Zápis programu v C a základní řídicí struktury

Jan Faigl

Katedra počítačů
Fakulta elektrotechnická
České vysoké učení technické v Praze

Přednáška 03

B0B36PRP – Procedurální programování

Přehled témat

Část I

Část 1 – Zápis programu v C

Program je „recept“

- Program je posloupnost kroků (výpočtů) popisující průběh výpočtu pro řešení problému

  (je to „recept“ na řešení problému)
- Pro zápis receptu potřebujeme jazyk

  Způsob zápisu programu
- Jazyk definuje základní sadu primitiv (operací/příkazů), které můžeme použít pro zápis receptu
- S konečnou množinou primitiv dobrý programátor naprogramuje „cokoliv“.

  Co může být vyjádřeno – Turing Machine (obecný model počítacího stroje (Alan Turing, 1939).
- V předmětu B0B36PRP používáme programovací jazyk **C**

  Programování není jen o znalosti konkrétního programovacího jazyka, je to o způsobu uvažování a řešení problému. Jazyk C nám dává příležitost osvojit si základní koncepty, které lze využít i v jiných jazycích.

Zdrojové soubory programu v C

- **zdrojový** soubor s koncovkou **.c**

  Zpravidla—základní rozlišení souborů, pozor na .C
- **hlavičkový** soubor s koncovkou **.h**

  Jména souborů volíme výstižné (krátké názvy) a zpravidla zapisujeme malými písmeny.
- Zdrojové soubory jsou překládány do binární podoby překladačem a vznikají objektové soubory (**.o**)

  Objektový kód obsahuje relativní adresy proměnných a volání funkcí nebo pouze odkazy na jména funkcí, jejichž implementace ještě nemusejí být známy.
- Z objektových souborů (object files) se sestavuje výsledný program, ve kterém jsou již všechny funkce známy a relativní adresy se nahradí absolutními.

  Program se zpravidla sestavuje z více objektových souborů umístěných například v knihovnách.

Definice programovacího jazyka

- **Syntax** – definice povolených výrazů a konstrukcí programu

  Plná kontrola a podpora vývojových prostředí
  - Příklad popisu výrazu gramatikou v Backus-Naurově formě
    <exp> ::= <exp> + <exp> | <exp> * <exp> | <exp> |
    <number> <number> ::= <number> <digit> | <digit>
    <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
- **Statická sémantika** – definuje jak jsou konstrukty používány

  Částečná kontrola a podpora prostředí
  - Příklad axiomatické specifikace: $\{P\}\ S\ \{Q\}$, P- precondition, Q-postcondition, S - konstrukce jazyka.
- **Plná sémantika** – co program znamená a dělá, jeho smysluplnost

  Kontrola a ověření správnosti je kompletně v režii programátora.

Platné znaky pro zápis zdrojových souborů

- Malá a velká písmena, číselné znaky, symboly a oddělovače

  ASCII – American Standard Code for Information Interchange
  - a–z A–Z 0—9
  - ! " # % & ' ( ) * + , - . / : ; < = > ? [\ ] ^ _ { | } ~
  - mezera, tab, nový řádek
- Escape sekvence pro symboly
  - \' – ', \" – ", \? – ?, \\– \
- Escape sekvence pro tisk číselných hodnot v textovém řetězci
  - \o, \oo, kde o je osmičková číslice
  - \xh, \xhh, kde h je šestnáctková číslice

```
1 int i = 'a';
2 int h = 0x61;
3 int o = 0141;
4
5 printf("i: %i h: %i o: %i c: %c\n", i, h, o, i);
6 printf("oct: \141 hex: \x61\n");
```

Např. \141, \x61 lec03/esqdho.c
- \0 – znak pro konec textového řetězce (null character)

Správnost programu

- Syntakticky i staticky sémanticky správný program neznamená, že dělá to co od něj požadujeme
- Správnost a smysluplnost programu je dána očekávaným chováním při řešení požadovaného problému
- V zásadě při spuštění programu mohou nastat tyto události:
  - Program havaruje a dojde k chybovému výpisu

    Mrzuté, ale výpis (report) je dobrý start řešení chyby (bug)
  - Program běží, ale nezastaví se a počítá v nekonečné smyčce.

    Zpravidla velmi obtížné detekovat a program ukončujeme po nějaké době, proto je vhodné mít představu o výpočetní náročnosti řešené úlohy a použitém přístup řešení (algoritmu).
  - Program včas dává odpověď

    Je však dobré vědět, že odpověď je korektní.

Správnost programu je plně v režii programátora, proto je důležité pro snadnější ověření správnosti, ladění a hledání chyby používat **dobrý programovací styl**.

Identifikátory

- **Identifikátory** jsou jména proměnných, funkcí, a vlastních typů

  Vlastní typy a funkce viz další přednášky
- Pravidla pro volbu identifikátorů

  Názvy proměnných, typů a funkcí
  - Znaky a–z, A–Z, 0–9 a _
  - První znak není číslice
  - Rozlišují se velká a malá písmena (case sensitive)
  - Délka identifikátoru není omezena

    Prvních 31 znaků je významných – může se lišit podle implementace
- Klíčová (rezervovaná) slova (**keywords**)$_{32}$

  auto break case char const continue default do double else enum extern float for goto if int long register return short signed sizeof static struct switch typedef union unsigned void volatile while C98

C99 dále rozšiřuje například o `inline`, `restrict`, `_Bool`, `_Complex`, `_Imaginary`
C11 pak dále například o `_Alignas`, `_Alignof`, `_Atomic`, `_Generic`, `_Static_assert`, `_Thread_local`

Funkce

- Funkce tvoří základní stavební blok **modulárního** jazyka C

  Modulární program je složen z více modulů/zdrojových souborů
- Každý spustitelný program v C obsahuje *alespoň* jednu funkci a to funkci `main()`
  - Běh programu začíná funkcí `main()`
- **Deklarace** se skládá z hlavičky funkce

```
typ_návratové_hodnoty jméno_funkce(seznam parametrů);
```

  C používá **prototyp (hlavičku) funkce** k **deklaraci** informací nutných pro překlad tak, aby mohlo být přeloženo správné volání funkce i v případě, že **definice** je umístěna dále v kódu
- **Definice** funkce obsahuje **hlavičku funkce a její tělo**, syntax:

```
typ_návratové_hodnoty jméno_funkce(seznam parametrů)
{
    //tělo funkce }
```

Definice funkce bez předchozí deklarace je zároveň deklarací funkce

Vlastnosti funkcí

- C nepovoluje funkce vnořené do jiných funkcí
- Jména funkcí se mohou exportovat do ostatních modulů

  Modul–samostatně překládaný soubor
- Funkce jsou implicitně deklarovány jako **extern**, tj. viditelné
- Specifikátorem **static** před jménem funkce omezíme viditelnost jména funkce pouze pro daný modul

  Lokální funkce modulu
- Formální parametry funkce jsou **lokální proměnné**, které jsou inicializovány skutečnými parametry při volání funkce

  *Parametry se do funkce předávají **hodnotou** (call by value)*
- **C dovoluje rekurzi** – lokální proměnné jsou pro každé jednotlivé volání zakládány znovu na zásobníku

  Kód funkce v C je reentrantní ve smyslu volání funkce ze sebe sama.
- Funkce nemusí mít žádné vstupní parametry, zapisujeme:

  `fce(void)`
- Funkce nemusí vracet funkční hodnotu–návratový typ je `void`

  `lec03/function.c`

Struktura programu / modulu

```
1  #include <stdio.h> /* hlavickovy soubor */
2  #define NUMBER 5 /* symbolicka (textova) konstanta */
3
4  int compute(int a); /* hlavicka/prototyp funkce */
5
6  int main(int argc, char *argv[])
7  { /* hlavni funkce */
8     int v = 10; /* definice promennych */
9     int r;
10    r = compute(v); /* volani funkce */
11    return 0; /* ukonceni hlavni funkce */
12 }
13
14 int compute(int a)
15 { /* definice funkce compute */
16   int b = 10 + a; /* telo funkce */
17   return b; /* navratova hodnota funkce */
18 }
```


Příkaz `return`

- Příkaz ukončení funkce `return vyraz;`
- `return` lze použít pouze v těle funkce
- `return` ukončí funkci, vrátí návratovou hodnotu funkce určenou hodnotou `vyraz` a předá řízení volající funkci
- `return` lze použít v těle funkce vícekrát

  Kódovací konvence však může doporučovat nejvýše jeden výskyt return ve funkci.
- U funkce s prázdným návratovým typem, např. `void fce()`, nahrazuje uzavírací závorka těla funkce příkaz `return;`

```
void fce(int a) {
   ...
}
```


Zápis hodnot datových typů

- Hodnoty datových typů označujeme jako **literály**
- Zápis celých čísel (celočíselné literály)
  - dekadický 123 450932
  - šestnáctkový (hexadecimální) 0x12 0xFAFF (začíná `0x` nebo `0X`)
  - osmičkový (oktalový) 0123 0567 (začíná `0`)
  - unsigned 12345U (přípona `U` nebo `u`)
  - long 12345L (přípona `L` nebo `l`)
  - unsigned long 12345ul (přípona `UL` nebo `ul`)
- Není-li přípona uvedena, jde o literál typu `int`
- Neceločíselné datové typy jsou dané implementací, většinou se řídí standardem IEEE-754-1985

  `float`, `double`

Literály

- Jazyk C má 6 typů literálů (konstantních hodnot)
  - Celočíselné
  - Racionální
  - Znakové
  - Řetězcové
  - Výčtové – *pojmenovaná celá čísla typu* `int`

    Enum
  - Symbolické – `#define NUMBER 10`

    Preprocesor

Literály racionálních čísel

- Formát zápisu racionálních literálů:
  - S řádovou tečkou – `13.1`
  - Mantisa a exponent – `31.4e-3` nebo `31.4E-3`
- Typ racionálního literálu:
  - `double` – pokud není explicitně určen
  - `float` – přípona `F` nebo `f`

    `float f = 10f;`
  - `long double` – přípona `L` nebo `l`

    `long double ld = 10l;`

Znakové literály

- Formát – jeden (případně více) znaků v jednoduchých apostrofech

  `'A'`, `'B'` nebo `'\n'`
- Hodnota – jednoznakový literál má hodnotu odpovídající kódu znaku

  `'0'`∼ 48, `'A'`∼ 65

  Hodnota znaků mimo ASCII (větší než 127) závisí na překladači.
- Typ znakové konstanty
  - **znaková konstanta je typu** `int`

Řetězcové literály

- Formát – posloupnost znaků a řídicích znaků (escape sequences) uzavřená v uvozovkách

  `"Řetězcová konstanta s koncem řádku\n"`
  - Řetězcové konstanty oddělené oddělovači (white spaces) se sloučí do jediné, např.

    `"Řetězcová konstanta"  "s koncem řádku\n"`

    se sloučí do

    `"Řetězcová konstanta s koncem řádku\n"`
- Typ
  - Řetězcová konstanta je uložena v poli typu `char` a zakončená znakem `'\0'`

    Např. řetězcová konstanta `"word"` je uložena jako

`'w'`	`'o'`	`'r'`	`'d'`	`'\0'`

Pole tak musí být vždy o 1 položku delší!

Více o textových řetězcích na 4. přednášce a cvičení

Konstanty výčtového typu

- Formát
  - Implicitní hodnoty konstanty výčtového typu začínají od 0 a každý další prvek má hodnotu o jedničku vyšší
  - Hodnoty můžeme explicitně předepsat

```
enum {                    enum {
   SPADES,                   SPADES = 10,
   CLUBS,                    CLUBS, /* the value is 11 */
   HEARTS,                   HEARTS = 15,
   DIAMONDS                  DIAMONDS = 13
};                        };
```

Hodnoty výčtu zpravidla píšeme velkými písmeny

- Typ – výčtová konstanta je typu `int`
  - Hodnotu konstanty můžeme použít pro iteraci v cyklu

```
enum { SPADES = 0, CLUBS, HEARTS, DIAMONDS, NUM_COLORS };
for (int i = SPADES; i < NUM_COLORS; ++i) {
   ...
}
```


Symbolické konstanty – `#define`

- Formát – konstanta je založena příkazem preprocesoru `#define`
  - Je to makro příkaz bez parametru
  - Každý `#define` musí být na samostatném řádku

```
#define SCORE 1
```

Zpravidla píšeme velkými písmeny

- Symbolické konstanty mohou vyjadřovat konstantní výraz

```
#define MAX_1   ((10*6) - 3)
```

- Symbolické konstanty mohu být vnořené

```
#define MAX_2   (MAX_1 + 1)
```

- **Preprocesor provede textovou náhradu definované konstanty za její hodnotu**

```
#define MAX_2   (MAX_1 + 1)
```

Je-li hodnota výraz, jsou kulaté závorky nutné pro správné vyhodnocení výrazu, např. pro `5*MAX_1` s vnějšími závorkami je `5*((10*6) - 3)=285` vs `5*(10*6) - 3=297`.

Proměnné s konstantní hodnotou modifikátor (`const`)

- Uvedením klíčového slova **const** můžeme označit proměnnou jako konstantní

Překladač kontroluje přiřazení a nedovolí hodnotu proměnné nastavit znovu.

- Pro definici konstant můžeme použít např.
- Proměnné s konstatní hodnotou mají typ

```
const float pi = 3.14159265;
```

- na rozdíl od symbolické konstanty

```
#define PI 3.14159265
```

- proto může překladač provádět pro proměnnou s konstantní hodnotou **typovou kontrolu**

Část II

Řídicí struktury

Výpočetní problém, algoritmus a program jako jeho řešení

Příklad: Najít největšího společného dělitele čísel 6 a 15.

- Víme co musí platit pro číslo d, aby bylo největším společným dělitele čísel x a y
- Známou deklarativní znalost o problému můžeme využít pro návrh výpočetního postupu jak takové číslo najít, např.
  1. Nechť máme nějaký odhad čísla d
  2. Potom můžeme ověřit, zdali d splňuje požadované vlastnosti
  3. Pokud ano, jsme u cíle
  4. Pokud ne, musíme d vhodně modifikovat a znovu testovat
- Výpočetní problém chceme vyřešit využitím konečné množiny primitivních operací počítače
- Konkrétní úlohu pro čísla 6 a 15 zobecňujeme pro „libovolná“ čísla x a y, pro který **navrhneme algoritmus**
- Algoritmus následně přepíšeme do programu využitím konkrétního programovacího jazyka

Algoritmus a program

- Algoritmus je postup řešení třídy problému
- Algoritmus je recept na výpočetní řešení problému
- Program je implementací algoritmu s vyžitím zápisu příkazů programovacího jazyka
- Program je posloupnost instrukcí počítače

- Předpokládáme, že náš problém lze výpočetně řešit a je výpočetně zvladatelný

Naše problémy (výzvy) na PRP takové jsou, v praktických úlohách tomu však vždycky být nemusí a můžeme narážet na problém jak úlohu vůbec formulovat či problém potřebného výpočetního výkonu.

Program jako algoritmus

- Program je implementací (realizací) algoritmu
- Algoritmus je posloupnost kroků vedoucí k řešení určité třídy úloh, je to syntetický postup řešení obecných úloh
- Vlastnosti algoritmu:
  - **hromadnost** a **univerzálnost** – řešení třídy úloh

Měnitelná vstupní data

  - **determinovanost** – každý krok jednoznačně definován
  - **konečnost** – pro přípustná data v konečné době skončí
  - **rezultativnost** – vždy vrátí výsledek (třeba chybu)
  - **korektnost** – výsledek je správný
  - **opakovatelnost** – stejný vstup vede na stejný výstup
- Prostředky pro zápis algoritmu
  - Přirozený jazyk
  - Vývojové diagramy
  - Strukturogramy, pseudojazyk, programovací jazyk

Výpočetní, algoritmické a programové řešení problému

- Množina primitivních instrukcí počítače je relativně malá a zahrnuje následující operace:
  - **Práce s číselnými hodnotami** v operační paměti počítače (**proměnné**)

Odkazované jmény definovaných proměnných

  - **Výpočetní operace** (výrazy, které zahrnují též funkce)

Unární nebo binární operace, tj. čtení jednoho nebo dvou číselných hodnot z paměti, provedení operace a zápis výsledku do operační paměti.

  - **Testování hodnot proměnných** – podmínky a větvení výpočtu

Pokud podmínka platí, vykonej instrukci, jinak udělej něco jiného nebo nedělej nic.

  - **Skoky** na provedení konkrétní posloupnosti instrukcí v závislosti na splnění podmínky nebo posloupnosti operace (*function call*, `return`)

„Program Counter“ (PC) jako ukazatel z jaké adresy v paměti čte počítač instrukce pro vykonání

- Tyto instrukce se objevují ve své abstraktní podobě
  - v zápisu algoritmu např. jako bloky vývojového diagramu
  - v zápisu programu jako příkazy a vyhrazená klíčová slova

Příklad největší společný dělitel

- Úloha:

  Najděte největší společný dělitel čísel 6 a 15.

Co platí pro společného dělitele čísel?

- Řešení

  Návrh postupu řešení pro dvě libovolná přirozená čísla

*Definice **vstupu** a **výstupu** algoritmu*

  - Označme čísla x a y
  - Vyberme menší z nich a označme jej d
  - Je-li d společným dělitelem x a y končíme
  - Není-li d společným dělitelem pak zmenšíme d o 1 a opakujeme test až d bude společným dělitelem x a y

- Symboly x, y a d reprezentují **proměnné** (paměťové místo), ve kterých jsou uloženy hodnoty, které se v průběhu výpočtu mohou měnit.

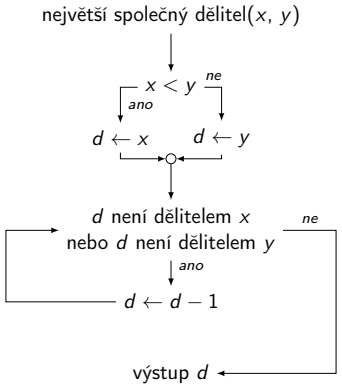
Slovní popis činnosti algoritmu

- Úloha:

  Najít největší společný dělitel přirozených čísel x a y.
- Popis řešení
  - **Vstup**: dvě přirozená čísla x a y
  - **Výstup**: přirozené číslo d – největší společný dělitel x a y
  - **Postup**
    1. Je-li $x < y$, pak d má hodnotu x, jinak má d hodnotu y
    2. Pokud d není dělitelem x nebo d není dělitelem y opakuj krok 3, jinak proveď krok 4
    3. Zmenši d o 1
    4. Výsledkem je hodnota d

Algoritmus = výpočetní postup jak zpracovat vstupní data a určit (vypočítat) požadované výstupní hodnoty (data) s využitím elementárních výpočetních instrukcí a pomocných dat.

Postup výpočtu algoritmu vyjádřený formou vývojového diagramu



Zápis algoritmu v pseudojazyku

- Zápis algoritmu využitím klíčových a dobře pochopitelných slov

Algoritmus 1: Nalezení největšího společného dělitele

Vstup: x, y – kladná přirozená čísla
Výstup: d – největší společný dělitel x a y

if $x < y$ **then**
  $d \leftarrow x$;
else
  $d \leftarrow y$;
while *d není dělitelem x nebo d není dělitelem y* **do**
  $d \leftarrow d - 1$;
return d

Neodpovídá přesně zápisu programu v konkrétním programovacím jazyku, ale je čitelný a lze velmi snadno přepsat.

Zápis algoritmu v C – motivační ukázka

```
1  int getGreatestCommonDivisor(int x, int y)
2  {
3     int d;
4     if (x < y) {
5        d = x;
6     } else {
7        d = y;
8     }
9     while ( (x % d != 0) || (y % d ! = 0)) {
10       d = d - 1;
11    }
12    return d;
13 }
```

- Nebo také s využitím ternárního operátoru

  podmínka ? výraz : výraz

```
1  int getGreatestCommonDivisor(int x, int y)
2  {
3     int d = x < y ? x : y;
4     while ( (x % d != 0) || (y % d ! = 0)) {
5        d = d - 1;
6     }
7     return d;
8  }
```

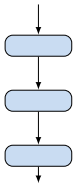
lec03/demo-gcd.c

Řídicí struktury

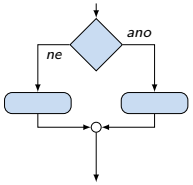
- Řídicí struktura je programová konstrukce, která se skládá z dílčích příkazů a předepisuje pro ně způsob provedení
- Tři základní druhy řídicích struktur
  - **Posloupnost** – předepisuje postupné provedení dílčích příkazů
  - **Větvení** – předepisuje provedení dílčích příkazů v závislosti na splnění určité podmínky
  - **Cyklus** – předepisuje opakované provedení dílčích příkazů v závislosti na splnění určité podmínky

Typy řídicích struktur 1/2

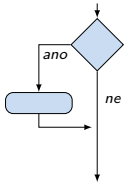
- **Sekvence**



- Podmínka **If**

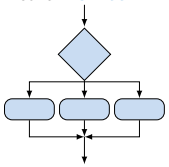


- Podmínka **If**

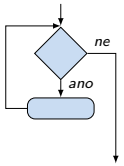


Typy řídicích struktur 2/2

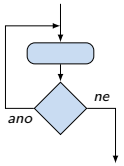
- Větvení **switch**



- Cyklus **for** a **while**



- Cyklus **do**



Složený příkaz a blok

- Řídicí struktury mají obvykle formu strukturovaných příkazů:
  - **Složený příkaz** – posloupnost příkazů
  - **Blok** – posloupnost definic proměnných a příkazů

```
{
   //blok je vymezen složenými závorkami
   int steps = 10;

   printf("No. of steps %i\n", steps);
}

steps += 1; //nelze - mimo rozsah platnosti bloku
```

Definice – alokace paměti podle konkrétního typu proměnné. Rozsah platnosti proměnné je lokální v rámci bloku.

- Budeme používat složené příkazy:
  - složený příkaz nebo blok pro posloupnost
  - příkaz **if** nebo **switch** pro větvení
  - příkaz **while**, **do** nebo **for** pro cyklus

Podmíněné opakování bloku nebo složeného příkazu

- Funkce je pojmenovaný blok příkazů, který můžeme znovupoužít

Větvení **if**

- Příkaz **if** umožňuje větvení programu na základě podmínky
- Má dva základní tvary
  - **if** (*podmínka*) `příkaz`$_1$
  - **if** (*podmínka*) `příkaz`$_1$ **else** `příkaz`$_2$
- `podmínka` je logický výraz, jehož hodnota je logického (celočíselného) typu

Tj. false (hodnota 0) nebo true (hodnota různá od 0)

- `příkaz` je příkaz, složený příkaz nebo blok

Příkaz je zakončen středníkem ;

- Ukázka zápisu na příkladu zjištění menší hodnoty z x a y:

Varianta zápisu 1

```
int min = y;
if (x < y) min = x;
```

Varianta zápisu 2

```
int min = y;
if (x < y)
   min = x;
```

Varianta zápisu 3

```
int min = y;
if (x < y) {
   min = x;
}
```

Která varianta splňuje kódovací konvenci a proč?

Příklad větvení **if**

Příklad: Jestliže $x < y$ vyměňte hodnoty těchto proměnných

Nechť proměnné x a y jsou definovány a jsou typu `int`.

Varianta 1
```
if (x < y)
   tmp = x;
   x = y;
   y = tmp;
```

Varianta 2
```
if (x < y)
   int tmp = x;
   x = y;
   y = tmp;
```

Varianta 3
```
int tmp;
if (x < y)
   tmp = x;
   x = y;
   y = tmp;
```

Varianta 4
```
if (x < y) {
   int tmp = x;
   x = y;
   y = tmp;
}
```

- Která varianta je správně a proč?

Příklad větvení **if**-then-**else**

Příklad: do proměnné *min* uložte menší z čísel x a y a do *max* uložte větší z čísel.

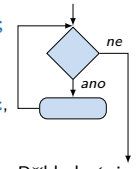
Varianta 1
```
if (x < y)
   min = x;
   max = y;
else
   min = y;
   max = x;
```

Varianta 2
```
if (x < y) {
   min = x;
   max = y;
} else {
   min = y;
   max = x;
}
```

- Která varianta odpovídá našemu zadání?

Cyklus `while ()`

- Příkaz **while** má tvar `while (vyraz) prikaz;`
- Příkaz cyklu **while** probíhá
  1. Vyhodnotí se výraz `vyraz`
  2. Pokud `vyraz != 0`, provede se příkaz `prikaz`, jinak cyklus končí
  3. Opakování vyhodnocení výrazu `vyraz`
- Řídicí cyklus se vyhodnocuje na začátku cyklu, cyklus se nemusí provést ani jednou
- Řídicí výraz `vyraz` se musí aktualizovat v těle cyklu, jinak je cyklus nekonečný



Příklad zápisu
```
int i = 0;
while (i < 5) {
   ...
   i += 1;
}
```


Příklad cyklu `while`

- Základní příkaz cyklu **while** má tvar **while** (*podmínka*) `příkaz`

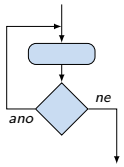
Příklad
```
int x = 10;
int y = 3;
int q = x;

while (q >= y) {
   q = q - y;
}
```

- Jaká je hodnota proměnné q po skončení cyklu?

Cyklus `do...while ()`

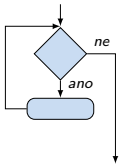
- Příkaz **do...while ()** má tvar
  `do prikaz while (vyraz);`
- Příkaz cyklu **do...while ()** probíhá
  1. Provede se příkaz `prikaz`
  2. Vyhodnotí se výraz `vyraz`
  3. Pokud `vyraz != 0`, cyklus se opakuje provedením příkazu `prikaz`, jinak cyklus končí
- Řídicí cyklus se vyhodnocuje na konci cyklu, tělo cyklu se vždy provede nejméně jednou
- Řídicí výraz `vyraz` se musí aktualizovat v těle cyklu, jinak je cyklus nekonečný



Příklad zápisu
```
int i = -1;
do {
   ...
   i += 1;
} while (i < 5);
```


Cyklus **for**

- Základní příkaz cyklu **for** má tvar **for** (*inicializace*; *podmínka*; *změna*) `příkaz`
- Odpovídá cyklu while ve tvaru:
  inicializace;
  while (*podmínka*) {
  `příkaz;`
  `změna;`
  }



Příklad
```
for (int i = 0; i < 10; ++i) {
   printf("i: %i\n", i);
}
```

- Změnu řídicí proměnné lze zkráceně zapsat operátorem inkrementace nebo dekrementace $++$ a $--$
- Alternativně lze též použít zkrácený zápis přiřazení, např. $+=$

Cyklus **for** – příklady

- Jak se změní výstup když použijeme místo prefixového zápisu $++i$ postfixový zápis $i++$
```
for (int i = 0; i < 10; i++) {
   printf("i: %i\n", i);
}
```
- V cyklu můžeme také řídicí proměnou dekrementovat
```
for (int i = 10; i >= 0; --i) {
   printf("i: %i\n", i);
}
```
Kolik program vypíše řádků?
- A kolik řádků vypíše program:
```
for (int i = 10; i > 0; --i) {
   printf("i: %i\n", i);
}
```
- Řídicí proměnná může být také například typu **double**
```
#include <math.h>

for (double d = 0.5; d < M_PI; d += 0.1) {
   printf("d: %f\n", d);
}
```


Část III

Část 3 – Zadání 2. domácího úkolu (HW02)

Zadání 2. domácího úkolu HW02

Téma: **První cyklus**

Povinné zadání: **2b**; Volitelné zadání: není; Bonusové zadání: není

- **Motivace**: „Automatizovat“ a zobecnit výpočet pro „libovolně“ dlouhý vstup
- **Cíl**: Osvojit si využití cyklů jako základní programového konstrukce pro hromadné zpracování dat.
- **Zadání**: `https://cw.fel.cvut.cz/wiki/courses/b0b36prp/hw/hw02`
  - Zpracování libovolně dlouhé posloupnosti celých čísel
  - Výpis načtených čísel
  - Výpis statistiky vstupních čísel
    - Počet načtených čísel; Počet kladný a záporných čísel a jejich procentuální zastoupení na vstupu
    - Četnosti výskytu sudých a lichých čísel a jejich procentuální zastoupení na vstupu
    - Průměrná, maximální a minimální hodnota načtených čísel
- Termín odevzdání: 19.10.2019, 23:59:59 PDT

PDT – Pacific Daylight Time

Shrnutí přednášky

Diskutovaná témata

- Zápis programu v C
- Literály a konstantní hodnoty
- Program jako algoritmus
- Řídicí struktury

- Příště: Dokončení řídicích struktur, výrazy