Programování v jazyce C

5 Preprocesor jazyka C

- Lexikální konvence preprocesoru
- Příkazy preprocesoru
- Definice maker
- Makra s parametry
- Podmíněný překlad
- Ovládání překladače

Preprocesor jazyka C
Zpracování zdrojového kódu před překladem

- ***Makro procesor*** (lexikálně-syntaktický analyzátor), který zpracovává zdrojový text programu ještě před překladačem.
- Příkazy preprocesoru jsou součástí ZK, vždy na samostatné řádce, začínají znakem **#**.

```
#include <math.h>
```

- Při preprocessingu se řádky s příkazy preprocesoru odstraní; **výsledkem předzpracování musí být platný program v C**.
- Po rozvinutí makra preprocesoru se ve výsledném programu už nesmí vyskytovat žádné příkazy preprocesoru:

Rozvinutím makra INCLUDEMATH vznikne další příkaz preprocesoru, který se ale předá překladači, který ho nezná...

```
#define INCLUDEMATH #include <math.h>
INCLUDEMATH
```

Lexikální konvence preprocesoru
Základní způsoby zápisu příkazů preprocesoru

- Nepřekládá ZK, ale rozděluje ho na ***lexikální atomy***.
- Nevejde-li se příkaz na jeden řádek, je možné pokračovat na dalším po ukončení předchozího řádku zpětným lomítkem:

```
#define err(flag, msg) if (flag) \
   printf(msg)
```

- **POZOR:** Řádek následující po řádku, který končí ****, se nikdy nepovažuje za nový příkaz preprocesoru, i když začíná **#**.

```
#define BACKSLASH \
#define ASTERISK *
```

Tuto definici preprocesor **ignoruje!** (resp. chápe to jako tělo předchozího příkazu #define)

- Řešením je tzv. ***escapování***:

```
#define BACKSLASH \\
```

Příkaz preprocesoru #define
Definice symbolu a jeho nahrazení

- Ve zdrojovém textu **nahradí** výskyt definovaného symbolu (identifikátoru makra) tělem makra (pokud existuje).

```
#define sum(x,y) x+y
…
i = sum(5, a * b);
```

Preprocesor zajistí rozvinutí makra na
`i = 5 + a * b;`

- Preprocesor **neidentifikuje klíčová slova** jazyka C („neumí" C), takže je možné makro pojmenovat stejně, jako klíčové slovo jazyka C, ale to je **nebezpečné** (špatný progr. styl).
- Typické užití – definice konstant, jednoduchých „funkcí":

```
#define ITEM_SIZE 0x100
#define BUFFER_SIZE (256*ITEM_SIZE)
#define QUITMSG "Konec...\n"
```

Příkaz preprocesoru #define
Obvyklé problémy a chyby

```
#define WIDTH = 640
#define HEIGHT 480;
…
size_x = WIDTH;
msize = HEIGHT * 8;
```

Preprocesor zajistí rozvinutí makra na
`size_x = = 640;`

Preprocesor zajistí rozvinutí makra na
`msize = 480; * 8;`

- **Pozor na středníky** za definicí těla symbolu (makra).
- Nevkládat **žádné znaky navíc**, za jménem následuje za mezerou okamžitě definice těla makra (nahr. symbolu).
- **Mezera** slouží jako oddělovač – nespoléhejte na syntaktickou analýzu preprocesoru a nevkládejte mezery do definice těla, příp. identifikátoru makra.

Příkaz preprocesoru #define
Definice makra s parametry („funkce”)

- Obecně `#define <jméno>(<par1>, <par2>, …, \ <parN>) <posloupnost-atomů>`

```
#define add(x,y) ((x)+(y))
…
return add(a + 3, b);
```

Závorky nejsou nutné, ale jsou **vhodné** – zajišťují dodržení priority operátorů.

```
#define getchar() getc(stdin)
…
while ((c = getchar() != EOF) { … }
```

- Makro může mít **nulový počet** parametrů, v tom případě musí i při volání být seznam parametrů prázdný.

Rekurze maker preprocesoru
Chování při výskytu makra v definovaném těle

- Makra, která se objeví ve svém vlastním rozvoji, se v ANSI C **znovu nerozvíjejí** (norma) ⇒ pokud se název makra objeví v jeho vlastním těle, přepíše se při zpracování preprocesorem do výsledného zdrojového kódu:

```
#define sqrt(x) ((x) < 0 ? \
    sqrt(-x) : sqrt(x))
```

- Starší preprocesory tuto „rekurzi" nedokážou detekovat a **zacyklí se**.

```
#define plus(x,y) add(x,y)
#define add(x,y) ((x)+(y))
```

V pořádku – makra se nerozvíjejí v definici jiného makra.

Makra s parametry
Příklad

```
#define step(v,l,h) \
    for ((v) = (l); (v) <= (h); (v)++)

int main() {
    int i;

    step(i, 1, 20)
       printf("%2d %6d\n", i, i * i);

    return 0;
}
```

- Zdánlivé nadužití závorek zajišťuje správnou interpretaci komlexních výrazů v parametrech makra (např. složek struktur, apod.)

Předdefinovaná makra v ANSI C
Použitelná zejména pro ladění programu

__LINE__	číslo právě zpracovávaného řádku programu (desítková celočíselná konstanta)
__FILE__	jméno právě zpracovávaného souboru (řetězcová konstanta)
__DATE__	aktuální kalendářní datum (řetězcová konstanta tvaru *mm dd yyyy*)
__TIME__	čas překladu (řetězcová konstanta tvaru *hh:mm:ss*)
__STDC__	má hodnotu 1 ⇔ překladač ANSI C

- Typické použití při ladění spolu s makrem **`assert`** (viz dále) a pro testování, zda je prostředí vhodné pro překlad:

```
if (n != m) printf("Chyba na radku %d "
  "v souboru %s\n", __LINE__, __FILE__);
```

Příkaz preprocesoru #undef
Zrušení (definovaného) symbolu/makra

- Definice makra se ruší příkazem **#undef** <jméno-makra>
- Lze zrušit i makro (symbol), které **nebylo nadefinováno**.

```
#define madd(a,b) ((a)+(b))

int main() {
    int i = 3, j = 5, k, l;

    k = madd(i, j);
#undef madd
    l = madd(i, j);

    return 0;
}
```

Chyba!
(ovšem hlásit ji bude až linker)

Příkaz preprocesoru `#include`
Vkládání (hlavičkových) souborů

```
#include <stdio.h>
#include "mylib.h"
#include "ext/mylib.h"
```

Win: lomítko může být **** i **/**
UNIX: jen **/**

- Je-li jméno vkládaného souboru uzavřeno v ‘<’…’>’, hledá se v instalaci překladače (obvykle adresář …/include).
- Je-li v ‘"’…’"’, pak se prohledává aktuální adresář (kde je ZK).
- Vkládaný soubor může sám obsahovat příkaz **`#include`** – min. garantovaná povolená hloubka vnoření (ANSI C) je **6**.
- Bezpečná strategie při vkládání hlavičkových souborů z pod-adresářů je užívat **normální lomítko** (jako v UNIXu).

Příkaz preprocesoru `#include`
Technika zabránění vícenásobnému vložení

- Je-li hlavičkový soubor vkládán z více překladových jednotek projektu, mohlo by dojít k **vícenásobnému vložení** ⇒ redefinice typů, konstant, apod. – překladač bude hlásit chyby.
- Uvedený postup spolehlivě brání vícenásobnému vložení, je **nutné** jej používat v každém hlavičkovém souboru:

```
#ifndef MYLIB_H
#define MYLIB_H
...
<obsah hlavičkového souboru>
...
#endif
```

Symbol volíme tak, aby reflektoval název hlav. souboru.

- Moderní překladače (resp. jejich preprocesory) nabízí ke stejnému účelu příkaz `#pragma once` – **nespolehlivé!**

Podmíněný překlad
za pomoci příkazů `#ifdef–#else–#endif`

```
int main() {
    printf("Environment: ");
#ifdef WIN32
    printf("Win32\n");
#elif UNIX
    printf("UNIX\n");
#elif OS2
    printf("OS/2\n");
#else
    printf("Unknown...\n");
#endif
    return 0;
}
```

Použití `#elif` v kombinaci s `#ifdef` funguje, ale není to úplně čisté.

```
C:\>cl -DWIN32 preproc2.c
```

- Symbol lze „nadefinovat zvenčí" – prostřednictvím přepínače překladače (např. v dávkovém souboru či makefile).

Podmíněný překlad
Příkazy preprocesoru `#if` a `#endif`

```
#if 1<<16
...
#endif
```

konstantní výraz preprocesoru (KVP) (vzhledem k tomu, že preprocesor neprovádí komplexní syntaktickou analýzu, **téměř nic mu nevyhovuje**)

- Tento kód **nesprávně** testuje, zda je **`int`** větší než 16 bitů, ve skutečnosti ale preprocesor pracuje vždy pouze s typem **`long`** nebo **`unsigned long`** podle znaménka operandu (jednoduše řečeno používá vlastní celočíselný typ).
- KVP vyhovují jen celočíselné konstanty, znakové konstanty, jednoduché výrazy s nimi a speciální operátor preprocesoru **`defined`** (pouze ANSI C).

Podmíněný překlad
Příkazy **#elif**, **#else** a operátor **defined**

```
...
#if defined(WIN32)
    printf("Win32\n");
#elif defined(UNIX)
    printf("UNIX\n");
#else
    printf("Unknown\n");
#endif
...
```

- Silnější nástroj, než prostý **#ifdef** nebo **#ifndef**.
- Umožňuje budovat složitější výrazy, např.:

```
#if defined(WIN32) && !defined(UNIX)
...
#endif
```

Příkaz preprocesoru `#pragma`
Ovládání překladače

- Zaveden v ANSI C, slouží k přidávání nových funkcí preprocesoru nebo k **předávání informací překladači** (závisí na konkrétní implementaci překladače)
- Příkaz **`#pragma`** je **silně závislý** na platformě a implementaci překladače (velmi se liší) ⇒ **používat podmíněně!**

```
#if defined(VAX) && defined(__STDC__)
#pragma builtin(abs), inline(myfunc)
#endif
```

- Mezi výrobci překladačů neexistuje dohoda o standardních tvarech, konkrétní lze nalézt v dokumentaci k překladači:

```
align, pack          extend
rom                  small, medium, large
optimize             debug
check stack          inline
```

Příkaz preprocesoru `#error`
Vynucení chybového stavu

- Zaveden v ANSI C, umožňuje **vyvolat chybu po zachycení nesrovnalosti** např. při překladu řízeném dávkovým souborem či skripty (makefile, CMake, apod.):

```
#if defined(WIN32) && defined(UNIX)
#error "Chyba v Makefile!"
#endif
```

```
#include "sizes.h"
…
#if (SIZE % 256) != 0
#error "SIZE musí být násobek 256"
#endif
```

Operátor preprocesoru
Převod atomu na řetězcovou reprezentaci

- **Pouze ANSI**, starší preprocesory operátor **#** neznají

```
#define test(a,b) \
  printf(#a "<" #b ": %d\n", (a) < (b))

int main() {
   test(-5, 5);
   return 0;
}
```

```
printf("-5<5: %d\n", (-5)<(5));
```

- Každá posloupnost prázdných znaků v rozvoji parametru se nahradí právě jednou mezerou.
- Každému znaku **** a **"** se **předřadí zpětné lomítko**, aby se zachoval jejich význam v řetězci.

Operátor preprocesoru
Spojování atomů (*Token Pasting*)

- Při rozvoji makra **spojuje atomy** (své dva operandy).
- **Pouze ANSI**, starší preprocesory operátor **##** neznají

```
#define var(i) var ## i
…
var(1) = var(2);
```

var1 = var2;

- Užitečný nástroj, ale **používat s rozumem!**
- Častá nutnost použití tohoto operátoru naznačuje nevhodné řešení problému.
- Tímto způsobem nelze zřetězit znaky ‘**/**’ a ‘*****’ a tak vytvořit komentář – to proto, že **preprocesor převádí komentáře na bílé znaky ještě předtím, než začne zpracovávat makra**.

Operátory preprocesoru # a
Příklad

```
struct command {
   char *name;
   void (*function) (void);
};

struct command commands[] = {
   {"quit", quit_command},
   {"help", help_command},
   …
};
```

čistší řešení

```
#define COMMAND(NAME) {#NAME, NAME ## _command}

struct command commands[] = {
   COMMAND(quit),
   COMMAND(help),
   …
};
```