



Programování v jazyce C

2 Řídicí konstrukce



- Základní konstrukce řízení běhu programu
- Podmíněné větvení programu
- Konstrukce switch
- Cykly
- Přerušování cyklů a ukončování funkcí





Syntax jazyka C

Jazykové konstrukce

- řídicí konstrukce
- definice datových typů (struktur)
- deklarace proměnných (definice konstant)
- deklarace/definice funkcí
(pokud je funkce dána hlavičkou i tělem, je **definovaná** – má-li jen hlavičku, je **deklarovaná** → *prototyp*)
- operátory
- přiřazovací příkaz(y)
- návěští a příkaz skoku

Knihovní funkce vstupně-výstupních operací **printf** a **scanf** jsou často vykládány v rámci syntaxe, protože tvar tzv. řídicích řetězců ("**%x\n**") je tradičně definován v gramatice jazyka.





Řídicí konstrukce jazyka C

Aparát pro ovlivňování běhu programu

Řídicí konstrukce jsou syntaktické prvky umožňující **řízení běhu programu**, tj. větvení výpočtu na základě (ne)splnění podmínek, opakování úseku kódu (cykly) a skoky v kódu.

- podmíněné větvení programu – **if-else**
- podmíněné větvení programu – **switch**
- cykly – **while, do, for**
- přerušení provádění cyklů – **break, continue**
- přerušení provádění funkcí – **return**





Podmíněné větvení programu

Konstrukce `if` (bez `else` – neúplné větvení)

```
if (i > 5)  
▶▶▶ printf("i je větší než 5!\n");
```

```
if (i > 5) {  
▶▶▶ printf("i je větší než 5!\n");  
▶▶▶ i = 0;  
}
```

- **podmínka musí být v závorkách** (narozdíl od Pascalu, aj.)
- **jeden následující příkaz** (může být ovšem složený, tzv. *compound statement*, *block*) se vykoná při splnění podmínky
- **POZOR:** jazyk C nemá logický typ (boolean) – podmínka je **aritmetický výraz** a je splněna, je-li hodnota výrazu různá od nuly!



Podmíněné větvení programu



- znak ; (středník) má v C význam **prázdného příkazu** (NOP)
- při použití (omylem) za podmínkou způsobí naprosté ignorování podmínky a podmíněný příkaz bude vykonán **vždy**:

```
if (i > 5);  
    printf("i je větší než 5!\n");
```

- tento kód doslova znamená:
 - (1) je-li **i** větší než 5, pak udělej “nic”
 - (2) vytiskni na konzoli text „*i je větší než 5!*”





Podmíněné větvení programu

Úplná podmínka, konstrukce **if-else**

```
if (i > 5) i = 0; else i = -1;
```

středník je **nezbytný!**
ačkoliv působí rušivě

```
if (i > 5) {  
    printf("i je větší než 5!\n");  
    i = 0;  
}  
else {  
    printf("i je menší/rovno 5!\n");  
    i = -1;  
}
```

- ze zápisu by mělo být zjevné, ke kterému **if** patří které **else** (zapisovat pod sebe)



Podmíněné větvení programu

Překlad do strojového kódu

```
...  
if (i > 0) printf("i > 0\n");  
else printf("i <= 0\n");  
...
```

adresa proměnné **i**

```
00401028: cmp dword ptr 0040803C, 0  
0040102F: jle 00401041  
00401031: mov eax, 00408004  
00401036: push eax  
00401037: call near ptr printf_  
0040103C: add esp, 4  
0040103F: jmp 0040104F  
00401041: mov eax, 0040800B  
00401046: push eax  
00401047: call near ptr printf_  
0040104C: add esp, 4  
0040104F: ...
```



Zahnížděné podmínky

Jak překladač řeší nejednoznačnost v syntaxi?

- **else** je vždy překladačem přiřazeno **k nejbližšímu if**

```
if (i > 0)
    if (i > 1000)
        printf("i je velké!\n");
    else
        printf("i je malé!\n");
```

přiřazení **else** lze ovlivnit závorkami bloků

```
if (i > 0) {
    if (i > 1000)
        printf("i je velké!\n");
}
else
    printf("i je 0 nebo záporné!\n");
```



Podmíněné větvení programu konstrukcí `if-else if`

```
if (c == 'S')
    printf("Obsah\n");
else if (c == 'V')
    printf("Objem\n");
else if (c == 'F')
    printf("Síla\n");
else
    printf("Nevím.\n");
```

- mnohdy nepraktické (je-li příliš mnoho možností)
- pokud je třeba dosáhnout maximální čitelnosti a přehlednosti kódu, je dobré se této konstrukci vyhnout a postavit podmínku jinak



Podmíněné vícenásobné větvení programu

Konstrukce `switch`

nepoužití **break** má za následek provedení i dalších větví **switch**

```
switch (c) {  
    case 'S': printf("Obsah\n");  
              break;  
    case 'V': printf("Objem\n");  
              break;  
    case 'F': printf("Síla\n");  
              break;  
    default: printf("Nevím.\n");  
             break;  
}
```

zde už je **break** zbytečný

- nelogická syntax (např. v porovnání s Pascalem), chybějící **break** je častým zdrojem obtížně hledatelných chyb (pomůže debugger)



Konstrukce switch

Překlad do strojového kódu

```

...
switch (c) {
  case 'F':
    printf("Síla...");
    break;
  case 'm':
    printf("Hmot...");
    break;
  case 'a':
    printf("Zryc...");
    break;
  default:
    printf("?\n");
}
...

```

```

0040103C: mov  al, byte ptr 0040A000
00401041: mov  byte ptr -04[ebp], al
00401044: cmp  byte ptr -04[ebp], 6D
00401048: je   00401068
0040104A: cmp  byte ptr -04[ebp], 61
0040104E: je   00401078
00401050: cmp  byte ptr -04[ebp], 46
00401054: je   00401058
00401056: jmp  00401088
00401058: mov  eax, 00409007
0040105D: push eax
0040105E: call near ptr printf_
00401063: add  esp, 4
00401066: jmp  00401096
00401068: mov  eax, 0040900D
0040106D: push eax
0040106E: call near ptr printf_
00401073: add  esp, 4
00401076: jmp  00401096

```

m

a

F



Konstrukce switch

Více podmínek pro stejnou větev

```
switch (c) {  
    case 's':  
    case 'S': printf ("Obsah\n");  
              break;  
  
    case 'v':  
    case 'V': printf ("Objem\n");  
              break;  
  
    case 'f':  
    case 'F': printf ("Síla\n");  
              break;  
  
    default: printf ("Nevím.\n");  
             break;  
  
}
```





Konstrukce `switch`

Významné detaily a upozornění

- lze testovat **jen ordinální typy** (celá čísla, znaky)
- nelze testovat proměnnou

```
switch (i) {  
    case j: printf("i = j\n");  
}
```



- když není splněná podmínka žádné z **case** větví, provede se **default** větev (je-li definována)
- pokud není **default** větev definována, nestane se nic
- **nezapomínejte na `break`**





Cyklus **while**

Cyklus, který nemusí proběhnout ani jednou

- cyklus **while** má podmínku před tělem cyklu, tj. není-li splněna při vstupu do cyklu, cyklus vůbec neproběhne
- cyklus probíhá dokud je výraz v podmínce **nenulový**

```
while (i <= 5)
    printf("i = %i\n", i++);
```

```
while (i <= 5) {
    printf("i = %i\n", i);
    i++;
}
```

**řídící proměnná cyklu –
nezapomenout zvyšovat/snižovat,
jinak vznikne nekonečná smyčka**



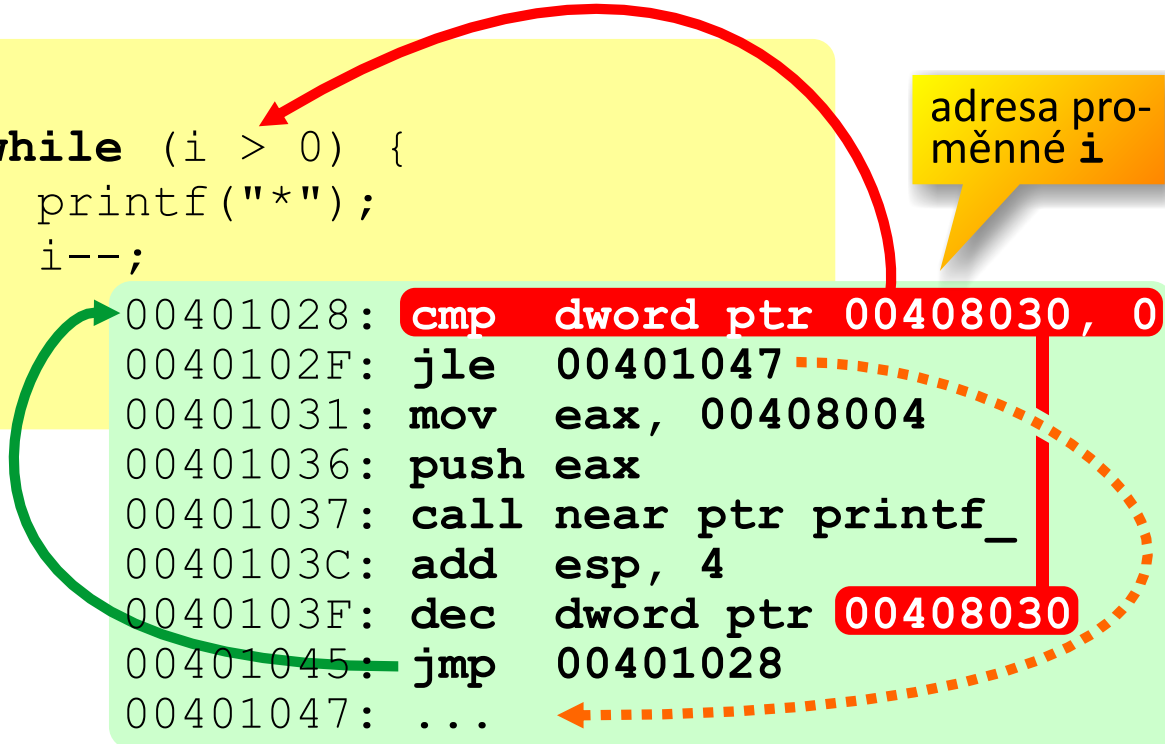
Cyklus while

Překlad do strojového kódu

```
⋮  
while (i > 0) {  
    printf("*");  
    i--;  
}
```

adresa proměnné **i**

```
00401028: cmp dword ptr 00408030, 0  
0040102F: jle 00401047  
00401031: mov eax, 00408004  
00401036: push eax  
00401037: call near ptr printf_  
0040103C: add esp, 4  
0040103F: dec dword ptr 00408030  
00401045: jmp 00401028  
00401047: ...
```





Cyklus while

!!! POZOR !!!

tento středník navíc způsobí,
že vznikne nekonečná smyčka

```
while (i <= 5);  
    printf("i = %i\n", i++);
```



```
while (!keypressed())
```



takto je hned zřejmé, že se jedná o úmysl
a že potřebujeme **prázdné tělo cyklu**

- **Nekonečná smyčka** (celkem rozumné provedení):

```
while (1) { ..... }
```





Cyklus `while`

Logika udržovací podmínky cyklu

- **`while`** znamená **dokud platí**, nikoliv **dokud neplatí** nebo **dokud nenastane**

```
int i = 1;

while (i == 10) {
    printf("i = %i\n", i);
    i++;
}
```

cyklus neproběhne **ani jednou**, protože podmínka není při vstupu do cyklu splněna

- záměrem programátora bylo zřejmě, aby smyčka probíhala dokud **`i`** nenabyde hodnoty 10 (tedy logika pascalského cyklu **`repeat-until`**)



Cyklus do-while

Cyklus, který proběhne alespoň jednou

```
do {  
    printf("i = %i\n", i);  
    i--;  
} while (i > 0);
```

- stejné problémy při použití středníku za **do**
- z důvodů přehlednosti a zamezení vzniku chyby je lepší **vždy používat složený příkaz** (blokové závorky) { ... }
- vyhodnocování udržovací podmínky se provádí na konci cyklu, tj. je zaručen minimálně jeden průchod tělem cyklu
- obdoba pascalského cyklu **repeat-until**, až na to, že podmínka je zde opět v pozitivním smyslu, tj. cyklus probíhá, pokud je výraz v podmínce nenulový



Cyklus do-while

Překlad do strojového kódu

```
...  
do {  
    printf("*");  
    i--;  
} while (i > 0);  
...
```

```
00401028: mov    eax, 00408004  
0040102D: push  eax  
0040102E: call  near ptr printf_  
00401033: add   esp, 4  
00401036: dec   dword ptr 00408030  
0040103C: cmp   dword ptr 00408030, 0  
00401043: jg    00401028  
00401045: ...
```

adresa proměnné i





Cyklus for

Nejsilnější a nejkomplicovanější cyklus

inicializace

podmínka

aktualizace

```
for (i = 1; i <= 10; i++)  
    printf("i = %i\n", i);
```

nepříliš povedená syntax
(středníky – **neměnit za čárky!**,
čárky mají **jiný význam**)

- „nesahat“ na řídicí proměnnou cyklu jinde, než v hlavičce
- cyklus for dokáže nahradit ostatní cykly, tj. jedná se vlastně o jejich zevšeobecněný a zesílený zápis



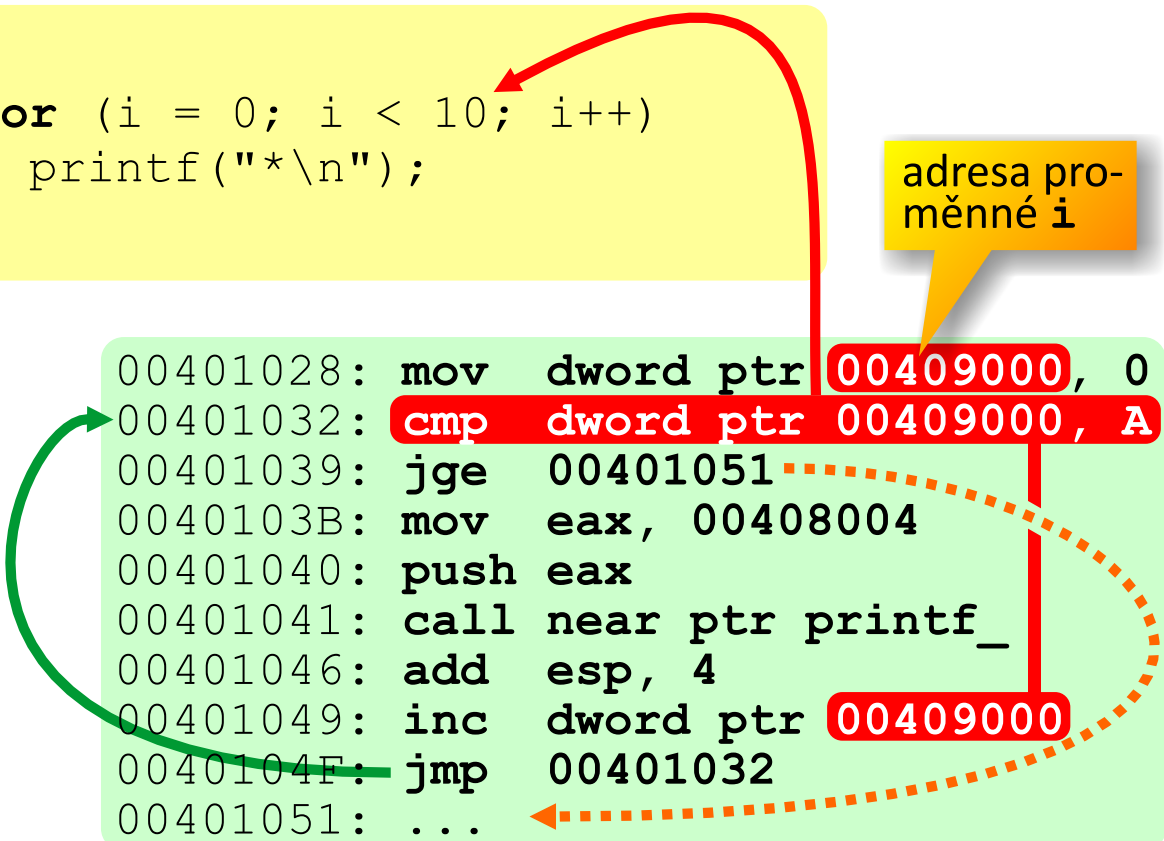
Cyklus for

Překlad do strojového kódu

```
...  
for (i = 0; i < 10; i++)  
    printf("*\n");  
...
```

adresa proměnné i

```
00401028: mov    dword ptr [00409000], 0  
00401032: cmp   dword ptr [00409000], A  
00401039: jge   00401051  
0040103B: mov   eax, 00408004  
00401040: push  eax  
00401041: call  near ptr printf_  
00401046: add   esp, 4  
00401049: inc   dword ptr [00409000]  
0040104F: jmp   00401032  
00401051: ...
```





Cyklus for

Logika udržovací podmínky cyklu

- u `for` je podmínka také ve významu **dokud platí**, nikoliv **dokud neplatí** nebo **dokud nenastane**!

tento zápis znamená „*vykonávej cyklus, dokud se i rovná 10*“, tj. **nikdy**, protože při vstupu do cyklu je `i` nastaveno na 1.

```
for (i = 1; i == 10; i++)  
    printf("i = %i\n", i);
```

- programátor zamýšlel zapsat cyklus tak, aby probíhal **dokud i nenabyde hodnoty 10** – **takhle to ale nejde!**



Cyklus `for` s krokem jiným než 1

`x += 0.1` je zrychlený
zápis `x = x + 0.1`

```
double uh;
```

```
for (uh = 0.0; uh < 3.14; uh += 0.1) {  
    printf("sin(%f) = %f\n", uh, sin(uh));  
}
```

- **drobná nevýhoda:** dopředu nevíme, kolikrát cyklus proběhne (což je obvykle vlastnost, pro kterou si programátor cyklus `for` vybírá) → nejedná se o úplně „čisté“ použití, z hlediska algoritmické správnosti je vhodnější v tomto případě volit cyklus `while` či `do`
- většina příbuzných algolských jazyků (Pascal, Ada) umožňuje cyklus `for` pouze s krokem 1



Cyklus for

Náhradní použití za jiné typy cyklů

```
for (i = 0, j = 0; i < 10; i++, j--) {  
    ...  
}
```

```
for (; i < 10; i++) {  
    ...  
}
```

možno použít v případě
implicitní inicializace, např.
parametrem funkce

```
for (; i < 10;) {  
    ...  
}
```

náhrada **while**



```
for (;;) {  
    ...  
}
```

nekonečná smyčka



- poslední dva zápis rozumný programátor **nepoužívá**



Příkazy `break` a `continue`

Přerušeni/přeskočení běhu těla cyklu

```
while (f < 10.0) {  
    f = vypocet(f);  
    printf("V %i. průchodu f = %f\n",  
        c, f);
```

```
    if (c >= 100) break;  
    c++;  
}
```

**pokud proběhne
více než 100 iterací
výpočtu, ukončí se**

```
for (i = 0; i < 10; i++) {  
    if (i == 5) continue;  
    printf("i = %i\n", i);  
}
```

přeskočí 5





Příkaz **return**

Přerušení provádění funkce

- podobný účinek jako **break**, ale opouští celou funkci a zároveň zajišťuje předání návratové hodnoty

```
unsigned int factorial(unsigned int n) {  
    switch (n) {  
        case 0:  
        case 1: return 1;  
        default: return n * factorial(n - 1);  
    }  
}
```

