodnotami poněkud odlišně. Musíme tedy využít další nástroje.

Pro vstup jednotlivých znaků bez přeskakování bílých míst můžeme užít metodu `get` vstupního proudu `cin`. Příklad:

```

1426 |   char ch;
1427 |   cin.get(ch);

```

Změnu chování vstupních operací lze změnit metodami `setf` nebo `unsetf`. Tyto metody jsou schopny zapnout nebo vypnout požadovanou vlastnost. Vlastnosti jsou pojmenovány identifikátory, které se nacházejí ve jmenném prostoru `ios`. Některé z nich platí i pro výstup. Příklady:

<code>skipws</code>	vstup přeskakuje/nepřeskakuje bílé znaky
<code>boolalpha</code>	vstup čte a výstup vypisuje logické hodnoty jako řetězce
<code>showbase</code>	výstup u celých čísel vypisuje prefix zobrazující číselnou soustavu
<code>showpoint</code>	výstup u desetinných čísel vynucuje výpis desetinné tečky

Příklad:

```
1428 | cin.unsetf(ios::skipws); //vypnutí přeskakování bílých znaků
1429 | cin >> ch; //přečte i bílý znak
```

Pro čtení řetězců znaků (znakových polí, nikoliv proměnných typu `string`) je k dispozici metoda `getline`. Metoda má dva nebo tři parametry:

```
1430 | char S[20];
1431 | cin.getline(S, 10); // čte max. 9 znaků plus znak \0 nebo do konce řádku
```

Čtení můžeme ovlivnit zadáním oddělovače:

```
1432 | cin.getline(S, 15, ':');
1433 | //čte max. 15 znaků nebo po dvojtečku nebo do konce řádku
```

Poslední varianta je velmi užitečná například pro čtení hodnot ze souborů formátu CSV.

Pro čtení řetězců typu `string` je k dispozici procedura `getline` (nikoliv metoda objektu `cin`).

```
1434 | string S;
1435 | getline(cin, S); //čte řetězec do konce řádku
1436 | getline(cin, S, ':'); //čte řetězec po dosažení znaku dvojtečka
```

18.3.2 Ovlivnění výstupu

Pro formátování výstupu ve výstupním proudu jsou k dispozici tzv. manipulátory. Manipulátory jsou zapisovány stejně jako výstupní proměnné nebo výrazy v objektu `cout`. Následující tabulka přináší některé užitečné manipulátory:

<code>endl</code>	vloží do výstupu konec řádku
<code>dec</code>	výstup čísel bude v desítkové soustavě
<code>oct</code>	výstup čísel bude v osmičkové soustavě
<code>hex</code>	výstup čísel bude v šestnáctkové soustavě
<code>showpos</code>	bude se zobrazovat znaménko plus u kladných čísel
<code>noshowpos</code>	nebude se zobrazovat znaménko plus u kladných čísel
<code>left</code>	hodnota bude ve výstupu zarovnána doleva
<code>right</code>	hodnota bude ve výstupu zarovnána doprava
<code>internal</code>	výplňový znak bude zobrazován mezi znaménko a hodnotu
<code>fixed</code>	zobrazí desetinná čísla s pevnou pozicí desetinné tečky
<code>scientific</code>	zobrazí desetinná čísla ve vědeckém tvaru (s exponentem)
<code>uppercase</code>	všechna písmena (hex, scientific) budou velká
<code>nouppercase</code>	všechna písmena (hex, scientific) budou malá
<code>setprecision(p)</code>	nastaví přesnost desetinných čísel na <code>p</code> číslic
<code>setw(x)</code>	nastaví šířku výstupu na <code>x</code> znaků
<code>setfill(c)</code>	nastaví výplňový znak na <code>c</code> místo mezery

Manipulátory `setw`, `setprecision` a `setfill` jsou obsaženy v knihovně `iomanip`.

Některé příklady:

```

1437 cout << showpos << 56; //zobrazí +56
1438 cout << setw(5)<<left<<-56; //zobrazí dvě mezery a -56
1439 cout << fixed << setprecision(3) << 3.141592; //zobrazí 3.142
1440 cout << scientific << setprecision(2) << 142.0; //zobrazí 1.42e+02
1441 cout << setw(10) << left << setfill('/') << fixed
1442 << setprecision(3) << 3.115926; //zobrazí 3.142/////

```

Nastavení platí pro všechny následující výstupy, dokud neprovedeme změnu. Výjimkou je nastavení manipulátorem `setw`, které platí pouze pro první následující výraz.

18.4 Zkrácené vyhodnocování logických výrazů

Z důvodu optimalizace výsledného kódu překladač obvykle generuje instrukce tak, aby se nedělaly zbytečné operace. Proto se při vyhodnocování složených logických výrazů některé části již nepočítají, pokud je celý výsledek výrazu již znám z dosud vyhodnocené části.

Jde o tyto optimalizace:

- Ve výrazu `A and B` se nevyhodnocuje část `B`, pokud je část `A` již vypočtena s hodnotou `false`. Ať je hodnota podvýrazu `B` jakákoliv, je výsledná hodnota celého výrazu `false`.
- Ve výrazu `A or B` se nevyhodnocuje část `B`, pokud je část `A` již vypočtena s hodnotou `true`. Ať je hodnota podvýrazu `B` jakákoliv, je výsledná hodnota celého výrazu `true`.

V mnoha případech na tom nezáleží a je vzhledem ke komutativitě obou operátorů **and** a **or** úplně jedno, zda napíšeme logický výraz ve tvaru **A and B** nebo **B and A**. Mějme například logický výraz:

```
1443 | N >= 10 and N <= 100
```

Je-li hodnota **N** menší než 10, druhá část se vůbec nevyhodnotí, výsledkem bude **false** a nikdy nedojde k nějakému problému.

Jiný příklad: mějme logický výraz použitý například v podmínce cyklu:

```
1444 | int a; unsigned int pocet=0;
1445 | cin >> a;
1446 | while (a % 2 == 0 and cin >> a) pocet++;
```

Záměrem autora bylo zjistit počet sudých vstupních hodnot. Ovšem vzhledem k tomu, že čtení probíhá jen tehdy, je-li první část podmínky splněna, získáním prvního lichého čísla ze vstupu se program ukončí, neboť **a % 2 == 0** neplatí, celá podmínka rovněž neplatí a cyklus skončí. Záludnost této chyby spočívá i v tom, že se nikde neohlásí žádná chyba a program někdy pracuje správně a někdy ne (záleží na datech).

Další příklad: využijeme předchozí deklarace a přidáme funkci **JeTreti**. Tu využijeme k počítání vstupních hodnot – chceme zjistit, kolik vstupních hodnot je sudých nebo dělitelných třemi:

```
1447 | bool JeTreti(int &Cislo){
1448 |     bool splnuje = Cislo % 3 == 0;
1449 |     cin >> Cislo;
1450 |     return splnuje;
1451 | }
1452 | ...
1453 | while (a % == 0 or JeTreti(a)) pocet++;
```

Tentokrát je funkce **JeTreti** volána jen tehdy, pokud hodnota **a** je lichá. Získáním první sudé hodnoty dojde k zacyklení, protože hodnota proměnné **a** nebude naplněna ze vstupu, podmínka bude neustále platit a cyklus nikdy neskončí.

Můžete namítnout, že funkce **JeTreti** dělá ošklivý vedlejší efekt, když současně s vyhodnocováním aktuální hodnoty ještě čte ze standardního vstupu další hodnotu. Co se týče samotného vedlejšího efektu, není to zde na první pohled viditelné, protože jsme uváděli, že standardní soubory se v parametrech nepředávají a jejich obsluha se často nepovažuje za vedlejší efekt. Tou zásadní logickou chybou je ale spojení dvou téměř nesouvisejících činností do jednoho podprogramu (test nějaké podmínky a čtení další hodnoty), což může vést právě k nepřijatelným chybám.

Po dvou záludných situacích uvedeme ještě jednu, kde se naopak zkrácené vyhodnocování logických výrazů velmi hodí, dokonce na ně spoléháme, a můžeme tak zápis zkrátit. Jde například o sekvenční hledání v poli – hledáme hodnotu **Hledana** v poli **Pole**, v němž je tolik prvků, kolik udává hodnota proměnné **Obsazeno**:

```
1454 | Index=0;  
1455 | while (Index<Obsazeno and Pole[Index]!=Hledana) Index++;
```

Nebýt zkráceného vyhodnocování logických výrazů, mohlo by dojít k chybě indexace. V první části podmínky se zjistí, zda je index v pořádku. Pokud ano, pak se teprve použije k indexaci pole ve druhé části podmínky. Pokud ne, k indexaci vůbec nedojde, nemůžeme se tedy odkázat na chybnou položku pole.

Jak lze zkrácené vyhodnocování v nepříznivých případech obejít? Z diskutovaných příkladů vyplývá, že musíme dávat pozor na volání podprogramů v podmínkách – některý podprogram nesmí být volán v druhé části, kde hrozí její přeskočení. Protože operátory logického součtu i součinu jsou komutativní, většinou stačí přehodit pořadí operandů. Pokud s touto úpravou nevystačíme, pak další možností je podmínku rozdělit a každou část uvést zvlášť, aby se zkrácené vyhodnocení neuplatnilo:

```
1456 | if (A and B) S;  
1457 |     lze nahradit za:  
1458 | if (A)  
1459 |     if (B) S;  
1460 |  
1461 | if (A or B) S;  
1462 |     lze nahradit za:  
1463 | if (A) S;  
1464 | if (B) S;  
1465 |  
1466 | while (A and B) S;  
1467 |     lze nahradit za:  
1468 | bool Jeste=true;  
1469 | while (A and Jeste) //zde nehrozí problém zkráceného vyhodnocení  
1470 |     if (B) S;  
1471 |     else Jeste = false;  
1472 |  
1473 | while (A or B) S;  
1474 |     lze nahradit za:  
1475 | bool Jeste=true;  
1476 | while (Jeste)  
1477 |     if (A) S;  
1478 |     else if (B) S;  
1479 |     else Jeste = false;
```

19 Literatura

- Heineman, G. T., Pollice, G., Selkow, S. *Algorithms in a Nutshell*. O'Reilly, 2009. ISBN 978-0-596-51624-6.
- Henney, K. *97 klíčových znalostí programátora*. Brno: Computer Press, 2010. ISBN 978-80-251-3145-9.
- idnes.cz. *Zemřel Felix Holzmán, legenda české zábavy*. [online], 2002. Dostupné na https://www.idnes.cz/kultura/film-televize/zemrel-felix-holzmán-legenda-ceske-zabavy.A020920_114411_filmvideo_vlk
- Kokeš, J. *Algoritmy pro inženýrskou informatiku*. Praha: ČVUT, 2006. ISBN 80-01-03515-8.
- *Kopenogram*. [online], 2020. Dostupné na <http://www.kopenogram.org/>
- Knuth, D. E. *Umění programování. Základní algoritmy*. Brno: Computer Press, 2008. ISBN 978-80-2512-025-5.
- Knuth, D. E. *Umění programování. Seminumerické algoritmy*. Brno: Computer Press, 2010. ISBN 978-80-2512-898-5.
- Krček, B., Kreml, P. *Algoritmizace a programování v jazyce Pascal*. Ostrava: Vysoká škola báňská, 1993. ISBN 80-7078-215-3.
- Virius, M. *Základy algoritmizace*. Praha: ČVUT, 2008. ISBN 978-80-01-04003-4.
- Wirth, N. *Algoritmy a struktury údajov*. Bratislava: Alfa, 1988.
- Wróblewski, P. *Algoritmy. Datové struktury a programovací techniky*. Brno: Computer Press, 2007. ISBN 80-251-0343-9.

doc. Ing. Jiří Rybička, Dr.
ALGORITMIZACE
verze dokumentu 7 (26. 9. 2022)
Mendelova univerzita v Brně
Zemědělská 1, 613 00 Brno