Práce s preprocesorem

V první kapitole jsme si řekli, jakým způsobem je kód napsaný v jazyce C zpracováván do podoby spustitelného souboru. Zdrojový kód je nejprve předzpracován pomocí preprocesoru, následně je takto upravený kód předán překladači a poté je zpracován linkerem.
V této kapitole se budeme věnovat preprocesoru, který jsme doposud využívali nevědomky (například jsme používali direktivu `#include`). Preprocesor má mnoho možností, které dávají jazyku C další sílu.
Jak již bylo řečeno, preprocesor zpracovává zdrojový kód před překladačem, ale nekontroluje jeho syntaktickou správnost. Pouze provádí záměnu textů, odstraňuje z kódu komentáře a připravuje podmíněný překlad.
Řádky zpracovávané preprocesorem začínají # (přičemž před ani za # by neměla být mezera).

Makra

Makro je definice pravidla, podle kterého vstupní posloupnost transformujeme na výstupní posloupnost. Tuto transformaci označujeme jako *substituci* nebo *expanzi* makra.

Makra bez parametru (konstanty)

Makra bez parametrů nazýváme *symbolické konstanty*. Symbolické konstanty zbavují kód tzv. magických nepojmenovaných čísel.[117]
Například v následujícím vzorečku na výpočet obvodu kružnice je jím číslo 3.14.

```
o = 2 * 3.14 * r;
```

Makra se obvykle definují na začátku programu. Preprocesor při úpravě kódu nahradí tyto konstanty skutečnou hodnotou. Konvencí je, jak již bylo řečeno, že se název píše velkými písmeny. Syntaxe je následující.

[117] Takto definované konstanty oproti proměnným definovaným modifikátorem `const` zabírají méně místa, ale mohou způsobovat problémy při ladění (překladač nezná jejich názvy, debugger je nevidí).

```
#define JMENO hodnota
```

Na konec řádku se nepíše znak ;. Následuje několik příkladů.

```
#define PI 3.14
#define AND &&
#define ERROR printf("Chyba programu");
```

Při zpracování je každý výskyt jména konstanty v následujícím textu zdrojového kódu nahrazen hodnotou této konstanty.

```
#define PI 3.14

int main(){
    float r = 3;
    float o = 2 * PI * r;  /* o = 2*3.14*r; */
    return 0;
}
```

Výjimkou jsou výskyty uzavřené v uvozovkách (části textového řetězce), viz následující příklad.

Úkol 71
Jakým způsobem bychom vypsali jméno?

```
#define JMENO "Marketa"

int main(){
    /* Vypise Moje jmeno je JMENO */
    printf("Moje jmeno je JMENO");
    return 0;
}
```

Rozsah platnosti konstanty je od definice až do konce souboru. Pokud je nutné změnit hodnotu konstanty, je nutné jí zrušit pomocí `#undef JMENO` a znovu definovat. Viz následující příklad.

```
#define JMENO "Marketa"

#undef JMENO

#define JMENO "TRNECKOVA"
```

Pokud je potřeba definici makra napsat na více řádků, přidáme na konec každého řádku znak \

```
#define DLOUHA_KONSTANTA 1.23456798\
910111213
```

Preprocesor tento znak vynechá a pokračuje ve zpracování hodnoty na následujícím řádku. Je-li potřeba použít znak \, je nutné ho zdvojit.
V ANSI C jsou předdefinovaná následující makra. Využívají se zejména při ladění.

- `__LINE__` – číslo právě zpracovávaného řádku programu (desítkové číslo),
- `__FILE__` – jméno právě zpracovávaného programu (řetězec),
- `__DATE__` – aktuální datum (řetězec ve tvaru mmm dd yyyy),
- `__TIME__` – čas překladu (řetězec ve tvaru hh:mm:ss),
- `__STDC__` – má hodnotu 1 pokud se jedná o ANSI C .

Makra s parametry (inline funkce)

Pokud v programu používáme funkci, která je tvořena velmi malým počtem příkazů, bývá výpočet značně neefektivní, protože režie spojená s voláním funkce (předání parametrů funkci, skok do funkce, úschova návratové adresy, ...) je výpočetně náročnější, než provedení příkazů v těle funkce. Místo klasické funkce lze použít makro s parametry, které nevytváří žádnou režii za běhu programu. Nevýhodou je vznik delšího (většího) programu a nemožnost použít rekurzi (viz dále).
Dle konvence se názvy takovýchto maker píšou malým písmem, jako klasické funkce. Syntaxe je následující.

```
#define jmeno(arg_1, arg_2, ..., arg_n) telo_makra
```

Mezi jménem a závorkou nesmí být mezera (bylo by to bráno jako konstanta). `arg_1, ..., arg_n` se chovají podobně jako argumenty funkcí.[118]
Vzhledem k tomu, že při expanzi maker dochází pouze k nahrazení jednoho textu jiným, je doporučováno uzavřít celé tělo makra do závorek a stejně tak každý výskyt argumentů v makru, viz následující příklad.

```
#define na2(x) ((x) * (x))
#define na2a(x) x * x

na2(f + g);  /* ((f+g)*(f+g)) */
na2a(f + g); /* f+g*f+g */
```

Měli bychom se vyvarovat použití argumentů s vedlejším efektem.[119]

[118] Samozřejmě je možné vytvořit makro s nulovým počtem argumentů.

[119] Například argumenty obsahující operátory ++ nebo --.

Úkol 72
Podívejte na následující kód. Než ho zkusíte přeložit, zamyslete se, co bude výsledkem.

```
#define na2(x) ((x)*(x))

int i = 2;
na2(i++);
```

Makra, která se objeví ve svém vlastním těle se v ANSI C znovu nerozvíjejí. Pokud se název makra objeví v jeho vlastním těle, přepíše se tento název do výsledného zdrojového kódu.[120]

```
#define m(x) ((x) < 0 ? m(-x) : m(x))
```

Avšak je v pořádku použít v těle jednoho makra jiné makro. Viz následující příklad.

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

Úkol 73
Pro porovnání různého chování maker a funkcí naprogramujte funkci `fna2`, která bude vracet druhou mocninu. Co bude jejím výsledkem pro argument i++?

[120] Starší preprocesory se mohou zacyklit.

Operátory # a

Tyto operátory se používají v makrech obsahujících parametry. Operátor # se píše před název parametru. V makru je tento výraz nahrazen řetězcem, který je stejný, jako argument makra. Operátor ## je používá ke spojení dvou argumentů v jeden řetězec. Příklad následuje.

```
#define plus(X,Y) printf("%s + %s = %d", #X, #Y, (X) + (Y))
/* plus(3,2);
   printf("%s + %s = %d", "3", "2", (3) + (2));*/

#define var(X) promenna ## X
/* var(10)
   promenna10 */
```

Podmíněný překlad

Podmíněný překlad používáme pro dočasné vynechání části kódu při kompilaci. Typicky se používá pro vynechání ladících částí programu, překlad platformově závislých částí zdrojového kódu či dočasné odstranění (zakomentování) větší části zdrojového kódu. Syntaxe vypadá následovně.

```
#if podminka
    kod
#endif
```

Preprocesor vyhodnocuje podmínku, která následuje za `#if`. Pokud je tato podmínka vyhodnocena jako `false`, pak vše mezi `#if` a `#endif` je ze zdrojového kódu vypuštěno.

Za podmínkou `#if` může být další podmínka `#elif` (může jich být více). Ta se vyhodnotí v případě, že předchozí podmínka nebyla splněna. Pokud nejsou splněny žádné podmínky, vyhodnotí se část kódu za `#else` (není povinný). Podmíněný překlad se ukončuje `#endif`. Syntaxe je v takovém případě následující.

```
#if podminka1
    cast_1
#elif podminka2
    cast_2

    ...

#else
    cast_n
#endif
```

V podmínce musí být výrazy, které může vyhodnotit preprocesor (například v nich nelze používat hodnoty proměnných).
Podmíněný překlad můžeme řídit konstantním výrazem. Syntaxe je následující.

```
#if konstantni_vyraz
    cast_1
#else
    cast_2
#endif

/* NEBO */

#if konstantni_vyraz
    cast_1
#elif konstantni_vyraz2
    cast_2
#else
    cast_3
#endif
```

Pokud je hodnota `konstantni_vyraz` nenulová, překládá se `cast_1`, jinak se překládá `cast_2`.[121] Konkrétní příklad následuje.

```
#define ENG 1

#if ENG
    #define ERROR "error"
#else
    #define ERROR "chyba"
#endif
```

[121] Ve druhém případě se zjišťuje, zda je `konstantni_vyraz2` nenulový a `cast_2` se překládá jen v tom případě, jinak se překládá `cast_3`.

Úkol 74
Upravte kód, tak aby se rozeznávaly čtyři různé jazyky pro chybovou hlášku.

K dispozici jsou i direktivy `#ifdef` a `#ifndef`, které slouží k otestování, zda byla nějaká symbolická konstanta definovaná. `#ifdef JMENO` znamená, že kód následující za touto podmínkou se vykoná jen pokud je makro `JMENO` definováno (obdobně `#ifndef JMENO`, pokud `JMENO` není definováno).

```
#define ENG 1

#ifdef ENG
    #define ERROR "error"
#else
    #define ERROR "chyba"
#endif
```

`#ifdef` a `#ifndef` testují existenci jednoho makra. Pro složitější podmínky je potřeba použít operátor `defined` a logické spojky. Viz následující modifikace příkladu.

```
#define ENG 1

#if defined(ENG) && ENG
    #define ERROR "error"
#else
    #define ERROR "chyba"
#endif
```

Navíc na rozdíl od `#ifdef` a `#ifndef`, je možné použít složitější větvení pomocí `#elif`.

```
#define ENG 1

#if defined(ENG) && ENG
    #define ERROR "error"
#elif !defined(ENG)
    #define ENG 0
    #define ERROR "chyba"
#else
    #define ERROR "chyba"
#endif
```

Vkládání souborů

Bez předchozího vysvětlení jsme používali příkaz `#include <nazev>`, abychom mohli používat funkce z knihovny `nazev`. Preprocesor tento příkaz nahradí obsahem souboru `nazev` (tento soubor je do kódu inkludován).

`#include` lze použít dvěma způsoby následovně.
`#include <nazev>` (jak jsme používali doposud) a nebo `#include ''nazev''`. Tyto dva příkazy se liší tím, kde se soubor `nazev` má hledat.
`#include ''nazev''` hledá soubor ve stejném adresáři, ve kterém je uložen „volající" program. Používá se zejména pro vkládání souborů, které jsme vytvářeli sami.
`#include <nazev>` hledá v systémovém adresáři. Používá se pro vkládání standardních knihoven.

Další direktivy preprocesoru

Kromě výše zmíněných direktiv existují ještě další, které se používají k řízení překladu. Jsou to direktivy `#line`, `#error`, `#warning` a `#pragma`.
Direktiva `#pragma` se používá k uvození implementačně závislých direktiv. Pokud překladač narazí na tuto direktivu a nerozumí jí, pak ji ignoruje. Každý překladač rozumí různým direktivám. Vývojová prostředí je například používají k uložení různých informací, které nemají se zdrojovým kódem nic společného.
Direktiva `#line` se používá k nastavení hodnot maker `__LINE__` a `__FILE__`.
`#error` a `#warning` vypisují při překladu varování (`#error` překlad ukončí). Nejčastěji se používají s podmíněným překladem. Za direktivou následuje text chybového hlášení, který se má vypsat.

Cvičení

Úkol 75
Napište makro `na_3(x)`, které bude počítat třetí mocninu. Proměnné `i` a `j` předem definujte. Vyzkoušejte ho na následujících výrazech.

```
na_3(3)
na_3(i)
na_3(2 + 3)
na_3(i * j + 2)
```

Úkol 76
Definujte makro `je_velike(c)`, které vrátí `0` není-li znak velké písmeno a `1`, pokud je.

Úkol 77
Definujete makro `cti_int(i)`, které čte z klávesnice celé číslo. Makro musí jít použít i ve výrazu
`if((j=cti_int(k)) == 0)`
Nápověda: možná budete potřebovat operátor čárky.

Úkol 78
Následující kód upravte tak, aby se ve funkci `funkce()` hodnoty proměnných vypisovaly jen v případě, že je makro `VERBOSE` nastaveno na 1.

```
#include <stdio.h>

int funkce(int a, int b){
    int i, j = 0;

    for(i = 0; i < a; i = i + b){
        printf("i = %d \n", i);
        if(i > j)
        {
            j = j + i;
        }
        printf("j = %d \n", j);
    }
    return i + j;
}

int main(){
    printf("Vysledek: %d ", funkce(50,5));
    return 0;
}
```

Úkol 79
Přeložte předchozí programy s použitím přepínače `-E` (jak už víme, při překladu zdrojového kódu se provede jen předzpracování pomocí preprocesoru). Podívejte se na výsledek.

```
gcc program.c -E -o program.txt
```