Příkazy preprocesoru

- Před překladem kódu překladačem mu „předpřipraví“ kód preprocesor
- Preprocesor vypouští nadbytečné (prázdné) mezery a řádky
- Preprocesor je možné ovládat pomocí příkazů - řádky začínající znakem #
- Nejčastěji používané příkazy preprocesoru – `include, define, ifdef, ifndef, if, defined(), endif`
- Příkaz `include` (`#include <jmeno_souboru>` nebo `#include "jmeno_souboru"` – slouží k vložení souboru do daného místa (preprocesor „odbočí“ do jiného souboru a posílá ho do překladače – překladači se zdá soubor jako jeden celek
- Příkaz `define` slouží k definici symbolů podmíněného překladu, definici konstant, k náhradě složitých kódů (makra s funkčním voláním) – všechny definice platí od místa zveřejnění do konce souboru
- Příkazy `define` je možné „zrušit“ pomocí `undefine`

Definice konstant

- jsou dva typy definic jednoduchých symbolů: „pro překladač“ a definice konstant
- příkaz s jedním parametrem `#define EXISTUJE12` slouží k tomu, že překladač si zařadí daný řetězec (proměnnou `EXISTUJE12`) do tabulky a můžeme se dále ptát na její existenci (především ve spojení s příkazy preprocesoru `#if`)
- příkaz `define` se dvěma parametry: `#define MOJE_PI 3.1415` slouží k definici textových řetězců. Preprocesor si „zařadí“ první řetězec (od mezery za `define` po první další mezeru – zde `MOJE_PI`) jako referenční a přiřadí k němu řetězec druhý (od druhé mezery do konce řádku). Pokud preprocesor v dalším textu najde první řetězec, nahradí ho řetězcem druhým (který potom „vidí“ překladač).
  Tento mechanizmus slouží k lepší čitelnosti kódu (člověk vidí `MOJE_PI`, překladač pracuje s hodnotou `3.1415` – každý má to čemu rozumí lépe).
- jelikož se proměnná makra fyzicky nevytváří (v programu), bývají definice umístěny do hlavičkových souborů. Hodnota PI definovaná pomocí `MOJE_PI` má potom v celém programu stejnou přesnost, a je-li potřeba přesnost změnit, učiníme tak na jednom místě a změní se v celém programu.

- názvy za `define` je zvykem (a nepsanou dohodou) psát celé velkými písmeny
- definice symbolů (náhradu za `#define XXX`) je možné zadat i jako parametry prostředí (překladače) – platí potom pro všechny soubory

Zjištění „přítomnosti“ konstant

- pomocí dalších direktiv preprocesoru se můžeme ptát na to, zda jsou symboly definované. Jedná se především o příkazy s if, kdy pokud je podmínka splněna, potom preprocesor „pošle“ následující blok (po end) překladači, jinak ho vynechá („zahodí“) a překladač ho nezpracovává
- `#ifdef XXX` – vrací true, je-li `XXX` definováno (preprocesor ho má v tabulce)
- `#ifndef XXX`– vrací true, není-li `XXX` definováno
- `#else, #elif` – větvení a větvení s další podmínkou
- `#endif` – ukončení bloku (začátek dán `#if`....)
- `#if výraz` (pomocí defined se můžeme ptát na hodnotu proměnných a skládat je pomocí logických operátorů (jazyka C) - && || !).
- Není-li symbol nadefinován a přesto se ptáme na jeho hodnotu pomocí: `#if SYMBOL`, je výsledkem hodnota nula (false)

Realizujte tzv. „podmíněný překlad“ části textu. Pokud je zapnutý přepínač LADENI vytiskněte text „ladění“. Pokud je zapnutý LADENI a HODNOTA vytiskněte text „ladění“ a hodnotou proměnné xx dekadicky a hexa. Jinak proveďte tisk pouze dekadicky

```
#define LADENI
#define _HODNOTA
    //definice proměnných. Proměnná HODNOTA je „vypnuta“ pomocí úvodního
    //podtržítka (lépe možná použít „přepínání“ pomocí #undefine HODNOTA)

#ifdef LADENI
    //test na přítomnost proměnné, je-li přítomna, pak se tato část překládá, jinak ne
    printf(" Ladeni ");
#endif
    //konec podmíněného překladu
```

// pokud potřebujeme složitější podmínku, převedeme na logické hodnoty

```
#if defined(LADENI) && defined (HODNOTA)
   printf(" Ladeni %d %x", xx, xx);
#else
   printf(" Vysledek %d", xx);
   //tisk při nesplnění podmínkách („normální“ stav)
#endif
```

Pomocí `define` nadefinujte konstantu 2*PI

```
#define 2*PI 6.28
```
- nelze, jelikož první část („2*PI“) není regulérní řetězec

```
#define 2PI 6.28
```
- nelze, protože název nemůže začínat číslem

```
#define MOJE_PI 3.1415
#define DVE_PI MOJE_PI+MOJE_PI
```
- v pořádku (?). Pokud se nalezne text `DVE_PI`, nahradí se zbytkem řádku.

Co se stane v následujícím případě?

```
obvod = DVE_PI * polomer;
```

dojde k postupnému nahrazení řetězců
zdroj:

```
obvod = MOJE_PI+MOJE_PI * polomer;
```

výsledek:

```
obvod = 3.1415+3.1415 * polomer;
```

- z obou výrazů je patrné, **že nedošlo k tomu, co jsme požadovali**, poloměr násobí pouze druhé „Pi“.
- řešením je využití vysoké priority operátoru (), takže to co je mezi nimi se vyčíslí nejdříve

Opravte předchozí příklad

změna definice

```
#define DVE_PI (MOJE_PI+MOJE_PI)
```

- při použití již dělá očekávané

zdroj:

```
obvod = DVE_PI * polomer;
```

první nahrazení:

```
obvod = (MOJE_PI+MOJE_PI) * polomer;
```

výsledek po druhém nahrazení:

```
obvod = (3.1415+3.1415) * polomer;
```

Pozn.

- předchozí mechanizmus definice konstant by měl být postupně nahrazen definicí proměnné s klíčovým slovem **`const`**.
- „klasická“ definice s **`const`** umožní zadat konstantě přesný datový typ.
- proměnná se nemusí fyzicky vytvořit???

```
int const MAX_POCET = 50;
double const PI = 3.1415;
```

- inicializace je jediné místo, kdy je možné přiřadit hodnotu, dále již manipulaci hlídá (a nové přiřazení "zakáže") překladač.

Použití:

```
if(i < MAX_POCET) {...}

obvod = 2 * PI * polomer;
```

Vkládání (hlavičkových) souborů - include

- umožňuje rozdělení souborů do modulů
- jsou v ní uvedeny deklarace proměnných, funkcí, maker, nových datových typů (**`enum, struct, union`** ...)
- jelikož se počítá s vícenásobných vložením (do různých .cpp souborů), nesmí žádná část hlavičkového souboru tvořit (explicitně) kód – linker by našel více stejných proměnných a zahlásil by chybu.

Proveďte ošetření hlavičky proti vícenásobnému načtení. Stane-li se, že je hlavičkový soubor načten vícekrát (například v různých vkládaných souborech) je vhodné, aby se nezpracovával vícekrát než jednou (ztráta času, ...). Dalším problémem je, jsou –li dva soubory volány „do kříže“ – pak dojde k zacyklení. Zabraňte těmto chybám. uvnitř hlavičkového souboru vytvoříme následující konstrukci

```
// dotaz zda už jsme tu byli
#ifndef JEDINECNY_IDENTIFIKATOR_H_SOUBORU
// ještě ne (jinak bychom tento blok přeskočili)
#define JEDINECNY_IDENTIFIKATOR_H_SOUBORU
// ihned jako první si poznačíme, že už jsme tu byli

// tady bude vše potřebné – include jiných souborů, deklarace proměnných, funkcí, typů ...

#endif
// ukončení bloku
```

Pozn.: tento mechanizmus by byl vhodnější použít již při „volání“ souboru pomocí include – ušetřilo by se otevírání souboru. Ale výše uvedenou konstrukci by musel psát uživatel při každém volání, což by bylo pracné a při více načítaných souborech i nepřehledné. K tomuto v některých překladačích používaly direktivu `#pragma once` – ta ovšem není součástí normy a proto ji nepoužívejte (není přenositelná mezi různými překladači)

Makra s funkčním voláním – define s parametry

- podobné jako definice konstant. V prvním řetězci jsou však proměnné, kterými jsou při vkládání do textu nahrazeny proměnné z druhého řetězce
- většinou se snažíme, aby makro fungovalo jako funkce
- dosti často používané v systémových knihovnách (a souborech .h)
- vhodné pro zpřehlednění textu, pro zápisem dlouhé, ale kódem krátké bloky programu

// při definici

```
#define VYPOCET(a,b) a*a+b
```

je při použití nahrazení:

```
cc = VYPOCET (dd+ee,ff) * 6;
```

// VYPOCET=> a*a+b a => dd+ee b=>ff

```
cc = a*a + b * 6 = dd+ee*dd+ee+ff * 6
```

opět vidíme, že to není to, co bychom chtěli. Opravte definici a rozepište výsledné vyčíslení.

```
#define VYPOCET(a,b) ((a)*(a)+(b))
```

závorky kolem „parametru“ zajistí vyčíslení před použitím „parametru“
závorky kolem celého výrazu zajistí vyčíslení výrazu před dalším použitím

je při použití nahrazení:

```
cc = VYPOCET (dd+ee,ff) * 6;
```

// VYPOCET=> a*a+b a => dd+ee b=>ff

```
cc = ((a)*(a) + (b) )* 6 = ((dd+ee)*(dd+ee)+(ff)) * 6
```

nyní se makro chová jako funkce.

Jsou zde však stále rozdíly mezi použitím makra a funkce. Jaké?

Funkce provádí konverze na typy dané v definici.
Předávaný parametr se vyčíslí a změní typ na typ parametru, který je uveden v hlavičce funkce. Obdobně se konvertuje návratová hodnota.
Makro počítá v typech použitých proměnných (v největší přesnosti z použitých). Při „volání“ s proměnnými typu `int` je výpočet prováděn v `int`, při volání s double je výpočet prováděn nad typem `double`.

Funkce je v celém programu pouze jednou. Při jejím použití je volána jako podprogram. Nevýhodou je časová režie spojená s předáváním (a úklidem) proměnných a skokem do podprogramu. Makro se vkopíruje pokaždé do textu zdroje a v každém místě použití se znovu přeloží. Takový program je rychlejší. Pokud je ale makro delší než kód režie (jednotky až desítky instrukcí), dochází k prodlužování kódu programu.

Zápis pomocí makra by měl být nahrazen pomocí tzv. inline funkcí. Jejich překlad je proveden jako u makra přímo v kódu bez volání, ale zároveň se provádí přetypování na základě typů z hlavičky funkce.

```
inline int Vypocet(double a, long  b) {return a*a+b;}
```

při použití :

```
double a;
a = Vypocet('c',3.245);
```

se v daném místě přeloží:

```
a = (int) ((double)('c') * (double)('c')+(long)(3.245) );
```