PB111 Nízkoúrovňové programování

Petr Ročkai

Část A: Organizace

Vítejte v předmětu PB111, kde se budeme zabývat modelem výpočtu, který popisuje fungování běžných počítačů. Tato skripta podávají **teoretický** pohled na věc a slouží jako předloha pro přednášky.

A.1: Informace o kurzu

A.1.1 Prerekvizity Tento kurz předpokládá jen minimum znalostí – je ale velmi důležité, abyste měli zvládnutou látku předmětu IB111. Znalosti z předmětů PB150 nebo PB151 o základech fungování počítačů budou jistě výhodou. Překvapivě důležitým požadavkem tohoto předmětu je schopnost soustředit se na psaný text (to platí jak pro teoretickou, tak praktickou část předmětu).

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

A.1.2 Studijní materiály Studijní materiály tohoto předmětu jsou rozděleny do dvou částí – tyto poznámky jsou ta více teoretická, která se váže k přednášce. Praktická část předmětu pak používá sbírku příkladů, která obsahuje krom samotných úloh také řadu prakticky zaměřených ukázek a další materiál spíše referenčního charakteru.
XXX Literatura

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

A.1.3 Ukončení Tento předmět je ukončen **zkouškou**. Je-li pro Vás předmět nepovinný, můžete si jej zapsat také na zápočet – v takovém případě budete hodnoceni pouze ze semestrální práce. Přesný popis hodnocení předmětu a zejména bodování naleznete v kapitole A sbírky.

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

A.1.4 Seminář XXX

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

A.1.5 Přehled semestru Kurz je rozdělen na 3 velké tematické celky. První měsíc se budeme zabývat převážně tím, jak probíhá výpočet na úrovni procesoru – popíšeme si jaké prostředky a operace nám procesory nabízí a jak nám tento velmi jednoduchý model umožňuje spouštět libovolně složité programy. Naučíme se zejména jak se na úrovni stroje realizují standardní prvky jazyků vyšší úrovně.[1]
Druhý blok se bude zabývat organizací paměti. V kapitolách 5–8 probereme základní stavební prvky datových struktur – pole a ukazatele – a základy dynamické alokace paměti.
V posledním bloku se pak budeme blíže zabývat konkrétními datovými strukturami: dynamické pole, binární halda, hašovací tabulka a vyhledávací strom.[2] Zejména se zaměříme na jejich nízkoúrovňovou realizaci za pomoci znalostí a dovedností nabytých v prvních dvou blocích.

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

Část B: Základní pojmy a definice

Tato kapitola předchází prvnímu bloku a zavádí pojmy a koncepty, které budeme potřebovat během celého semestru. Bude-li některý hrát v pozdější přednášce obzvláště důležitou roli, jeho definici si připomeneme. Jinak ale budeme v dalším předpokládat, že tyto pojmy znáte.

[1] Nebudeme se zde ovšem zabývat interakcí s vnějším světem ani souběžností – tato témata spadají do kurzu PB152 Operační systémy. Pro účely tohoto kurzu pracujeme s izolovaným sekvenčním počítačem který má pouze výpočetní jednotku (procesor) a paměť.
[2] Tou dobou je budete již dobře znát z předmětu IB002 Algoritmy a datové struktury.

B.1: Výpočetní stroje

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

B.1.1 Počítač Počítač je složitý elektronický stroj – **vnitřní detaily** jeho fungování jsou mimo náš současný obzor a zejména bychom se jimi radši nezabývali. Díky **abstrakci** se ale můžeme na počítač dívat jako na relativně jednoduché **výpočetní zařízení**, které má jednotné **vnější chování** – bez ohledu na to, kdo vyrobil paměťové moduly nebo procesor, jaké tento obsahuje tranzistory, jaké je chemické složení polovodičů v těchto tranzistorech, atd.

Obecně používanou abstrakcí pro počítač je tzv. **von Neumanova architektura**[3], která má několik modulů, které lze rozdělit do dvou kategorií. Tou první (a jedinou, kterou se budeme v tomto předmětu zabývat) jsou **základní výpočetní zdroje**, totiž:

- **výpočetní jednotka** (procesor, CPU, procesorové jádro, atp.) je zařízení, které vykonává **instrukce** (elementární příkazy, pomocí kterých počítač programujeme),
- **operační paměť** (obvykle jednoduše „paměť“, případně RAM, pracovní paměť) je zařízení, které si **pamatuje data** (čísla), se kterými program při svém výpočtu pracuje, a které je schopno tato data předávat výpočetní jednotce a na její pokyn je měnit.

Instrukce, které výpočetní jednotka vykonává, jsou uloženy, podobně jako pracovní data programu, v operační paměti. Operační paměť je **adresovaná**: to znamená, že je složena z **očíslovaných buněk**, kde každá buňka si pamatuje jeden **bajt** (číslo v rozsahu 0 až 255). Buňky jsou očíslovány po sobě jdoucími celými čísly, obvykle počítáno od nuly. Instrukce mohou vyžádat načtení čísla z libovolně vypočtené adresy[4] – tím se paměť liší od **registrů**, které mají pevná jména (program nemůže „spočítat“ které dva registry má sečíst). Jedná se o podobný rozdíl (opět nikoliv náhodou), jako mezi celočíselnou **proměnnou** a **seznamem**.[5]

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

B.1.2 Registr Podobně jako paměť, registry slouží k ukládání čísel – existují dva klíčové rozdíly mezi registry a pamětí:[6]

1. pojmenování registru je pevnou součástí instrukce, kdežto paměťovou adresu lze vypočítat (paměť lze indexovat, registry nikoliv),
2. reprezentace čísla v registru je **monolitická** – registry nejsou složené ze slabik, daný registr obsahuje **celé slovo** (částečně důsledek předchozího bodu: registr lze pojmenovat pouze jako celek).[7]

> **Příklad:** Budeme-li potřebovat mluvit konkrétně, budeme používat výpočetní stroj `tiny`, který má 16 aritmetických registrů `r0`, `l1` … `l7`, `t1` … `t6`, `bp` a `sp`. Jak velká čísla lze do registrů ukládat je opět vlastností konkrétního procesoru – aby se nám dobře psala a četla, budeme používat 16b registry. Moderní počítače mají často registry o šířce 64 bitů (někdy též 32, méně jen u velmi malých počítačů).

Vyhrazený registr (programový čítač, angl. **program counter**, někdy také instruction pointer, budeme jej označovat `pc`) pak obsahuje adresu právě vykonávané instrukce.[3] Tento registr **rozhoduje** o tom, která instrukce se má vykonat, není do něj ale obvykle možné zapisovat běžnými (aritmetickými, atp.) instrukcemi. K tomu jsou určeny instrukce řízení toku, kterých hlavním efektem je právě změna hodnoty programového čítače.

- 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.1.3 Instrukce je **elementární příkaz** strojového kódu; to znamená:

- **elementární** – je to nejmenší jednotka činnosti, kterou lze procesoru zadat,
- **příkaz** – instrukce řídí činnost procesoru, „přikazují“ mu provedení nějaké akce.

Instrukcí budeme nazývat pouze celek, který obsahuje veškeré informace potřebné k provedení konkrétních akcí (zejména udává **operaci**, která se má provést, a **konkrétní registry**, se kterými se bude pracovat a také konkrétní **přímé operandy**).

---

[3] Je také obvyklé, že přístup do registrů je mnohem rychlejší než do paměti. To je pro nás v tuto chvíli celkem nepodstatné – budeme tedy pracovat s abstrakcí, kde každá instrukce trvá stejně dlouho, bez ohledu na to, co počítá, nebo jestli přistupuje do paměti.

[4] Možná z architektury x86 znáte např. registry `al`, `ah`, `ax`, `eax`, `rax`, které se „překrývají“. To lze ale chápat tak, že skutečný registr je pouze `rax` (resp. `eax` na 32b procesorech, `ax` na 16b procesorech) a ty ostatní jsou virtuální: „syntaktické zkratky“ pro extrakci příslušných bitů skutečného registru. Vztah mezi „pojmenovaným“ (virtuálním) registrem a fyzickým registrem (klopnými obvody) je ostatně ve skutečnosti dost komplikovaný. Zde si vystačíme s abstrakcí, kde se registry nepřekrývají a nebudeme virtuální a fyzické registry rozlišovat.

[5] V tomto předmětu vůbec neuvažujeme virtualizaci, nebudeme tedy ani rozlišovat virtuální a fyzické adresy. Z pohledu běžného (ne systémového) programu existuje jediný typ adres i v reálném světě.

[6] Z jednotlivých příkazů jazyka symbolických adres je nutné konkrétní instrukce nejprve vypočítat. Blíže je tento proces popsaný v sekci B.3 sbírky. Na rozdílu mezi instrukcí a příkazem jazyka symbolických adres sice nebudeme bazírovat, ale přesto je dobré mít ho na paměti.

[7] Operace je **obvykle** určena **operačním kódem**: autoři instrukčních sad často volí nějakou pravidelnou strukturu čísel, které jednotlivé instrukce kódují – v takovém případě je operační kód ta část, která určuje **operaci**. Jiné části obvykle určují operandy (registry, se kterými se bude pracovat). Pozor: kódování instrukcí žádnou takovou strukturu dodržovat **nemusí**. A aby nebylo zmatku málo, někdy se operačním kódem myslí i kódování instrukce jako celku.

Instrukcí je pouze konečně mnoho, je tedy zejména možné je **očíslovat** (nebo jinak konečně kódovat, např. do sekvencí bajtů). Každé takové číslo (kódování) popisuje konkrétní akci, kterou může procesor provést.

> **Příklad:** Instrukce z pohledu člověka vypadají například takto:
>
> - `add l1 l2 → l1`; jiná (i když podobná) instrukce je
> - `add l1 l2 → rv`; ještě jiná je
> - `add l1 l3 → l1`; atd.
>
> Takto instrukce zapisujeme pro vlastní potřebu – tomuto zápisu bychom mohli říkat mnemonický zápis strojového kódu, ale v dalším nebudeme striktně zápis a podstatu strojového kódu odlišovat – mezi číselným a mnemonickým zápisem je podobný vztah jako mezi „7“, „VII“ nebo „sedm“. V počítači jsou instrukce reprezentovány čísly (zakódovanými v paměti jako posloupnost bajtů):
>
> - `add l1 l2 → l1` může odpovídat např. `0x31112000`,
> - `add l1 l2 → rv` pak `0x31012000`,
> - `add l1 l3 → l1` zase `0x31113000`, atp.
>
> Pozor, mnemonický zápis není totéž co **jazyk symbolických adres** (angl. **assembly language**). Používají sice velmi podobnou syntaxi, ale v jazyce symbolických adres nezapisujeme nutně přímo instrukce.[8] To, co jsme zde nazvali mnemonickým zápisem, je podmnožinou jazyka symbolických adres.

Striktně vzato tedy nemůžeme obecně mluvit o „instrukci `add`“ – nejedná se o jedinou instrukci, ale o celou rodinu instrukcí. Samozřejmě má ale smysl mluvit o takto příbuzných instrukcích nějak souhrnně. Takovou rodinu instrukcí můžeme popsat pomocí společné **operace**[2] (např. sčítání), která typicky určuje také počet a typ **operandů** (registrů).

B.1.4 Výpočet Procesor **vykonává instrukce**, čím **realizuje výpočet**. Nejjednodušší třídou instrukcí jsou tzv. aritmetické a logické instrukce (tedy ty, které provádí ALU – aritmeticko-logická jednotka). Tím se myslí zejména:

- **aritmetika**: sčítání, odečítání, násobení a dělení,
- **bitové** operace: `and`, `or`, `xor` po bitech, bitové posuvy,
- **srovnání** dvou hodnot (rovnost, nerovnost) – výsledek se uloží do běžného registru nebo do stavového příznaku procesoru.

Další třídy instrukcí provádí složitější akce, které mění stav paměti, nebo řídí průběh výpočtu:

- již zmiňovaný přístup do paměti (`ld`, `st`),
- řízení toku – rozhodnutí o tom, kterou instrukcí bude výpočet pokračovat (zejména **podmíněný skok**, případně **nepřímý skok**[9]),
- instrukce pro realizaci podprogramů zahrnují jak přístupy do paměti, tak řízení toku: patří sem zejména
  - práce s hardwarovým zásobníkem (přesuny hodnot mezi zásobníkem a registry),
  - instrukce pro aktivaci podprogramu (přímo nebo nepřímo),
  - instrukce pro **návrat** z podprogramu.

Instrukční sadu výpočetního stroje `tiny`, kterou budeme v tomto předmětu používat, si popíšeme postupně, podle toho jak budeme nové operace potřebovat při výkladu.

- 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 Programem budeme v tomto kurzu nazývat **předpis**, kterým je určen nějaký **výpočet**. Tento předpis je možné zadat různým způsobem, my budeme využívat dva „textové“ (určené pro zápis a čtení člověkem):

- zápis v jazyce symbolických adres, nebo
- zápis v jazyce C.

V obou případech se jedná o **text**, který je utvořen podle specifických pravidel (dodržuje **syntaxi** daného jazyka) a kterého význam je přesně určen **sémantikou** tohoto jazyka.
Sémantika zároveň určuje, co považujeme pro daný program za **vstup** a **výstup** (výsledek). Pro jednoduchost (a trochu překvapivě i bez újmy na obecnosti) se můžeme omezit na programy, které nemají žádný vstup a jejich výstup je pouze „skončil bez chyby“ a „jiný výsledek“ (což může být „nikdy neskončí“ nebo „skončil chybou“).[1]
Takto omezeným programům se často říká „uzavřené“ – všechny hodnoty, které vstupují do výpočtu, jsou součástí programu a každý (deterministický) program tak může mít jeden jediný výsledek.[10]

---

[8] Uvedeno spíše pro úplnost – v tomto předmětu nepřímé skoky ani nepřímá volání používat nebudeme.

[9] Toto zjednodušení si ovšem můžeme dovolit pouze na úrovni programu jako celku – u podprogramů už se diskusi o vstupech a výstupech nevyhneme.

[10] Tento typ programů velmi dobře znáte – tuto formu totiž měla každá příprava v předmětu IB111, a proto by Vás ani nemělo příliš překvapit, že i takto omezené programy mohou mít velmi bohaté vnitřní chování (průběh výpočtu).

B.1.6 Překlad Zároveň budeme za program považovat předpis zadaný jako **strojový kód** – tento nebudeme nikdy přímo zapisovat, ale vznikne automatickým **překladem** z některé výše uvedené formy. Klíčovou vlastností překladu je, že **zachovává sémantiku** – původní a přeložený program vypočtou pro pro stejné vstupy ten stejný výsledek.
Abychom se vyhnuli komplikacím s ekvivalencí vstupů (vstupy jsou v různých jazycích zadávány různě), zaměříme se na již zmiňované „uzavřené" programy.[11]
Překladem z jazyka A do jazyka B nazveme funkci (v matematickém smyslu), která pro **každý** platný program v jazyce A určí **ekvivalentní** program v jazyce B, přitom programy považujeme za ekvivalentní, mají-li tentýž výsledek (který je jednoznačně dán programem).[12] Překladač je pak program, který tuto funkci realizuje.

Část 1: Výpočetní stroj

V této kapitole si představíme základní výpočetní model, který pak budeme používat celý semestr. Realizovat jej bude výpočetní stroj tiny, který je sice velmi jednoduchý (minimální implementace v jazyce Python má přibližně 200 řádků kódu), ale umožní nám spouštět libovolné programy napsané v jazyce C.[13]

1.1: Model výpočtu

Abychom se mohli o výpočtech přesně vyjadřovat (a také přesně uvažovat), musíme si zavést vhodný **model** výpočtu – efektivně matematický aparát, který nám umožní výpočet uchopit jako objekt. Standardní metodou, jak výpočet popsat, je jako sled **stavů** (konfigurací) nějakého **abstraktního stroje** a přechodů – **akcí** – mezi nimi.

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

1.1.1 Motivace Protože tento předmět je zaměřený spíše prakticky (a pragmaticky), chceme aby náš abstraktní výpočetní stroj co nejpřesněji odpovídal skutečným počítačům. Zároveň se ale nechceme takříkajíc „utopit v detailech" – proto bude náš konkrétní výpočetní model určitým kompromisem.
Většinu modelu a terminologie si půjčíme ze světa skutečných počítačů, v několika ohledech si ale situaci zjednodušíme:

1. nebudeme se zabývat virtualizací a souběžností – náš „počítač" bude najednou vykonávat jediný sekvenční program,
2. omezíme se na výpočty se 16bitovými hodnotami a na 16bitovou adresaci – to nám umožní mít lepší přehled o obsahu paměti, a zároveň nám velmi usnadní čtení adres a čísel obecně,
3. instrukční sada tiny je oproti těm reálným velmi malá (obsahuje mnohem méně operací), jednodušší (jednotlivé operace toho dělají méně najednou) a pravidelnější.

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

1.1.2 Stav Jak jsme zmínili výše, důležitým aspektem výpočtu jsou **stavy** výpočetního stroje. Stav stroje tiny má tři složky:[14]

1. obecné registry,
2. programový čítač,
3. paměť.

Přesněji řečeno, stavem myslíme konkrétní hodnoty „uložené" v těchto složkách. Pro ilustraci si na chvíli stroj ještě zmenšíme (na velikost, která už není pro výpočty prakticky použitelná, ale která se nám vejde na stránku). Přisoudíme mu pouze 2 obecné registry (prozatím je nazveme r_1 a r_2), programový čítač pc a 12 bajtů (slabik) paměti. Stavy takovéhoto vskutku mikroskopického výpočetního stroje vypadají jako řádky této tabulky:

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

[11] Situace v „reálném světě" je přirozeně mnohem komplikovanější, ale většina této komplikovanosti souvisí s definicí vstupu a výstupu, zejména v případech, kdy nedokážeme vstup a/nebo výstup jednoduše popsat. Za tento problém pak vděčíme souběžnosti (a s tím související komunikaci). V tomto předmětu se souběžností vůbec zabývat nebudeme. Na teoretické úrovni se ale souběžností zabývá kurz PB152 Operační systémy.

[12] Jednoznačnou výhodou tohoto přístupu je, že nám v sémantice stačí určit, co pro daný program znamená „skončil bez chyby". Zobecnění definice překladu na složitější formy programů navíc není nijak obtížné (máme-li k dispozici uspokojivou definici příslušného typu programu).

[13] Přesto, že v tomto předmětu budeme používat pouze podmnožinu jazyka C, není to na úkor obecnosti: tato podmnožina je dostatečně silná na to, aby do ní bylo možné přepsat libovolný program napsaný v plném ISO C99, a to včetně standardní knihovny. Také by bylo možné přímo jazyk ISO C99 překládat do strojového kódu stroje tiny.

[14] Viz také sekce B.2 sbírky.

Samozřejmě zatím nevíme, jaký mají takovéto stavy význam – tím se budeme zabývat za chvíli. Můžeme udělat ale jiné zajímavé pozorování – totiž kolik různých stavů takový výpočetní stroj má. Protože máme k dispozici pevný počet bitů, dva stavy se od sebe budou lišit v případě, že se liší alespoň v jednom bitu. To zejména znamená, že různých stavů bude **konečný počet**; velmi jednoduchá kombinatorická úvaha nás pak dovede k výsledku, že různých stavů je 2^{144} – i pro takto malinký stroj se jedná o velmi úctyhodný počet.
„Skutečný“ stroj tiny (ten, který budeme používat, resp. programovat) má 16bitový programový čítač, 16 16bitových registrů a 64 KiB paměti, tzn. jeho stav obsahuje 65570 bajtů resp. 524560 bitů informace. Různých stavů tedy existuje 2^{524560} což je přibližně 10^{157908}. Pro srovnání, celkový počet baryonů (hlavně protonů a neutronů) ve vesmíru se odhaduje na 10^{80}.

1.1.3 Akce Velmi důležitou vlastností takto definovaného stroje je, že libovolný stav **přesně určuje** celý následující výpočet – jinými slovy, náš výpočetní model je **deterministický**. Chování tohoto typu stroje je velmi jednoduché – je řízen sadou pravidel, podle které určíme jak z daného stavu odvodíme ten následovný.
Výpočet budeme obvykle začínat ve stavu, kdy jsou všechny registry nulové, ale obsah paměti nikoliv – počáteční obsah paměti budeme nazývat **programem**. S každým programem se tak váže právě jeden výpočet.[15]

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

1.1.4 Speciální stavy Výpočet může mít jeden ze dvou základních tvarů:

1. konečný výpočet, tvořen konečnou posloupností po dvou různých stavů, je obvyklá a žádoucí situace; může mít dvě varianty:
   a. úspěšný výpočet který byl ukončen instrukcí halt, přitom **výsledek** výpočtu jsme obvykle schopni odečíst z posledního stavu,
   b. neúspěšný výpočet, který byl ukončen chybou (viz také níže),
2. nekonečný výpočet, ve kterém se nějaká posloupnost stavů nekonečněkrát opakuje – je složen z konečného prefixu (kde se stavy neopakují) a nekonečné smyčky, celkově má tvar tzv. lasa.

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

Chyby, které mohou výpočet ukončit mohou být různých typů:

- nepodařilo se dekódovat instrukci, tzn. na adrese uložené v registru pc nezačíná platné kódování žádné instrukce,
- při vykonání instrukce došlo k fatální chybě (typickým příkladem je zde dělení nulou, případně neplatný přístup do paměti),
- selhalo **tvrzení** (angl. **assertion**) realizované instrukcí asrt – došlo k porušení programátorem určené vstupní nebo výstupní podmínky nebo invariantu,
- byla detekována sémanticky chybná operace, obvykle podle anotací vložených překladačem – z pohledu stroje by výpočet mohl bez problémů pokračovat, ale pravděpodobně by vedl k nesprávnému výsledku – situace, která se podobá na selhané tvrzení z předchozího bodu.

1.2: Instrukční sada

V dalším se budeme zabývat akcemi, které má stroj k dispozici. Abychom mohli popsat jejich **efekt**, musíme si ale nejprve upřesnit jak vypadají jednotlivé složky stavu a jaký mají význam.

1.2.1 Registry Nyní se na jednotlivé složky stavu podíváme blíže. Stroj tiny má 16 „obecných“ a jeden „speciální“ registr.
Obecné registry mají mnemonické názvy, které ale na výpočet nemají žádný vliv. Naznačují pouze typické použití (programy, které vzniknou překladem z jazyka C se budou tohoto konvenčního použití držet):

- rv od **return value** je registr určený pro předávání návratové hodnoty z podprogramu (více si o podprogramech povíme ve třetí kapitole),
- l1 až l7 jsou registry pro lokální proměnné a pro předávání parametrů podprogramům (jak později uvidíme, mezi formálním parametrem a lokální proměnnou není v jazyce C příliš velký rozdíl),
- t1 až t6 slouží pro dočasné hodnot – mezivýsledky, např. při výpočtu složených výrazů – při výpočtu a + b + c budeme potřebovat dočasně někam uložit výsledek a + b,
- bp a sp opět souvisí s podprogramy, konkrétněji se správou zásobníku.

Krom těchto 16 obecných registrů, které mohou být operandy libovolné aritmetické instrukce (a zároveň také jejich cílovým registrem), má stroj tiny ještě speciální registr

- pc (program counter, programový čítač), který obsahuje adresu ze které bude načtena, dekódována a provedena další instrukce.

- 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

[15] To může vypadat na pohled velmi nerealisticky – všichni zřejmě máme zkušenost, kdy spuštění téhož programu (v neformálním smyslu) vede k různým výsledkům. To je způsobeno jednak tím, že reálné počítače nejsou deterministické (reagují na vnější události) a také tím, že v reálných situacích často nemáme počáteční stav zcela pod kontrolou a to co se nám jeví jako tentýž stav je ve skutečnosti mnoho různých, ale těžko odlišitelných, stavů.

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

1.2.2 Paměť Stroj tiny disponuje 64 KiB paměti. Tím se myslí, že tato paměť je složená z 2^{16} buněk, přitom každá buňka je schopna uchovávat číslo od 0 do 255. Tyto buňky jsou navíc očíslované (mají adresy).
S paměťovými buňkami lze pracovat použitím instrukcí ld a st (více o nich ve čtvrté kapitole). Se kterou buňkou si přejeme pracovat určuje **hodnota uložená v registru**, tzn. adresa zejména může být výsledkem nějakého výpočtu. Všimněte si, že se jedná o zcela zásadní rozdíl oproti registrům – se kterými registry daná instrukce pracuje je její **pevnou součástí**.

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

1.2.3 Sémantika Krom **syntaxe** – toho, jak instrukce zapisujeme, potažmo kódujeme, nás bude zajímat jejich **sémantika** – význam. Co instrukce znamená budeme zejména popisovat tím, jaký má efekt na stav. Stav stroje jednoznačně určuje, jaká instrukce bude spuštěna – je to ta, které kódování je uloženo na adrese pc (každá instrukce je kódovaná do 4 bajtů, tzn. ve skutečnosti je uložena na adresách pc, pc + 1, pc + 2 a pc + 3).
Řada instrukcí bude mít podobné efekty. Zejména každá instrukce, která není instrukcí řízení toku, zvýší registr pc o 4, čím způsobí, že jako další bude spuštěna instrukce na nejbližší vyšší adrese.

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

1.2.4 Operandy Podobně obsahují téměř všechny instrukce nějaké **operandy**, které určují, s jakými daty bude operace pracovat. Typicky budou operandy **identifikátory registrů**. Počet a forma operandů se pro různé operace bude lišit, ale ve všech případech spadají do těchto mezí:

- součástí instrukce můžou být až 2 **vstupní registry** – operandy, které určí, ze kterých registrů se mají načíst vstupní hodnoty,
- instrukce může mít nejvýše jeden **výstupní registr** – operand, který určí, do kterého registru bude zapsán výsledek,
- instrukce může obsahovat nejvýše jeden **přímý operand** – 16bitové slovo, které je přímo součástí instrukce, a které se použije jako jedna ze vstupních hodnot (která to bude závisí od operace).

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

1.2.5 Kódování Instrukce stroje tiny mají velice jednoduché a pravidelné kódování – je to zejména proto, abychom byli v případě potřeby schopni instrukce dekódovat (alespoň přibližně) i „ručně“, z číselného výpisu obsahu paměti. Také se tím značně zjednodušuje implementace dekodéru (který máte k dispozici jako ukázku ve sbírce).
Instrukce jsou kódovány do dvou 16bitových slov, přitom:

1. horní slovo obsahuje:
   a. 8bitový kód operace v horní slabice,
   b. dva registrové operandy ve spodní slabice – výstupní registr v horní půlslabice a první vstupní registr v té spodní,
2. spodní slovo kóduje zbývající vstupní operand:
   - druhý vstupní registr (v nejvyšší půlslabice), nebo
   - přímý operand (využije celé spodní slovo).

Jak se dekódují operandy je určeno kódem operace, který je proto potřeba dekódovat jako první.

- 1× vstup + 1× výstup
- nastaví hodnotu registru
- copy $reg_1 \to reg_2$
- put $imm \to reg_1$

1.2.6 Kopírování hodnot Nejzákladnější operací, kterou můžeme v programu potřebovat, je nastavení registru, a to buď na předem známou konstantu, nebo na hodnotu aktuálně uloženou v některém jiném registru. K tomuto účelu můžeme použít operace copy (kopie mezi registry) a put (nastavení registru na konstantu).

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

1.2.7 Binární operace Binárních operací je k dispozici celá řada – odpovídají běžným aritmetickým a bitovým operacím, které z velké části již znáte. Patří sem:[16]

- aritmetika: sčítání add, odečítání sub, násobení mul, dělení (sdiv, udiv, smod, umod – rozdíly mezi verzemi s___ a u___ vysvětlíme později),
- srovnání – výsledkem je hodnota 0 nebo 1: rovnost eq, nerovnost ne, znaménkové porovnání slt, sgt, sle, sge, neznaménkové porovnání ult, ugt, ule, uge,
- bitové operace: logické and, or, xor a posuvy shl, shr, sar (aritmetický).

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

1.2.8 Operace řízení toku Do registru pc není možné běžnými instrukcemi přímo zapisovat ani z něj číst; efekt jmp addr je velmi podobný hypotetické instrukci put addr → pc, ale protože je tento efekt velmi odlišný od všech ostatních použití operace put, má svůj vlastní zápis.[17]

[16] Mnohem podrobněji je význam jednotlivých operací popsán ve sbírce.
[17] Tento přístup – striktně oddělovat instrukce řízení toku – je zcela běžný i u reálných procesorů.

Naproti tomu operace `jz` a `jnz` se na žádnou další známou instrukci nepodobají – jsou totiž **podmíněné** – jejich efekt se bude lišit podle hodnoty operandu. Uvažme instrukci `jz reg, addr` – tato instrukce se bude chovat jako `jmp addr` v případě, že je v registru `reg` uložena nula. V případě opačném bude výpočet pokračovat následující instrukcí. Jinými slovy:

- `jz reg, addr` nastaví `pc` na hodnotu přímého operandu `addr`, je-li v registru `reg` uložena nula,
- jinak zvýší hodnotu `pc` o 4.

Operace `jnz` je pak analogická, liší se pouze inverzí podmínky (skok se provede je-li registr nenulový).

1.2.9 Řízení stroje Jak jsme již zmiňovali dříve, za výstup programu budeme brát pouze formu jeho ukončení:

- výpočet může být úspěšně dokončen, což program signalizuje provedením instrukce `halt`, která nemá žádné operandy,
- výpočet může skončit chybou – tuto lze signalizovat instrukcí `asrt reg`, přitom chyba nastane pouze v případě, že v registru `reg` je uložena nula (v opačném případě výpočet pokračuje další instrukcí, tzn. registr `pc` se zvýší o 4) – jedná se o analogii tvrzení `assert` jak ho znáte z jazyka Python,
- výpočet může skončit jinou chybou (špatná instrukce, atp.) nebo se může „zacyklit“ – neskončí nikdy (v praxi budeme vždy délku výpočtu nějak uměle omezovat,[18] „nikdy“ je totiž velmi dlouhá doba).

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

1.3: Řízení toku

V poslední části první kapitoly si ukážeme, jak ve strojovém kódu zapsat standardní konstrukce řízení toku z vyšších programovacích jazyků – podmíněný příkaz a cyklus. Protože jediný vyšší programovací jazyk, který v tuto chvíli známe, je Python, budeme jej prozatím využívat jako pseudokód. V pozdějších kapitolách pak budeme používat jazyk C.

1.3.1 Konstrukce strojového kódu Jak již bylo zmíněno v kapitole B, překladem rozumíme zobrazení, kterého vstupem je nějaký program P, zatímco výstupem je nový program Q (typicky v jiném jazyce), přitom ale platí, že výpočet P končí úspěchem **právě když** výpočet Q končí úspěchem. Zobrazení je přitom definováno pouze pro **platné** vstupní programy.

Nyní jsme připraveni provést první náčrt toho, jak toto zobrazení **spočítáme** (jinými slovy, jak funguje překladač). Z předchozího studia[19] víte, že **výrazy** (zejména aritmetické, ale i libovolné jiné) můžeme reprezentovat pomocí stromů (ve smyslu datové struktury). Stejný přístup lze použít i pro příkazy a další prvky programovacích jazyků.

Jinak řečeno, zdrojové jazyky překladu mají typicky **rekurzivní strukturu**, kterou lze zachytit (abstraktním) **syntaktickým stromem**. Tento strom (angl. AST, z **abstract syntax tree**) je tak přesnou reprezentací **vstupního programu** a budeme jej považovat za startovní bod překladu.[20]

Jednoduchý překladač sestavuje strojový kód **rekurzivně**, podle struktury vstupního stromu. To zejména znamená, že překlad nějakého uzlu obdržíme vhodnou kombinací překladů jeho potomků.

V této kapitole si ukážeme pouze překlad uzlů, které odpovídají příkazům řízení toku – `if` a `while` – abychom získali představu, jaký je vztah mezi vstupním programem (tím, který typicky píšeme) a strukturou výstupního strojového kódu. Kompletní algoritmus překladu jazyka C[21] vybudujeme postupně (naleznete jej vždy ve sbírce, ve skriptech a přednášce vyzvedneme jen ty nejzajímavější části).

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

1.3.2 Podmíněný příkaz Použité instrukce:

- `jmp`, `jz`, `jnz`
- podmíněný skok realizuje **rozhodování**

- nejjednodušší forma: `if b:` $\mathtt{stmt}_1$
  - 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

[18] Jak již bylo zmíněno dříve, stroj `tiny` může v principu zacyklení spolehlivě detekovat (je to díky tomu, že konfigurace stroje má pevnou velikost – existuje tak pouze konečně mnoho různých konfigurací). V praxi může být takový cyklus velmi dlouhý a tedy těžce odhalitelný.

[19] Např. příklady z kapitoly 9 sbírky předmětu IB111, nebo mnohem podrobněji přednášky 11 a 12 téhož.

[20] Skutečné překladače tento strom konstruují ze vstupního textu. Touto částí překladu – syntaktickou analýzou – se zde zabývat nebudeme. Předpokládejme, že naším vstupem je již hotový abstraktní syntaktický strom.

[21] Resp. jeho podmnožiny, kterou budeme v tomto předmětu používat.

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

1.3.4 Cyklus `while`

Část 2: Základní prvky jazyka C

Tato kapitola se bude zabývat základními stavebními kameny jazyka C – hodnotami, proměnnými, výrazy a příkazy. U každé výpočetní konstrukce si zejména ukážeme, jak se abstraktní zápis na úrovni jazyka C realizuje sekvencemi instrukcí konkrétního výpočetního stroje.

2.1: Hodnoty, objekty, proměnné

V této části se budeme zabývat základními **datovými** prvky jazyka C – připomeneme si pojmy jako hodnota, objekt, proměnná nebo typ, které již znáte z jazyka Python, a zasadíme je do nového kontextu.

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

2.1.1 Hodnota Stejně jako v jazyce Python, fundamentálním předmětem výpočtu je v jazyce C **hodnota** – pro tuto chvíli se bude jednat o celé číslo v nějakém pevném rozsahu, nicméně v pozdějších kapitolách se setkáme i se složitějšími hodnotami.
Pozor, hodnota je **abstraktní** – hodnota **není** totéž co **reprezentace**. Zejména nesmíme hodnotám přisuzovat vlastnosti použité reprezentace. Zápisy 0x10, 16, šestnáct, XVI všechny reprezentují XXX tutéž XXX hodnotu.
[22]

Cf. Platón.

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

2.1.2 Operace Konceptuálně není výpočet ničím jiným, než mechanickou manipulací hodnot použitím kompatibilních **operací**. Klasickým příkladem operace je např. sčítání – vstupem jsou dvě celočíselné hodnoty a výstupem je nová hodnota. Aby bylo možné výpočet provést, je nutné si potřebné hodnoty **pamatovat** – při výpočtech ve středoškolské matematice k tomu používáme typicky papír a tužku. Počítač k tomu bude samozřejmě využívat nějaký typ elektronické **paměti**.

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

2.1.3 Objekt Přímá práce s pamětí a registry je pro větší programy značně nepohodlná – musíme při programování neustále pamatovat, kde máme uloženy které hodnoty (ve kterém registru nebo na jaké adrese), navíc jména registrů nejsou příliš popisná a je jich omezený počet.
Z jazyka Python jsme zvyklí hodnoty uchovávat v **proměnných**. Vazba mezi proměnnou a hodnotou ale není přímá – ani v Pythonu, ani v C. Hodnoty jsou totiž invariantní, anonymní entity – nemají identitu, ani schopnost se vnitřně měnit. Abychom obdrželi sémantiku, na kterou jsme při programování intuitivně zvyklí, musí do hry vstoupit ještě jeden prvek – **objekt**.
Hlavními vlastnostmi objektu jsou:

- schopnost uchovávat, číst a měnit **hodnotu** (abstraktní entitu programovacího jazyka),
- identita:
  - můžeme od sebe rozlišit různé objekty i v případě, že zrovna obsahují stejnou hodnotu,
  - jsme schopni určit, že se jedná o tentýž objekt, i když se hodnota v něm uložená během výpočtu změnila.

Objekt je tak zobecněním paměťové buňky – má operace „přečti“ a „ulož“ a jeho identita je obdobou adresy.

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

2.1.4 Proměnná Objekty mají sice identitu, ale nemají **jména** – jejich identita je více nebo méně abstraktní.[1] Abychom tedy mohli s objektem pracovat, potřebujeme mu přiřadit jméno – a to je přesně úloha **proměnné**.
Proměnná je (v jazyce C) pojmenovaný objekt, přičemž její jméno (identifikátor) má **syntakticky** omezený **rozsah platnosti**[23] – přímo ze zdrojového kódu umíme lehce identifikovat, ve kterých příkazech a výrazech je použití tohoto jména přípustné, a případně ke které deklaraci se váže. Vazba mezi jménem (proměnnou) a objektem je **pevná** – vznikne při deklaraci proměnné a až do jejího zániku tuto vazbu není lze měnit.[24]

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

2.1.5 Typ Různé hodnoty mají různé vlastnosti a různé operace. Uvažme 16bitové hodnoty bez znaménka $u_1 = 3, u_2 = 5$ a podobné (ale ne tytéž!) 16bitové hodnoty se znaménkem $s_1 = 3, s_2 = 5$. Zřejmě:

- $s_1 + s_2 = 8$, podobně $u_1 + u_2 = 8$,

[22] V standardní implementaci jazyka Python je každý objekt identifikovatelný adresou, na které je v paměti uložen. V jazyce C to ale neplatí, protože objekt nemusí mít adresu žádnou, nebo se jeho adresa může během výpočtu měnit.
[23] Známý též jako **lexikální** nebo **statický**, v kontrastu s **dynamickým**.
[24] Zde se objevuje důležitý rozdíl mezi C a Pythonem. Přiřazení v C, jak za chvíli uvidíme, značí **zápis do objektu**, kdežto v jazyce Python značí změnu vazby na **jiný objekt**.

- $s_2 - s_1 = -2$ ale $u_2 - u_1 = 65534$.

Je tedy potřeba podobné hodnoty rozlišovat – k tomu slouží **typy**. Typy mají v programovacím jazyce dvě funkce:

1. určí, jaké operace jsou pro dané hodnoty přípustné,
2. je-li operace přípustná, typy mohou ovlivnit její **význam** – např. existují dvě různé operace odečítání (viz výše) pro 16bitová čísla: znaménkové a neznaménkové.

Typ je **vlastnost hodnoty**, ale to neznamená, že je ke každé hodnotě „fyzicky" připojen její typ – řada programovacích jazyků, a C mezi nimi, typovou informaci uchovává pouze v **době překladu** – ve strojovém kódu bychom informaci o typech hledali marně.[25] To samozřejmě neznamená, že typy nemají pro výsledný strojový kód důsledky – bude na nich třeba záviset, jestli se pro srovnání použije operace `slt` nebo `ult`, atp.
Typy můžeme přisuzovat krom hodnot také objektům a skrze objekty také proměnným. Je-li objekt nějakého typu, znamená to, že je schopen uchovávat hodnoty pouze tohoto typu. Je-li proměnná nějakého typu, znamená to, že je svázána s objektem tohoto typu.[26]

2.1.6 Deklarace (jak vznikne proměnná, objekt, …)

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

2.2: Výrazy

Nyní známe základní datové (pasivní) prvky jazyka a můžeme se začít zabývat výpočetními (aktivními) – těmi nejzákladnějšími jsou **výrazy**.
(XXX denotační + operační sémantika)

2.2.1 Elementární výrazy (jméno proměnné, konstanta)
Pozor – číselný literál musí být platnou hodnotou příslušného typu: je-li typ `int` 16bitový, literál `0xffff` je chybný (tak velké číslo není možné reprezentovat). Naproti tomu, protože `0xffffu` je typu `unsigned`, je zde vše v pořádku.

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

2.2.2 Aritmetické a logické operace (operátory bez vedlejších efektů)

- 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 (strojový kód; „vyhodnocení do registru")

- 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 (je výraz typově správný?)

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

2.2.5 Implicitní konverze Aritmetické, bitové, atp. operace vyžadují, aby byly operandy stejného typu. Jazyk C zároveň nepodporuje operace na typech menších než `int` resp. `unsigned` (pro nás to znamená, že „jednobajtová" aritmetika neexistuje – všimněte si, že podobně neexistuje ani ve výpočetním stroji).
Povýšení: typy s rozsahem, který je menší než rozsah typu `int`, se nejprve zvětší na `int`.[27] Po povýšení se pak najde společný typ – je-li to možné, preferuje se znaménkový typ.[28]

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

[25] Tomuto konceptu se říká „vymazání typů", angl. „type erasure" – udržování informací o typech za běhu programu představuje dodatečnou režii a je-li to možné, překladače se tomu vyhýbají.
[26] Polymorfismus – schopnost objektu uchovávat různé typy hodnot – lze chápat např. tak, že takovému objektu přisoudíme součtový typ. V jazycích, kde je možné měnit vazbu mezi proměnnou a objektem může polymorfismus existovat jak na úrovni objektu (lze uložit různé typy hodnot) tak na úrovni proměnné (k jednomu jménu lze vázat různé typy objektů). To se ale nacházíme už mimo hranice tohoto předmětu.
[27] Rozsahy všech jednobajtových typů (`char`, `signed char`, `unsigned char`) jsou menší než rozsah typu `int`. Rozsah typu `int` není větší než rozsah typu `unsigned`, protože např. `40000` není přípustná hodnota typu `int` ale je to přípustná hodnota typu `unsigned`.
[28] V našem jazyce žádné znaménkové typy větší než `int` nejsou, proto se toto pravidlo neuplatní – společný typ je `unsigned` je-li alespoň jeden operand `unsigned`, jinak je to vždy `int`.

Pozor: povýšení se dotkne i unárních operátorů. Máme-li `signed char x = 5;`, bude hodnota výrazu `-x` typu `int`, nikoliv typu `signed char`!
V naší omezené verzi jazyka to dopadne takto (všechny případy jsou symetrické):

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

Pozor: na řádcích označených ! dochází ke konverzi operandu s menším rozsahem ve dvou krocích. Nejprve je rozšířen na typ `int` a poté až na společný `unsigned`. Zejména to znamená, že jednobajtové hodnoty jsou převedeny **znaménkovým rozšířením**. Např.:

```
signed char x = -3;         /*   0xfd */
unsigned y = 1;             /* 0x0001 */
assert( x + y == 65534u ); /* 0xfffe */
```

Součet je 65534 (`0xfffe`), nikoliv 254 (`0x00fe`) jak by mohl někdo čekat.

- $var = e_1$ (pozor na levou stranu)
- proběhne typová konverze pravé strany
- výraz s vedlejším efektem
- provede zápis do objektu
- hodnota je to, co bylo zapsáno

2.2.6 **Přiřazení** Prozatím budeme uvažovat pouze přiřazení tvaru $var = e_1$ – levá strana může být tedy pouze název proměnné. Pozor, vyhodnocení se bude pro složitější formy lišit!
Vyhodnocení tohoto typu přiřazení probíhá takto:

1. vyhodnotí se pravá strana,
2. výsledek se převede (konvertuje) na typ levé strany,
   - má-li typ levé strany větší nebo stejný rozsah, probíhá stejně jako konverze operandů aritmetických operátorů,
   - je-li levý operand menší a je neznaménkového typu, vyšší bity se implicitně ořízno,
   - je-li levý operand menší a je znaménkového typu, výsledek závisí na implementaci – program **může** spadnout, ale **nemá** nedefinované chování,
3. převedená hodnota se zapíše do objektu určeného levou stranou,
4. výsledná hodnota (přiřazení je výraz) je ta, která byla zapsaná (tzn. hodnota po konverzi z druhého bodu).

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

2.2.7 **Booleovské operace** (řízení toku)

- 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

2.2.8 **Vyhodnocení booleovských operací** (strojový kód)

2.3: Příkazy

Výrazy nám poskytují mocný výpočetní aparát, ale díky své deklarativní struktuře nejsou ideální pro popis sledu výpočetních kroků. Je-li potřeba provést nějakou sekvenci operací v daném pořadí, budou se mnohem lépe k zápisu hodit **příkazy**.

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

2.3.1 **Výrazový příkaz** (např. `a = b;`)

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

2.3.2 **Složený příkaz** (ve složených závorkách)

2.3.3 Podmíněný příkaz (if, else)

- if (expr) stmt
- if (expr) $stmt_1$ else $stmt_2$

2.3.4 Cyklus do … while (jump na konci)

- do stmt while (expr);

2.3.5 Řízení iterace (break, continue)

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

2.3.6 Cyklus while (while true + break)

- while (expr) stmt

2.3.7 Cyklus for (deklarace)

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

Část 3: Podprogramy

Programy mají dvě fundamentální složky:

1. pasivní datovou (hodnoty, na kterých je výpočet prováděn) a
2. aktivní řídící (kód, který výpočet řídí – určuje jaké operace, nad kterými hodnotami a v jakém pořadí se provedou).

Základní jednotkou organizace aktivní části programu – řízení výpočtu – je **podprogram**.

3.1: Podprogramy abstraktně

Nejprve se budeme zabývat podprogramem na abstraktní úrovni – jejich **významem** (sémantikou). To, jak se výpočet podprogramu realizuje na výpočetním stroji, si pak přiblížíme v další sekci.

3.1.1 **Účel** Podprogram je funkční celek, který popisuje nějaký výpočet – podobně jako samotný program. Tento výpočet je obvykle **parametrizován** – je možné podprogram používat (vykonávat, spouštět) opakovaně pro různé hodnoty **parametrů** a výsledek se může v závislosti na těchto parametrech měnit.
Výsledkem podprogramu jsou pak opět nějaké hodnoty – v nejjednodušším případě je pouze jediná a říkáme jí **návratová hodnota**. Díky této hodnotě je typicky použití (volání, spuštění, vykonání) podprogramu na syntaktické úrovni **výrazem**. Později (ve čtvrté kapitole) si ukážeme i jiné možnosti, zejména **výstupní parametry**.
Hlavní charakteristikou podprogramu je možnost jej používat opakovaně, a to zejména „staticky“ – stejný podprogram je možné použít v mnoha různých (nezávislých) výrazech.[29]

3.1.2 **Zápis** Abychom mohli o podprogramech mluvit, musíme být nejprve schopni je zapsat, a také zapsat jejich použití. Definice podprogramu (v jazyce C) sestává z:

1. hlavičky, která deklaruje

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

[29] Toto se může jevit jako naprostá samozřejmost, ale uvážíme-li, jak pracuje výpočetní stroj – zejména instrukce řízení toku – není na první pohled jasné, jak něčeho takového dosáhnout a zároveň mít strojový kód podprogramu v jediné kopii.

a. typ návratové hodnoty,
b. jméno podprogramu (musí být v celém programu[30] unikátní),
c. typy a jména formálních parametrů,
2. těla složeného z příkazů, které realizují výpočet podprogramu.

Chceme-li již definovaný podprogram **použít** (zavolat, vykonat), použijeme k tomu jeho jméno následované seznamem **skutečných parametrů** uzavřeným v kulatých závorkách.

3.1.3 **Vstupy a výstupy** Nyní nastala ta chvíle, kdy budeme nuceni konfrontovat otázku vstupů a výstupů. Protože podprogramy zavádíme na úrovni jazyka C, budeme se pro tuto chvíli bavit o vstupech pouze na této úrovni. Protože náš jazyk je v tuto chvíli velice omezený, vstupy a výstupy budou mít velmi jednoduchou formu:

1. **parametry** podprogramu jsou hodnoty, které v místě použití podprogramu explicitně uvádíme,
2. **návratová hodnota** je výsledek, který se při vyhodnocení výrazu „dosadí“ místo (již ukončeného) provedení podprogramu.

Krom explicitně uvedených parametrů je ale podprogram stále „uzavřený“ v tom smyslu, že jeho výsledek (návratová hodnota) nebude záviset na ničem dalším. Tento pohled se nám ovšem značně zamotá již příští týden.
Z výše uvedeného můžeme udělat možná poněkud překvapivý závěr – náš jazyk prozatím neumožňuje zapsat jiné podprogramy, než **čisté funkce**. K otázce vstupů, výstupů a čistoty funkcí se ale vrátíme hned v další kapitole.

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

3.1.4 **Sémantika** Omezíme-li se na čisté funkce, sémantika volání podprogramu je velmi jednoduchá – stačí ve výrazu celý podvýraz volání nahradit jeho výsledkem. V očekávání nových prvků jazyka ale takovýto denotační pohled není příliš uspokojivý a zvážíme tedy i jiný (operační), který blíže odpovídá tomu, jak se skutečně podprogramy chovají.
Předpokládejme (bez újmy na obecnosti), že program je zadán jako sbírka vzájemně se volajících podprogramů a jeden z nich je vyznačen jako „hlavní“ – jeho výpočet odpovídá výpočtu celého programu.[31]
Pak můžeme „výpočet programu“ stotožnit s „výpočtem hlavního podprogramu“, od této ekvivalence se odrazit, a zavést pojem „výpočet podprogramu“ (od výpočtu programu se liší pouze tím, že se musíme vypořádat s parametry – vstupem).
Nyní již můžeme přistoupit k popisu sémantiky volání (použití) podprogramu. Je-li během výpočtu potřeba získat výsledek takového volání:

1. hlavní výpočet v tomto místě **pozastavíme**,
2. poznačíme si hodnoty skutečných parametrů (které v tomto bodě již musí být známy),
3. spustíme **nový výpočet** zadaný **volaným podprogramem** (nad zadanými vstupy - skutečnými parametry),
4. poznačíme si výsledek tohoto vedlejšího výpočtu,
5. pokračujeme v hlavním výpočtu, přitom jako výsledek volání použijeme hodnotu z bodu 4.

Velice jednoduchým (i když ne zcela praktickým) způsobem, jak tohoto dosáhnout, je bod 3 realizovat na zcela novém výpočetním stroji (kterému jako program zadáme volaný podprogram). Jak později uvidíme, skutečné volání podprogramu se ale tomuto přístupu až podezřele podobá.

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

3.1.5 **Typy** (oddělený překlad)

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

3.1.6 **Parametry** (x)

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

3.1.7 **Rekurze**

[30] V programech složených z více překladových jednotek umožňuje jazyk C pojmenovávat podprogramy buď na úrovni programu (implicitně), nebo překladové jednotky (klíčovým slovem `static`). Ty druhé pak ale není možné přímo používat mimo tuto překladovou jednotku. V tomto předmětu nic z toho ale potřebovat nebudeme.

[31] Tento podrogram také jistě dobře znáte ze sbírky cvičení. Jmenuje se `main`. V našem zjednodušeném světě tento podprogram nebude mít žádné parametry – to odpovídá uzavřenosti programu jako celku.

3.2: Operační sémantika

3.2.1 Zásobník vstupní podmínka → není potřeba pamatovat si dno

- 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` „dole hlavou“

- `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

- $f(\ e_1, e_2, \ldots, e_n\ )$
  - 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

3.3: Rámec

3.3.1 Přelití registru (potřebujeme registr, žádný není volný)

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

3.3.2 Lokální proměnné (je-li jich víc než registrů)

- 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

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

3.3.5 Bookkeeping

Část 4: Ukazatele

Tato kapitola přináší první prostředek, který nám umožní s objekty pracovat **nepřímo** – rozhodnutí o tom, se kterým objektem budeme pracovat, provedeme **výpočtem** (za běhu programu). Zejména tedy konkrétní použitý objekt v pevně zvoleném příkazu může záviset na tom, s jakými vstupními daty byl program spuštěn.

4.1: Objekt, identita, adresa

Pojem **ukazatel** je těsně svázán s **objekty** a jejich **identitou**. Proto se nejprve vrátíme k těmto pojmům – bez nich není možné ukaztele správně zadefinovat.[32]

4.1.1 Hodnota Ukazatele jsou zejmena novým **typem hodnoty** – proto je potřeba mít na paměti, co je hodnota – prozatím jsme se v tomto kurzu setkali pouze s tzv. celočíselnými typy a jejich hodnotami. Může být tedy těžké oddělit koncept hodnoty od konceptu čísla, ale je to krok bezpodmínečně nutný pro hlubší pochopení výpočtu (a tedy i programování jako takového).
Jak jsme již zmiňovali ve druhé kapitole, výpočet je posloupnost **operací** (které s hodnotami **pracují**) nad sbírkou **objektů** (které hodnoty **uchovávají**). Abychom o něčem mohli říct, že je to hodnota, musí tedy vykazovat právě tyto dvě vlastnosti:

1. musí existovat nějaké **operace**, pro které jsou tyto hodnoty vstupy (operandy) a/nebo výstupy (výsledky),
2. hodnoty musí být možné **ukládat** (pamatovat si).

Naopak, není nutné, aby bylo možné libovolnou hodnotu přímo zapsat do programu (jako literál/konstantu), aby ji bylo možné „vypsat na obrazovku“ (vytvořit řetězec, který tuto hodnotu nějak reprezentuje) ani není určena žádná operace, která musí vždy existovat (některé hodnoty nemusí být možné sčítat, některé hodnoty nemusí být možné srovnávat, atp.).
Hodnoty také **nemají identitu** – není například možné říct, je-li kopie hodnoty (která vznikne načtením hodnoty z objektu a následným uložením do nového objektu) nějaká „nová hodnota“ nebo „tatáž hodnota“ – tuto otázku nedává smysl pokládat[33] – vztah „být tentýž“ na hodnotách vůbec není definovaný.[34]

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

4.1.2 Objekt Základním prvkem paměti programu v jazyce C (ale i mnoha jiných, včetně jazyka Python) je **objekt**. Pro pochopení ukazatelů není vůbec důležité, jak je takový objekt realizován – jestli a jak je reprezentován v paměti, jestli má nebo nemá nějakou adresu, atp.
Podobně jako je nutné vyčlenit pojem hodnoty, je také nutné rozlišovat mezi hodnotami a objekty a pochopit vztah mezi nimi.
Pro práci s objekty jsou důležité jen dvě vlastnosti – jeho **obsah** a jeho **identita** – vše ostatní je „implementační detail“ – podobně, jako při běžném programování nemusíte uvažovat o tom, jaké instrukce se použijí pro překlad kterého výrazu, nemusíme (a ani nechceme) uvažovat o tom, jak bude (nebo nebude) který objekt uložen v paměti.
Do této chvíle jsme s objekty pracovali pouze skrze proměnné – každá proměnná pevně identifikuje právě jeden objekt.[35] Typ tohoto objektu je stejný, jako typ proměnné, a je do něj možné uložit pouze hodnoty tohoto typu.

- identita je abstraktní
- ???

4.1.3 Identita objektu Zejména klíčový je pojem **identity** objektu.[36] Není důležité, jak taková identita vnitřně vypadá (podobně jako není důležité, jak jsou vnitřně reprezentovaná celá čísla), důležité ale je, že se jedná o **hodnotu**.

---

[32] Ukazuje se, že použití ukazatelů činí mnoha studentům značné problémy. Domníváme se, že je tomu zejména proto, že mají chybnou představu o tom, co ukazatel je. V obecném povědomí žel koluje mnoho mýtů a polopravd, a je pravděpodobné, že jste se s nimi už setkali. Zkuste prosím pro tuto chvíli zapomenout, že jste někdy něco o ukazatelích slyšeli, nebo je nějak používali. Ukazatele zejména nejsou nijak magické ani záhadné, jedná se o velice přímočarý koncept (i přesto, že jejich správné použití nemusí být vždy jednoduché).

[33] Jedná se o podobně nesmyslnou otázku jako „a mám přičíst tu jedničku, co mám v sešitě, nebo tu, co máte na tabuli?“

[34] **Pozor!** Jiná je **rovnost hodnot** – v tomto případě se jedná o **operaci** na hodnotách a neurčuje, zda se jedná o „tutéž“ hodnotu. Nakonec při vyhodnocení výrazu `a == a` dojde na levé straně k načtení hodnoty z objektu určeného proměnnou `a` a totéž se stane na pravé straně – jakákoliv intuice typu „je to přece tatáž hodnota, protože je uložena v tom stejném objektu“ ve skutečnosti také nefunguje.

[35] Platí pro jazyk C a některé jiné, ale zdaleka ne všechny. Vztah proměnných a objektů v Pythonu je například mnohem volnější.

[36] Spoiler alert: možné hodnoty ukazatelů jsou právě identity objektů.

Základní vlastností hodnoty je, že je možné ji ukládat (do objektů, a tedy i do proměnných) a že jsou na ní definované nějaké **operace**. Základní operace s identitami jsou dvě:

1. máme-li nějaký **objekt**, můžeme získat jeho identitu,
2. máme-li identitu, můžeme s její pomocí získat příslušný **objekt**,
3. máme-li dvě identity, můžeme zjistit, jestli jsou stejné nebo různé (na identitách je definovaná **rovnost**).

Protože různé objekty mají různé identity, a každý objekt má právě jednu identitu, rovnost identit přesně rozhoduje, identifikují-li obě tentýž objekt.

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 Adresa ~ reprezentace identity.

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

4.1.9 Operátor dereference `*` Identity `&*`, `*&`.

- 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

4.2: Přetypování

4.2.1 Přetypování aritmetických typů **4.2.2 Přetypování ukazatelů** **4.2.3 Typy a objekty** XXX vznik objektu zápisem do pole bajtů

Část 5: Pole

Tato kapitola se bude zabývat indexovanými kolekcemi objektů – koncept, který je pro výpočetní systémy (a tedy i jejich programování) zcela centrální. Indexace nám zejména umožňuje vybrat konkrétní objekt **výpočtem**.

5.1: XXX

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

5.1.1 Složené objekty

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

5.1.2 Pole jako paměť (po sobě jdoucí adresy slabik vs po sobě jdoucí ukazatele na podobjekty)

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

5.1.3 Syntaxe

5.1.4 Inicializace xxx

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

5.1.5 Operátor indexace

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

5.1.6 Ukazatelová aritmetika

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

5.1.7 Pasti

Část 6: Struktury, zřetězený seznam

Tato kapitola zejména uvede implementaci **zřetězeného seznamu** – datové struktury složené z buněk (uzlů) provázaných odkazy. K tomu budeme potřebovat **složené hodnoty** – uzel musí obsahovat jak hodnotu tak odkaz na další prvek a musí být tedy **složeného typu**. Složené typy v jazyce C zavádíme pomocí tzv. **struktur**. Konečně jednotlivé buňky budou uložené v poli a odkazy mezi nimi tak můžeme realizovat dvěma způsoby – indexy nebo ukazateli.

6.1: Struktury

Struktura je jazyková konstrukce, jenž nám umožní zavést do programu nový **složený typ**. Protože typ zavádíme – není součástí jazyka – říká se mu někdy i **uživatelský typ**.

6.1.1 Složená hodnota Než budeme diskutovat složené typy, připomeneme si pojem **složené hodnoty**. Jedná se o hodnotu, kterou lze chápat jako kombinaci několika jednodušších hodnot, a kterou můžeme smysluplně z takových jednodušších hodnot složit a opět je na ně rozložit.
Typickým příkladem složené hodnoty je uspořádaná dvojice (nebo obecněji n-tice), která vzniká kartézským součinem. Dvojici můžeme přímočaře vytvořit (konstruovat) ze dvou jednodušších hodnot a díky projekcím ji můžeme také jednoduše rozložit na dvě jednodušší hodnoty.
Typům, které zastřešují hodnoty tohoto charakteru, proto říkáme **součinové typy**. Součinové typy mají charakteristické operace: podobně, jako hodnoty aritmetických typů můžeme sčítat nebo násobit, hodnoty součinových typů můžeme:

- skládat (konstruovat) z jejich jednotlivých složek,
- vytvářet nové hodnoty nahrazením některé složky za novou,
- zjistit hodnotu některé složky (projekce).

- 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 Abychom mohli s hodnotami v programech pracovat, musíme být schopni nějak si je pamatovat – a u složených hodnot tomu není jinak. Pro uložení **složené hodnoty** se v jazyce C použije **složený objekt**, a to takový, že každé **složce** hodnoty odpovídá **podobjekt** příslušného typu (vzpomeňte si, že typ objektu a typ v něm uložené hodnoty se musí shodovat, a že podobjekt je také sám o sobě plnohodnotným objektem).
Díky tomuto uspořádání můžeme složené hodnoty přímo **upravovat po složkách**, pomocí přiřazení do podobjektu (to, co budeme v jazyce C zapisovat přibližně jako `X.i = N`). Protože podhodnoty a podobjekty se na sebe mapují 1:1, je zaručeno, že výsledek přiřazení do podobjektu je tentýž, jako „kanonická“ posloupnost operací:

1. načti složenou hodnotu V z objektu X,
2. vytvoř novou složenou hodnotu W výměnou i-té složky ve V, tzn. $W = (V_1, ..., V_{i-1}, N, V_{i+1}, ..., V_n)$,
3. takto vzniklou hodnotu W ulož zpátky do objektu X.

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

6.1.4 Záznamový typ Záznamový typ – struktura – je zřejmě nejběžnějším prostředkem k zavedení součinových typů do programovacího jazyka. Vytvoříme-li příslušnými jazykovými prostředky nový součinový typ, automaticky tím získáme:

1. možnost konstruovat hodnoty tohoto typu tak, že dodáme hodnotu pro každou složku,
2. možnost vytvářet nové hodnoty z již existujících tak, že změníme hodnotu některé složky,
3. možnost získat hodnotu libovolné složky.

Je zároveň relativně běžné (i mimo jazyk C), že bod 2 je realizován skrze přiřazení do podobjektu, jak bylo diskutováno výše. Z hlediska výrazových možností jazyka jsou obě možnosti ekvivalentní.

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

6.1.5 Struktura Nyní si stručně popíšeme syntaxi záznamových typů v jazyce C. Nový záznamový typ zavedeme klíčovým slovem `struct` například takto:

```
struct pair
{
    int a;
    int b;
};
```

Hodnoty uvedeného typu budou mít dvě složky, obě typu `int`, tzn. budou to celá čísla se znaménkem. Identifikátor `pair` není sám o sobě názvem typu, musíme ho nadále uvádět uvozený klíčovým slovem `struct`. Deklarace proměnné typu `struct pair` tedy vypadá například takto:

```
int main()
{
    struct pair var;
}
```

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

6.1.6 Inicializace Podobně jako u jednoduchých objektů, není-li hodnota objektu inicializovaná, není dovoleno z objektu číst.[37] Inicializaci záznamové proměnné zapisujeme složenými závorkami. Jména složek jsou při inicializaci nepovinná:

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

[37] Opět zde ale platí, že inicializovaný **podobjekt** je možné používat bez ohledu na to, je-li inicializovaný objekt jako celek.

```
int main()
{
    struct pair foo = { .a = 1, .b = 2 },
                bar = { -1, 3 };
}
```

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

```
struct array { int data[ 3 ]; }

int f( struct array a )
{
    assert( a.data[ 0 ] == 1 );
}

int main()
{
    struct array arr = { { 1, 2, 3 } };
    f( arr );
}
```

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

6.2: Zřetězený seznam

Nejjednodušší dynamická datová struktura – taková, která umožňuje uložit a později zpracovávat předem neznámý počtem prvků – je zřetězený seznam. Klíčovou vlastností datových struktur obecně je, že existují na úrovni **objektů** (nikoliv hodnot). Zřetězený seznam (a později i další datové struktury) tedy budeme zejména chápat jako vzájemně provázaný **soubor objektů**.

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

Tato kapitola se bude zabývat správou buněk – navážeme na zřetězené seznamy, o kterých byla řeč minule, a zaměříme se na to, jak udržovat informaci o tom, které buňky jsou používané – **alokované** (patří nějaké datové struktuře) a které jsou naopak **volné** – je možné je využít při stavbě nové, nebo rozšíření některé existující struktury.

1 Definice problému Dynamická alokace řeší následovný problém:

1. program potřebuje během výpočtu **postupně** vytvářet nové objekty (aby do nich mohl ukládat nějaké hodnoty),
2. **celkový** počet objektů může být velmi velký,
3. jednotlivé objekty jsou ale používány pouze po **omezenou dobu**.

Typicky bude zejména nastávat situace, kdy výpočet potřebuje jen relativně malý počet objektů **najednou** a bylo by tedy výhodné objekty využívat opakovaně – zejména tedy budeme hledat způsob, jak **již nepotřebné** objekty využít pro nové hodnoty.

Pro tuto chvíli se navíc omezíme na situace, kdy potřebujeme dynamicky pracovat s objekty jediného typu a tyto budou všechny uloženy v jediném poli. Předpokladem úspěšného výpočtu přirozeně bude, že toto pole má alespoň tolik položek, kolik hodnot potřebuje výpočet uložit najednou.

Část 8:

1 Správa paměti

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 $O(f(n))$
- n bude velikost problému
  - nemusí nutně být pouze vstup
- alokace na zásobníku – $O(1)$ OK
  - $O(\log n)$ – obvykle OK

8.1.3 Jazyk C

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

8.1.4 Rekurze

- podobné omezení – max. $O(\log n)$
- 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 $O(\log n)$
- 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

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 životi 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

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

9.2: Implementace

9.2.1 Dynamické pole

- zřetězený seznam → `get` je $O(n)$
- vyhledávací strom → `get` je $O(\log n)$
- 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

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

9.2.3 Organizace v paměti

3 položky: ◦ méně nepřímých přístupů ◦ více průhledné pro optimalizaci

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

9.2.4 Reprezentace položek

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

9.2.5 Platnost ukazatelů

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

9.2.6 Amortizace

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

9.2.7 Taktika 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

9.2.8 Exponenciální zvětšování

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

9.2.9 Další využití

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

9.2.10 Zmenšování

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

9.2.11 Využití paměti

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

9.2.12 Srovnání

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

9.2.13 Režie alokátoru

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

9.2.14 Efektivita v praxi

9.3: Binární halda

- vlastnost haldy
- sift-up, sift-down
- uložení v poli
- dijkstra → decrease-key vs reinsert

Část 10: Slovníky a množiny

10.1: Hašovací tabulky

- hašovací fce → krypto, nekrypto
- organizace: seznam vs probing
- složitost
- zvětšování (rehashing)

10.2: Vyhledávací stromy

Část 11: Paměť

11.1: Objekty

11.1.1 Připomenutí: objekt (má-li objekt adresu, může na něj existovat ukazatel; existuje-li ukazatel, objekt musí mít adresu)

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

11.1.2 Paměť (paměť je efektivně pole bajtů, adresy hrají roli indexů; na úrovni C odpovídá bajtu – nejmenší adresovatelné jednotce – typ unsigned char)

- „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

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

11.1.8 Ukazatel bez typu

11.2: Alokace

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

11.2.1 Zadání úlohy

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

11.2.2 Dynamická paměť

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

11.2.3 Další požadavky

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

11.2.4 Strategie

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

11.2.5 Předalokace

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

11.2.6 Datové struktury

11.3: Realokace paměti

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

11.3.1 Situace

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

11.3.2 Využití stávající alokace

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

11.3.3 Využití předchozího bloku

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

11.3.4 Přesun dat

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

Část 12: Vyhledávací stromy

- vyhledávací vlastnost
- stavba ze seřazeného pole
- self-balancing
- scapegoat