PB111 Nízkoúrovňové programování

Petr Ročkai

Část A: Organizace

Část A.1: Informace o kurzu

A.1.1 Prerekvizity

- princip fungování počítače (PB150, PB151)
- základy programování (IB111)
- porozumění psanému textu

A.1.2 Studijní materiály

- tyto poznámky
- sbírka úloh
- literatura
- příklady na internetu

A.1.3 Ukončení

- hodnocena pouze praktická část
- zejména práce během semestru
- programovací test ve zkouškovém
- podrobněji ve sbírce

A.1.4 Seminář

- praktické programování
- předpokládá znalosti z této přednášky
- předpokládá znalosti z IB111

A.1.5 Přehled semestru

1. model výpočtu
2. organizace paměti
3. datové struktury a algoritmy

Část B: Základní pojmy a definice

Část B.1: Výpočetní stroje

B.1.1 Počítač

- von Neumanova architektura
- výpočetní jednotka provádí instrukce
- operační paměť ukládá data
- paměť je **adresovatelná**

B.1.2 Registr

- ukládají čísla (slova, ne slabiky!)
- aritmetické registry
- programový čítač

B.1.3 Instrukce

- instrukce je **elementární příkaz**
- procesor zná jen konečný počet instrukcí
  - lze je tedy **očíslovat**
- instrukce lze sdružovat podle **operací**
  - `add`, `ld`, atp. jsou operace

B.1.4 Výpočet

- **aritmetické** a logické (ALU) instrukce
  - sčítání, odečítání, násobení, dělení
  - logické operace po bitech, bitové posuvy
  - relační operátory
- přístup do **paměti**: `ld` (load), `st` (store)
- **řízení toku**: podmíněné a nepřímé skoky
- (volitelně) realizace podprogramů

B.1.5 Program

B.1.6 Překlad

Část 1: Výpočetní stroj

Část 1.1: Model výpočtu

1.1.1 Motivace

- zjednodušený model počítače
- bez virtualizace, souběžnosti
- 16bitové registry a adresy
- jednoduchá instrukční sada

1.1.2 Stav

- hodnoty uložené v
  - registrech
  - paměti
- konečný počet možností

pc	r_1	r_2	paměť
0000	0000	0000	00 00 00 00 00 00 00 00 00 00 00 00
0004	0007	03ff	de ad de ad de ad de ad de ad de ad
0000	1007	7fff	00 00 00 00 00 00 00 de ad 00 co de

1.1.3 Akce

- počáteční stav = program
- akce = přechod mezi stavy
- stav přesně udává akci
  - determinismus
  - 1 program = 1 výpočet

1.1.4 Speciální stavy

- výpočet může být konečný
- rozlišení důvodu ukončení
  - konec programu
  - chybná instrukce
  - selhání tvrzení

Část 1.2: Instrukční sada

1.2.1 Registry

- 16 obecných registrů
  - rv, l1 … l7, t1 … t6, bp, sp
  - výpočetně zcela rovnocenné
  - sp má navíc speciální využití
  - jména jinak pouze pro lidi
- programový čítač pc

1.2.2 Paměť

- 64 KiB (65536 jednoslabičných buněk)
- adresace 16bitovým číslem
- každá adresa je platná
- program začíná adresou 0

1.2.3 Sémantika

- popis za pomoci mikroinstrukcí
- efekt instrukce na stav
- instrukce určena stavem
  - 4 bajty od adresy dané pc

1.2.4 Operandy

- registrové operandy
  - 0–2 vstupní registry
  - 0 nebo 1 výstupní registr
- přímý operand
  - hodnota je součást instrukce

1.2.5 Kódování

- dvě slova (2·16 = 32b)
- horní slovo
  - kód operace
  - výstupní registr
  - vstupní registr 1
- spodní slovo
  - vstupní registr 2 **nebo**
  - přímý operand

1.2.6 Kopírování hodnot

- 1× vstup + 1× výstup
- nastaví hodnotu registru
- copy $\mathrm{reg}_1 \rightarrow \mathrm{reg}_2$
- put imm $\rightarrow \mathrm{reg}_1$

1.2.7 Binární operace

- aritmetika, srovnání
- bitové operace (logické, posuvy)
- 2× vstup + 1× výstup
- vstup/výstup se mohou překrýt

1.2.8 Operace řízení toku

- efekt na hodnotu `pc`
- nepodmíněný přímý skok `jmp`
- podmíněný přímý skok `jz`, `jnz`
- volání podprogramu později

1.2.9 Řízení stroje

- zastavení `halt`
  - bez podmínky
  - indikuje úspěch
- tvrzení `asrt`
  - podmíněno vstupním registrem
  - nula = neúspěch → chyba

Část 1.3: Řízení toku

1.3.1 Konstrukce strojového kódu

- rekurzivní algoritmus
- postupujeme po struktuře programu
  - blok složíme z příkazů
  - výraz složíme z podvýrazů
- zatím pouze intuitivně

1.3.2 Podmíněný příkaz

- nejjednodušší forma: `if b:` $\texttt{stmt}_1$
  - realizace jedním podmíněným skokem
  - není-li `b` splněno, přeskoč tělo
- rozšíření: `else` větev
  - podmíněný + nepodmíněný skok
  - na konci „then“ přeskočíme za blok `else`

1.3.3 Nekonečný cyklus

- jak zapsat `while True`?
- realizace jedním skokem
  - nepodmíněným
  - na nižší adresu
- časem na něj opět narazíme

1.3.4 Cyklus `while`

- nekonečný cyklus + podmíněný `break`
- `break` je jeden podmíněný skok
  - pro `while` na začátku těla
  - skok za konec těla
- jednodušší: `do-while`

Část 2: Základní prvky jazyka C

Část 2.1: Hodnoty, objekty, proměnné

2.1.1 Hodnota

- význam podobný Pythou
- celé číslo
- později složitější
- 12 ~ XII ~ $(1100)_2$ ~ 0xc

2.1.2 Operace

- pracuje s hodnotami
- hodnoty je potřeba si pamatovat
- realizace výpočetním strojem
- příklad: součet dvou hodnot

2.1.3 Objekt

- abstrakce (zobecnění) paměťové buňky
- pamatuje si **hodnotu**
- má identitu
  - různost při stejné hodnotě
  - stejnost během výpočtu

2.1.4 Proměnná

- vazba jména na objekt
- syntaktický rozsah platnosti
- vazba je neměnná (pevná)
- platnost jména ~ živost objektu

2.1.5 Typ

- je vlastnost hodnoty
- určuje přípustné operace
- určuje chování operací
- pouze v době překladu

2.1.6 Deklarace

- proti Pythonu nový prvek
- `typ jméno = výraz;`
- vytvoří zároveň objekt i vazbu
- bez inicializace = zapovězená hodnota

Část 2.2: Výrazy

2.2.1 Elementární výrazy

- název proměnné: `x` (typ dle proměnné)
- číselný literál:
  - typu `int`: `3`, `-1`, `0x1f`
  - typu `unsigned`: `3u`, `0x1fu`

2.2.2 Aritmetické a logické operace

- popisují hodnotu, žádný vedlejší efekt
- e_1 `+` e_2, ..., e_1 `%` e_2
- e_1 `<<` e_2, e_1 `>>` e_2
- e_1 `&` e_2, e_1 `|` e_2, e_1 `^` e_2
- `-`e_1, `~`e_1, `!`e_1

2.2.3 Výpočet hodnoty výrazu

- vyhodnocení do registru R
- var ~ copy A → R
- e_1 + e_2, ..., e_1 ^ e_2
  - vyhodnoť e_1 do t_1
  - vyhodnoť e_2 do t_2
  - add t_1, t_2 → R

2.2.4 Kontrola typů

- ověří spávnost operace × hodnoty
- vkládá implicitní konverze
  - přesná pravidla jsou složitá
- špatně utvořený program zamítne

2.2.5 Implicitní konverze

1. povýšení – vše menší než `int`
   - vejde se do `int` → `int`
   - jinak `unsigned`
2. stejná znaménkovost → pouze zvětšení
3. různá vede na:
   a. znaménkový je-li striktně větší
   b. jinak na neznaménkový

operand	operand	společný typ
signed char	signed char	int
	unsigned char	int
	int	int
	unsigned	unsigned !
unsigned char	unsigned char	int
	int	int
	unsigned	unsigned
int	int	int
	unsigned	unsigned
unsigned	unsigned	unsigned

2.2.6 Přiřazení

- `var = e1` (pozor na levou stranu)
- proběhne typová konverze pravé strany
- výraz s **vedlejším efektem**
- provede zápis do objektu
- **hodnota** je to, co bylo zapsáno

2.2.7 Booleovské operace

- binární e_1 `&&` e_2, e_1 `||` e_2
- ternární e_1 `?` e_2 `:` e_3

2.2.8 Vyhodnocení booleovských operací

- vyhodnocení e_1 `&&` e_2 do R:
  - vyhodnoť e_1 do R
  - je-li R true, vyhodnoť e_2 do R
- vyhodnocení e_1 `||` e_2 do R:
  - vyhodnoť e_1 do R
  - je-li R false, vyhodnoť e_2 do R

Část 2.3: Příkazy

2.3.1 Výrazový příkaz

- e_1; – jakýkoliv výraz
- provede vedlejší efekty
- hodnota je zapomenuta

2.3.2 Složený příkaz

- sekvence příkazů
- uzavřena do složených závorek
- připouští navíc deklarace
  - rozsah platnosti jmen

2.3.3 Podmíněný příkaz

- `if ( expr ) stmt`
- `if ( expr ) stmt`$_1$ `else stmt`$_2$

2.3.4 Cyklus do … while

- `do stmt while ( expr );`

2.3.5 Řízení iterace

- `break;` – ukončí cyklus
- `continue;` – ukončí aktuální iteraci

2.3.6 Cyklus `while`

- `while ( expr ) stmt`

2.3.7 Cyklus for

- for (decl; $expr_1$; $expr_2$) stmt

Část 3: Podprogramy

Část 3.1: Podprogramy abstraktně

3.1.1 Účel

3.1.2 Zápis

- definice
  - návratový typ
  - jméno
  - formální parametry
- použití (volání)
  - jméno
  - skutečné parametry

3.1.3 Vstupy a výstupy

3.1.4 Sémantika

- čistá
  - nahrazení volání výsledkem
- obecně
  - nahrazení tělem?
  - výsledek + efekt

3.1.5 Typy

- návratový typ
  - patří **výrazu volání**
- typy parametrů
  - výrazy **skutečných** parametrů
- kontrola použití bez těla

3.1.6 Parametry

- formální parametr ~ proměnná
  - má **vlastní objekt**
- předání parametru
  - jako **inicializace** proměnné
  - předání **hodnotou**

3.1.7 Rekurze

- použití v těle (definici)
- neznámá hloubka zanoření
- koncová vs obecná

Část 3.2: Operační sémantika

3.2.1 Zásobník

- oblast v paměti
- speciální registr sp = adresa
- souvisí s datovou strukturou
  - last in, first out
  - operace push, pop
  - sp ~ top

3.2.2 Sémantika push, pop

- push reg
  - sub sp, 2 → sp
  - st reg → sp
- pop reg
  - ld sp → reg
  - add sp, 2 → sp

3.2.3 Předání řízení

- vstup: `call`
  - pouze návratová adresa
  - **neřeší** parametry
- konec: `ret`
  - `pop` + pouze skok
  - **neřeší** parametry

3.2.4 Sémantika `call`, `ret`

- `call fn` (jeden krok!)
  - `push pc`
  - `jmp fn`
- `ret` ~ `pop pc`
  - nebo: `pop X`, `jmp X`
  - **nepřímý** skok

3.2.5 Předávání parametrů

- přednostně v registrech
  - `l1`, `l2`, ...
  - další na zásobník
- podobně návratová hodnota (`rv`)
- složitější typy později

3.2.6 Volací sekvence

- f(e_1, e_2, …, e_n)
  - vyhodnoť e_1 do l1
  - vyhodnoť e_2 do l2
  - call f

3.2.7 Lokální proměnné

- hodnota se musí zachovat
- „zálohování“ registrů
  - na zásobník (push, pop)
  - do rámce

Část 3.3: Rámec

3.3.1 Přelití registru

- potřebujeme volný registr
- co když jsou všechny „živé“
- přesuneme hodnotu na zásobník
  - stejně jako při „zálohování“

3.3.2 Lokální proměnné

- lokálních proměnných může být „moc“
- uložíme je na zásobník
- načtení do „dočasných“ registrů
- příště: adresa proměnné

3.3.3 Vyhodnocení proměnné

- vyhodnocujeme do `tmp`
- „žije“ v registru:
  - `copy reg → tmp`
- „žije“ na zásobníku:
  - `ld bp, offset → tmp`
  - offset = vlastnost proměnné

3.3.4 Rámec

- privátní paměť podprogramu
- alokuje se na zásobníku
- různé aktivace → různé rámce
- pevná velikost

3.3.5 Bookkeeping

- při vstupu do podprogram:
  - `push bp`
  - `copy sp → bp`
  - `sub sp, N → sp`
- při výstupu:
  - `copy bp → sp`
  - `pop bp`

Část 4: Ukazatele

Část 4.1: Objekt, identita, adresa

4.1.1 Hodnota

4.1.2 Objekt

- buňka pro uložení hodnoty
- typicky skrze proměnnou
- má identitu
- nemá pevnou adresu

4.1.3 Identita objektu

- identita je abstraktní
- ???

4.1.4 Adresa

- číselné označení buňky
- adresu lze vypočítat
- označuje **slabiky**/slova
  - **nikoliv hodnoty** jazyka
  - čtení/zápis z/na adresu

4.1.5 Adresa vs identita

- jak reprezentovat identitu
- adresa prvního bajtu
- co objekt v registru?
- co přesun objektu

4.1.6 Ukazatel

- identita jako hodnota
- načti/zapiš **hodnotu**
  - nikoliv slabiku/slovo

4.1.7 Typ ukazatele

- obsahuje **typ objektu**
- zapisujeme `typ *`
- existuje pro každý typ
- kolik různých typů?

4.1.8 Operátor adresy

- nový tvar výrazu – &var
- pracuje s **objektem**
- výsledek je **ukazatel**
- použití **fixuje** adresu objektu

4.1.9 Operátor dereference *

- nové tvary výrazů
  - $*e_1$ – výsledek je **objekt**
  - $*e_1 = e_2$ (nepřímé přiřazení)
- objekt vs hodnota
  - l-hodnota vs r-hodnota
  - l-kontext vs r-kontext

4.1.10 Platnost ukazatele

- platná adresa ≠ platný ukazatel
- **musí** ukazova na objekt
- typ ukazatele = typ objektu
- vstupní podmínka dereference

4.1.11 Výstupní parametr

- realizace ukazatelem
- `void foo( int *out )`
- `out` → kam zapsat výsledek
- volající odpovídá za objekt

Část 4.2: Přetypování

4.2.1 Přetypování aritmetických typů

4.2.2 Přetypování ukazatelů

4.2.3 Typy a objekty

Část 5: Pole

Část 5.1: XXX

5.1.1 Složené objekty

- umožňují uložit více hodnot
- složené z **podobjektů**
- není totéž jako složená hodnota!

5.1.2 Pole

- typ složeného objektu
- pevný počet podobjektů
  - podobjekt ~ prvek
- podobjekty jsou **očíslovné**
- index = celé číslo

5.1.3 Syntaxe

- deklarace `typ jméno[počet]`
  - lokální proměnná
- počet musí být konstanta
- použití = ukazatel (decay)
  - `jméno` ~ `&jméno`

5.1.4 Inicializace

5.1.5 Operátor indexace

- nový tvar výrazu
- zjednodušeně $\mathtt{var}[e_1]$
- obecně $e_1[e_2]$
  - e_1 musí být ukazatel
- výsledek $\rightarrow$ i-tý podobjekt

5.1.6 Ukazatelová aritmetika

- analogie aritmetiky adres
  - ukazatel + číslo → ukazatel
  - ukazatel − ukazatel → číslo
- odpovídá indexaci
- `a[i]` ~ `*(a + i)`

5.1.7 Pasti

- pole není hodnota
  - nelze kopírovat jako celek
  - předáváme jako ukazatel
- nemá informaci o velikosti
- ani o (ne)využitých položkách

Část 6: Struktury, zřetězený seznam

Část 6.1: Struktury

6.1.1 Složená hodnota

- hodnota složená z jiných hodnot
- podhodnoty = složky
- příklad: uspořádaná dvojice
- příklad: seznam (v Pythonu)

6.1.2 Složený typ

- typ složené hodnoty
- určuje typy složek
- vztahuje se i na objekty

6.1.3 Složené hodnoty vs objekty

- složený objekt → má podobjekty
- složená hodnota → má složky
- podobjekt obsahuje hodnotu
- složky musí „pasovat“ na podobjekty

6.1.4 Záznamový typ

- anglicky **record type**
- obecná, běžná konstrukce
- pevné typy a pořadí složek
- **pojmenované** složky

6.1.5 Struktura

- záznamový typ v C
- zavedení klíčovým slovem `struct`
- tělo: typy a jména složek
- jméno typu: `struct jméno`
- literál neexistuje

6.1.6 Inicializace

- proměnné lze inicializovat
- proběhne po složkách
- `struct pair x = { 1, 2 };`
- `… x = { .a = 1, .b = 2 };`
- inicializace ≠ přiřazení

6.1.7 Přístup ke složce

- výraz tvaru e_1`.jméno`
- může se vyhodnotit na objekt
  - když je e_1 objekt
  - při dočasném zhmotnění […]
- adresy musí být „pohromadě“

6.1.8 Nepřímý přístup

- výraz tvaru e_1`->jméno`
- je vždy objektem (l-hodnotou)
- e_1 musí být typu ukazatel
- vstupní podmínka: platnost e_1

6.1.9 Přiřazení

- uložení hodnoty to objektu
- u struktur proběhne po složkách
- rekurzivně do podstruktur, atp.

6.1.10 Pole ve struktuře

- jméno může označovat i pole
- každému prvku odpovídá složka
- struktura je stále hodnota
- pole stále není hodnota

6.1.11 Dočasné zhmotnění

- co znamená přístup do pole
  - $e_1[e_2] \sim *(e_1 + e_2)$
  - co je `f().foo[1]`?
- při přístupu vytvoříme objekt
- zanikne s koncem příkazu

Část 6.2: Zřetězený seznam

6.2.1 Princip

- složený ze samostatných uzlů
- uložených na nezávislých adresách
- každý reprezentuje jeden prvek
- ukazatel na další uzel

6.2.2 Výhody

- jednoduchá implementace
- velmi flexibilní
  - fronta
  - zásobník
  - seznam (iterace)

6.2.3 Nevýhody

- nahodilý přístup do paměti
- větší paměťová režie
- nelze indexovat
- nešikovná abstrakce

6.2.4 Reprezentace

- struktura pro uzel
- lze mít samostatnou hlavu
- `struct node *next`
- pevný typ uložené hodnoty

6.2.5 Dvojité řetězení

- jednodušší odstranění uzlu
- složitější přidání uzlu
- výrazně větší režie
- obvykle se nepoužívá

6.2.6 Základní operace

- vložení
- iterace
- odstranění

6.2.7 Iterace

- při práci s celým seznamem
- typické využití cyklu `for`:
  - `struct node *n = head`
  - podmínka `n`
  - posun `n = n->next`

6.2.8 Vložení uzlu

- potřebujeme znát pozici
- ukazatel na předchozí uzel
- na začátek → triviální
- na konec → ukazatel navíc

6.2.9 Odstranění

- opět potřebujeme znát pozici
- ukazatel na **předchozí** uzel
- ze začátku → triviální
- z konce → dvojité řetězení

Část 7: Dynamická alokace

1 Definice problému

2

Část 8: Správa paměti

Část 8.1: Dynamická velikost

8.1.1 Automatické proměnné

- uložené v rámci
- zabírají prostor na zásobníku
- zásobník má omezenou velikost

8.1.2 Paměťová složitost

- podobně jako časová $\mathcal{O}(f(n))$
- n bude **velikost problému**
  - nemusí nutně být pouze vstup
- alokace na zásobníku – $\mathcal{O}(1)$ OK
  - $\mathcal{O}(\log n)$ – obvykle OK

8.1.3 Jazyk C

- proměnné mají konstantní velikost
- alokace tedy pouze $\mathcal{O}(1)$
- pozor ale na počet proměnných

8.1.4 Rekurze

- podobné omezení – max. $\mathcal{O}(\log n)$
- např. merge sort, binární hledání
- co backtracking?
  - exponenciální čas
  - lineární hloubka rekurze

8.1.5 Meze rekurze

- co stromové struktury?
  - může a nemusí být OK
  - hloubka ne vždy $\mathcal{O}(\log n)$
- koncová rekurze
  - lze v konstantní paměti
  - v C to ale není povinné

8.1.6 Dynamická alokace

- bez striktních omezení
- adresní prostor
  - 64b bez problémů
  - omezený pro ≤ 32b adresy
- paměť není nekonečná

8.1.7 Standardní C

- knihovna: `malloc`, `free`
- vyžaduje interakci s OS
- `malloc` je (trochu) magický

Část 8.2: Dynamická živost

8.2.1 Statická živost

- lokální proměnné
- život = rozsah platnosti
- jednoduché ale omezující

8.2.2 Dynamická živost

- dynamická paměť
- rozhoduje se za běhu
  - např. v podmínce
- významný zdroj chyb

8.2.3 Uvolnění

- konec živosti objektu
- ukazatele dále existují
  - jsou již ale **neplatné**
  - mrtvý objekt nesmí být použit
- koordinace je kritická

8.2.4 Klasifikace chyb

- použití po uvolnění
  - lze rozlišit zápis/čtení
- dvojité uvolnění
- chybějící uvolnění (únik)

8.2.5 Použití po uvolnění

- dereference neplatného ukazatele
  - nedefinované chování
- čtení → nesmyslná hodnota
- zápis → poškození dat
- ohrožení metadata alokátoru

8.2.6 Dvojité uvolnění

1. neplatný ukazatel
   - poškození metadat
   - nedefinované chování
2. omylem platný ukazatel
   - uvolní jiný objekt
   - další použití je pak UB

8.2.7 Únik zdrojů

- objekt nebyl uvolněn
  - striktně – už nebude použit
  - volně – neexistuje ukazatel
- situačně závislé
  - použití může být zbytečné

8.2.8 Obecněji o zdrojích

- paměť není jediný zdroj
- platí stejná pravidla
  - zdroj ~ objekt
- soubory, mutexy, atp.

8.2.9 Pseudostatická živost

- statická živost je přirozená
- nelze vždy použít i když stačí
  - dynamicky velké pole
  - jiné zdroje než objekty
- syntaktické párování alokace/dealokace

Část 9: Dynamické pole

Část 9.1: Abstraktní datové struktury

9.1.1 Indexovatelný seznam

- abstraktní datová struktura
- operace:
  - `append` – vloží prvek na konec
  - `remove` – odstraní poslední prvek
  - `get` – vrátí prvek na daném indexu
  - `size` – vrátí aktuální počet prvků
- srovnejte zásobník

Část 9.2: Implementace

9.2.1 Dynamické pole

- zřetězený seznam → get je $\mathcal{O}(n)$
- vyhledávací strom → get je $\mathcal{O}(\log n)$
- standardní pole → neumí append

9.2.2 Implementace: dynamické pole

- **dynamické pole** → vše konstantní
  - ale pouze amortizovaně
- vyhradíme paměť jako u běžného pole
- dojde místo → realokace
  - alokujeme větší pole
  - dosavadní prvky přesuneme
  - původní paměť uvolníme

9.2.3 Organizace v paměti

- ukazatel, velikost, kapacita
- struktura se 3 položkami
- hlavička před začátkem pole
- předávání, vstupně-výstupní parametr

9.2.4 Reprezentace položek

- v poli pouze ukazatel
  - výhodné pro velké položky
  - nahradit zřetězeným seznamem?
- položka přímo v poli
  - ideální pro malé položky
  - ukazatel na položku není trvalý

9.2.5 Platnost ukazatelů

- zvětšení zneplatňuje ukazatele
  - nelze dlouhodobě ukládat
- pozor na append během iterace
- nevíme který append je bezpečný

9.2.6 Amortizace

- uvažme posloupnost n operací
- jaká je průměrná složitost jedné?
  - n vložení v čase $\mathcal{O}(n)$
  - na jedno vložení připadne čas $\mathcal{O}(1)$
- metody analýzy $\rightarrow$ IB114

9.2.7 Taktika zvětšování

- o konstantu
  - špatná asymptotická složitost
  - pouze speciální situace
- na dvojnásobek
  - zaručuje správnou složitost
  - jednoduché a účinné

9.2.8 Exponenciální zvětšování

- na abstraktní úrovni optimální
- dopad na využití paměti?
  - fragmentace adresního prostoru
  - (ne)využitelnost uvolněné paměti
- možnosti řešení:
  - alternativní schéma zvětšování
  - speciální podpora v alokátoru

9.2.9 Další využití

- zásobník → přímočaré
- fronta → kruhová + přesuny
- fronta → dva zásobníky
- prioritní fronta

9.2.10 Zmenšování

- nezvratný růst nemusí být ideální
- naivní zmenšování nefunguje
  - „thrashing“ okolo meze zvětšení
- hystereze, zaplnění < 0,5

9.2.11 Využití paměti

- n počet prvků, k velikost
- zarovnání na mocninu dvou
- minimum (ideální) $n \cdot k$
- maximum (nejhorší) $2n \cdot k$
- průměr $1,5n \cdot k$

9.2.12 Srovnání

- p = velikost ukazatele
- zřetězený seznam: $n \cdot (k + p)$
- binární strom: $n \cdot (k + 2p)$
- dynamické pole: $1,5n \cdot k$
- velikost, kapacita: $+2p$

9.2.13 Režie alokátoru

- velmi variabilní
- minimální velikost alokace
- fragmentace zvětšováním pole
- výrazný dopad na obojí

9.2.14 Efektivita v praxi

- sekvenční vs náhodný přístup
- efektivita využití cache
  - umístění mrtvé paměti

Část 9.3: Binární halda

Část 10: Slovníky a množiny

Část 10.1: Hašovací tabulky

Část 10.2: Vyhledávací stromy

Část 11: Paměť

Část 11.1: Objekty

11.1.1 Připomenutí: objekt

- objekt je zobecnění paměti
- vzniká **deklarací proměnné**
- ukládá **hodnotu** (ne bajty!)
- může mít přidělenu adresu

11.1.2 Paměť

- „pole bajtů“
- adresa ~ index
  - (± díry v adresním prostoru)
- bajt ~ `unsigned char`
  - hodnota 0–255

11.1.3 Překrývání (aliasing)

- objekty se stejnou adresou
- typicky není dovoleno
- výjimky:
  - reprezentace
  - kompatibilní typy

11.1.4 Reprezentace objektu

- bajty objektu = reprezentace
- reprezentace je sama objektem
- kódování definováno implementací
- nemusí být jednoznačná

11.1.5 Přetypování

- objekt a reprezentace sdílí adresu
- ukazatele mají různé typy
- přesto lze bezpečně přetypovat
  - reprezentace → `unsigned char *`
- jen pozor na `const`

11.1.6 Vznik objektu

- paměť ~ `unsigned char[N]`
- zápisem reprezentace
  - kopie po bajtech
  - přiřazení

11.1.7 Neplatné objekty

- objekt lze konstruovat z bajtů
- ne každá reprezentace je platná
- čtení z neplatného objektu → UB

11.1.8 Ukazatel bez typu

- `void *` – neurčuje typ objektu
- není dovoleno dereferencovat
- povoluje implicitní přetypování
- nesmí porušit pravidla o překrývání

Část 11.2: Alokace

11.2.1 Zadání úlohy

- nalézt nepoužitou paměť
- v nějaké větší oblasti
  - typicky od operačního systému
  - nemusí být souvislá
- zadané minimální velikosti

11.2.2 Dynamická paměť

- nekonstantní velikost alokace
- živost určená za běhu
  - vedlejší efekt výrazu
  - detailněji příště

11.2.3 Další požadavky

- co nejrychleji
- co nejméně nevyužitelné paměti
  - fragmentace adresního prostoru
  - metadata alokátoru

11.2.4 Strategie

- rozdělit volný blok
  - musí být dostatečně velký
  - first-fit → první takový
  - best-fit → nejmenší takový
- sub-alokátory
  - kombince různých strategií

11.2.5 Předalokace

- typicky pro malé velikosti
- paměť se chystá dávkově
  - třeba po 4KiB blocích
  - všechny objekty stejně velké

11.2.6 Datové struktury

- zřetězený seznam volných bloků
  - uzel **uvnitř** volného bloku
  - zaužívaný název freelist
- tabulka indexovaná velikostí
- hashovací tabulky

Část 11.3: Realokace paměti

11.3.1 Situace

- implementujeme dynamické pole
- potřebujeme pole zvětšit
- triviálně:
  - nová alokace + kopie dat
  - dealokace původního
- lze provést i lépe?

11.3.2 Využití stávající alokace

- vyžaduje spolupráci alokátoru
- použít následující volný blok
  - optimální – není potřeba přesun
  -
- použít předchozí blok

11.3.3 Využití předchozího bloku

- vyžaduje přesun dat
  - překryv podle poměru velikostí
  - dvojnásobek $\rightarrow$ bez překryvu
- celková potřebná paměť m
  - oproti $m + n$ pro novou alokaci
  - dvojnásobek $\rightarrow$ $2n$ vs $3n$

11.3.4 Přesun dat

- kopie for cyklem po bajtech
- co překrývající se oblasti
  - někdy je potřeba iterovat odzadu
  - mnohem méně efektivní
- alternativní metody dle platformy

11.3.5 Situace v praxi

- knihovna jazyka C má realloc
  - umožňuje i zmenšení alokace
  - poněkud komplikované rozhraní
- Rust, D také nabízí realloc
- C++ od podpory upustilo