Preprocesor a koncepce (větších) programů

Úvod do programování 2

Tomáš Kühr

Práce s preprocesorem

Preprocesor

- Zpracovává zdrojový kód ještě před překladačem
- Provádí pouze záměny textů
  (např. identifikátor konstanty za její hodnotu)
- Vypouští ze zdrojového kódu komentáře
- Umožňuje podmíněný překlad programu (vypuštěním jeho částí)
- Vůbec nekontroluje syntaktickou správnost programu
- Řádky pro zpracování preprocesorem
  - Začínají znakem „#”
  - Za nímž následuje tzv. direktiva preprocesoru
  - Mezi „#” a direktivou nesmí být mezera

Konstanty

- Definice symbolické konstanty obecně:
  `#define jméno hodnota`
- Příklady definic:
```
#define PI 3.14
#define AND &&
#define POSUN ('a'-'A')
#define ERROR printf("Chyba v programu!");
```
- Výskyty jména konstanty se v dalším textu nahrazují její hodnotou
- Nenahrazují se výskyty jména uzavřená v uvozovkách
```
printf("Logicka spojka AND...");
```
- Předefinování symbolické konstanty není možné přímo
- Konstantu lze zrušit a pak ji případně znova definovat (s jinou hodnotou)
```
#undef PI
#define PI 3.1416
```

Konstanty

- Jména se obvykle píší velkými písmeny (příp. s podtržítkem)
- Hodnotu konstanty je možné rozdělit na více řádků (znak „\“ se při kopírování vynechá)

```
#define NASTALA_CHYBA {\
    printf("Nastala chyba!");\
    exit(1);\
}
```

- Pozor – hodnoty se opravdu jen kopírují na místo jmen

```
#define POSUN 'a'-'A'
…
znak = znak - POSUN;
```

- Bezpečnější tedy bude výrazy vždy uzávorkovat

```
#define POSUN ('a'-'A')
…
znak = znak - POSUN;
```

Makra

- Krátké opakující se části kódu lze realizovat
  - Pomocí krátkých funkcí → neefektivita
  - Pomocí maker → efektivní výpočet, ale větší program
- Definice makra obecně:
  `#define jméno(arg1, ..., argN) tělo_makra`
- Příklad definice:
  `#define na2(x) ((x)*(x))`
- Mezi jménem makra a závorkou nesmí být mezera
- Tělo makra může být rozděleno na více řádků pomocí „\“
- Jména maker se píší malými písmeny, příp. s podtržítkem
- U maker nelze použít rekurzi

Makra

- Uzavírejte tělo makra a každý argument v těle do závorek
  `#define je_cislice(c) (((c)>='0')&&((c)<='9'))`
- Typická chyba při vynechá závorek kolem argumentů (modré)
  `#define na2(x) (x*x)`

  …

  `v=na2(f-g);` se přepíše na `v=(f-g*f-g);`
- Typická chyba při vynechání vnějších závorek (červené)
  `#define min(a,b) (a)<(b)?(a):(b)`

  …

  `if(min(3,4)<0)…` se přepíše na `if((3)<(4)?(3):(4)<0)…`
- Pozor na argumenty s vedlejším efektem
  `min(x++, y--)…` se přepíše na `(x++)<(y--)?(x++):(y--)…`

Podmíněný překlad (PP)

- Dočasné vynechání i větších částí zdrojového kódu
- Typické situace:
  - Vynechání ladících částí programu
  - Platformově závislé části zdrojového kódu
  - „Zakomentování“ větší části kódu (dříve)
- Preprocesor má několik možností, jak PP zajistit

PP řízený hodnotou výrazu

- Syntaxe obecně:

```
#if výraz
    část_1
#else
    část_2
#endif
```

- Pokud je *výraz* pravdivý, nechá se v programu jen *část_1*, jinak jen *část_2*
- Direktivu `#else` a *část_2* lze vynechat
- Příklad:

```
#define WINDOWS 1
#if WINDOWS
    #define JMENO "C:\\Data\\input.txt"
#else
    #define JMENO "/data/input.txt"
#endif
```

PP řízený definováním konstanty

- Syntaxe obecně:

```
#ifdef jméno
    část_1
#else
    část_2
#endif
```

- Pokud je *jméno* definováno, nechá se v programu jen *část_1*, jinak jen *část_2*
- Direktivu `#else` a *část_2* lze vynechat
- Obdobně funguje i direktiva `#ifndef`
- Příklad:

```
#define WINDOWS
#ifdef WINDOWS
    const char *cesta = "C:\\Data\\input.txt";
#else
    const char *cesta = "/data/input.txt";
#endif
```

Pokročilejší možnosti PP

- Pro konstrukci složitější podmínky lze použít
  - operátor `defined`
  - direktivu `#elif` (význam „else if“)
- Direktiva `#error` oznámí chybu již při zpracovaní preprocesorem
- Příklad:

```
#if defined(WINDOWS) && defined(DEBUG)
    #define LADENI 1
#elif !defined(DEBUG)
    #error Chyba nelze debugovat!
#else
    #define LADENI 2
#endif
```

Vkládání souborů

- Direktiva `#include` přidá celý soubor na dané místo
- Technicky lze vložit obsah jakéhokoli souboru
- Prakticky vkládáme jen tzv. hlavičkové soubory
- Vkládané soubory mohou být
  - Ve stejném adresáři
    `#include "název_souboru"`
  - V systémovém adresáři
    `#include <název_souboru>`

Vstupy a výstupy programu

Parametry a návratová hodnota programu

- Již víme, že `main` je funkce
- A také tušíme, že je „trochu jiná“ než ostatní funkce v programu
- Funkce `main` je volána operačním systémem při spuštění programu
- Při tomto volání mohou být programu předávány parametry
- Funkce `main` může mít
  - Žádný parametr
    ```
    int main(){ ... }
    ```
  - Nebo dva parametry
    ```
    int main(int argc, char* argv[]){ ... }
    ```
  - Typu int a pole textových řetězců
- Návratová hodnota `main` může být také zpracována operačním systémem

Předání parametrů programu

- Parametry lze předat při spuštění z příkazového řádku (či skriptem)
  ```
  ./xpm2eps -f nam.xpm -o nam.eps -by 10
  xpm2eps.exe -f nam.xpm -o nam.eps -by 10
  ```
- Do `argc` se načte počet řetězců
- Do pole `argv` se vloží uvedené řetězce (včetně jména programu)
- Jako oddělovače řetězců v příkazovém řádku slouží bílé znaky (mezery)
- Lze předat i řetězec obsahující mezery, stačí jej dát do uvozovek
  ```
  pokus.exe 125 "ahoj svete" neco
  ```
- Řetězce v poli `argv` se nedoporučuje měnit
- Pro konverzi textového řetězce na číslo lze použít
  - Funkci atoi: `int pocet = atoi(argv[1]);`
  - Funkci atof: `double cislo = atof(argv[2]);`

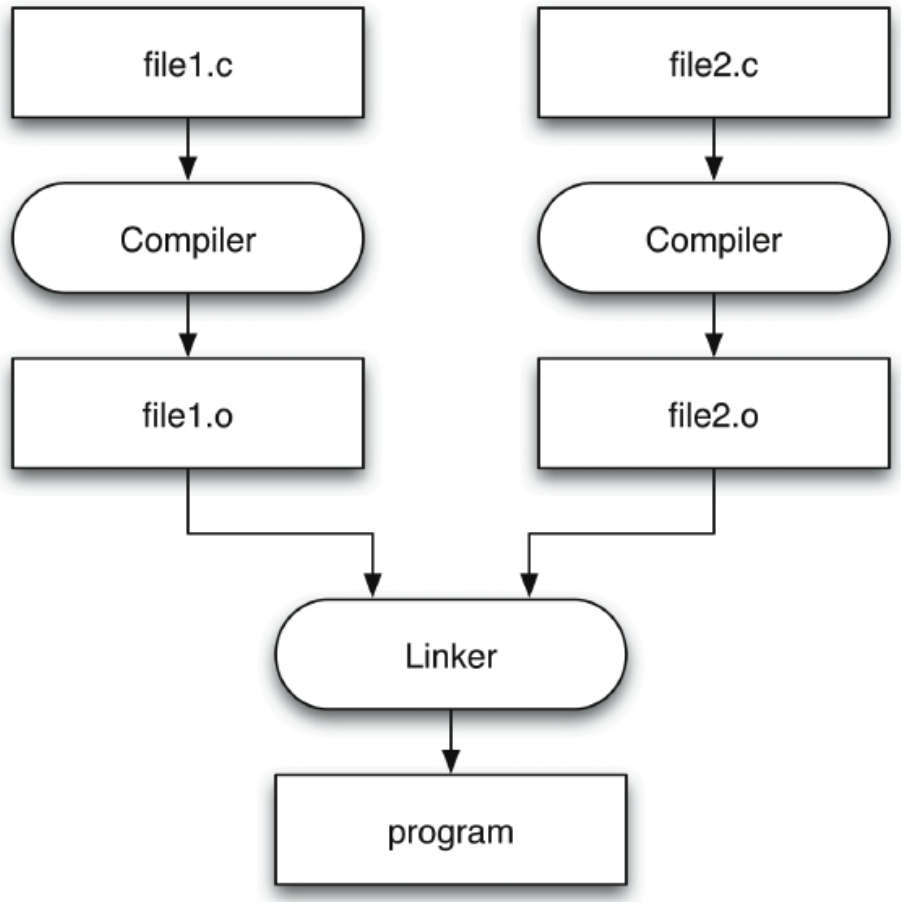
Celková koncepce programu

Koncepce větších programů

- Při vytváření rozsáhlejších programů je potřeba
  - Umožnit souběžnou práci více programátorů
  - Zajistit co největší přehlednost kódu
  - Minimalizovat množství kódu, který se kompiluje při drobných změnách (ušetřit čas)
- Bývá rozumné rozdělit zdrojový kód do více souborů
  - Každý soubor by měl být co nejvíce nezávislý na ostatních
  - Nelze se ale zcela vyhnout volání funkcí z jiných souborů
  - Ojediněle je potřeba i sdílení globálních proměnných nebe datových typů
  - Na druhou stranu je nutné zabránit nechtěnému sdílení při náhodném použití stejného identifikátoru na více místech

Oddělený překlad

- Překlad každého zdrojového souboru probíhá samostatně
- Z každého zdrojového souboru vznikne tzv. objektový soubor (OBJ soubor)
- Objektové soubory jsou poté spojeny do jednoho spustitelného souboru (EXE soubor)
- Stačí tedy překládat jen změněné zdrojové soubory
- Ve větším týmu lze objektové soubory sdílet místo zdrojových souborů
- Vývojové nástroje organizují soubory do projektů
- Překlad i spojení se pak provádí jediným kliknutím
- V prostředí Linuxu (UNIXu) se často používá utilita make



Hlavičkové soubory

- Zajišťují předání informací o zdrojových souborech, které jsou nezbytné už při kompilaci jiných zdrojových souborů
- Obsahují typicky informace
  - o funkcích, globálních proměnných, datových typech
  - které jsou používané i v dalších zdrojových souborech
- Mají příponu .h, název odpovídá zdrojovému souboru, který popisují
- Přidávají se pomocí direktivy `#include` do dalších souborů
- Problémy s vícenásobným vložením běžně řešíme pomocí

```
#ifndef JMENO_H
#define JMENO_H
...
#endif
```

Sdílení a obrana proti sdílení

- Sdílené proměnné a funkce definujeme v jediném zdrojovém souboru
- Jejich deklarace vložíme do hlavičkového souboru s označením extern

```
extern int pocet;
extern double fce(double);
```

- Pokud na programu pracuje více programátorů
  - Mohou náhodně zvolit stejné identifikátory
  - I když tyto proměnné a funkce nemají být sdílené
- Nesdílené globální proměnné a funkce necháme jen ve zdrojovém souboru a označíme je jako static

```
static int pocet = 12;
static double fce(double x){ ... }
```

- Použití static je silnější než označení extern v jiném souboru

Shrnutí – doporučení

- Obsah následujících snímků je převzat z knihy Pavla Herouta
- Větší program rozdělit do částečně samostatných modulů
- Modul obvykle tvoří hlavičkový a zdrojový soubor
- Mezi moduly sdílet co nejméně funkcí
  a nejlépe žádné globální proměnné
- Vše globální, co není sdílené, označit jako static
- Vše sdílené umístit do hlavičkového souboru modulu
  a přidat označení extern
- Hlavičkové soubory přidat pomocí include do dalších modulů
  (jen tam, kde jsou potřeba)
- Zásadně nepřidávat přímo zdrojové soubory

Doporučená struktura zdrojových souborů

1. Dokumentace – komentář (jméno modulu, verze, autor, datum, ...)
2. Všechny potřebné direktivy include
   - Nejprve všechny systémové hlavičkové soubory
   - Pak vlastní hlavičkové soubory
3. Definice sdílených globálních proměnných (tady bez extern)
4. Lokální direktivy define (konstanty a makra)
5. Definice nesdílených typů (nejlépe pomocí typedef)
6. Nesdílené globální proměnné označené jako static
7. Deklarace nesdílených funkcí (sdílené funkce nedeklarujeme, ale jen vložím hlavičkový soubor modulu)
8. Definice funkcí – doporučené pořadí: main (pokud je), sdílené, nesdílené s označením static)

Doporučená struktura hlavičkových souborů

1. Dokumentace – komentář (jméno modulu, verze, autor, datum, ...)
2. Podmíněný překlad proti opakovanému vložení souboru
   (vše co následuje je uvnitř konstrukce PP)
3. Případné direktivy include (pokud jsou nezbytné)
4. Definice sdílených konstant
5. Definice sdílených maker
6. Definice sdílených typů (nejlépe pomocí typedef)
7. Deklarace sdílených proměnných (s označením extern)
8. Deklarace sdílených funkcí (s označením extern)