B33POS: Počítačové systémy

Lekce 05. Make

Petr Štěpán
stepan@fel.cvut.cz

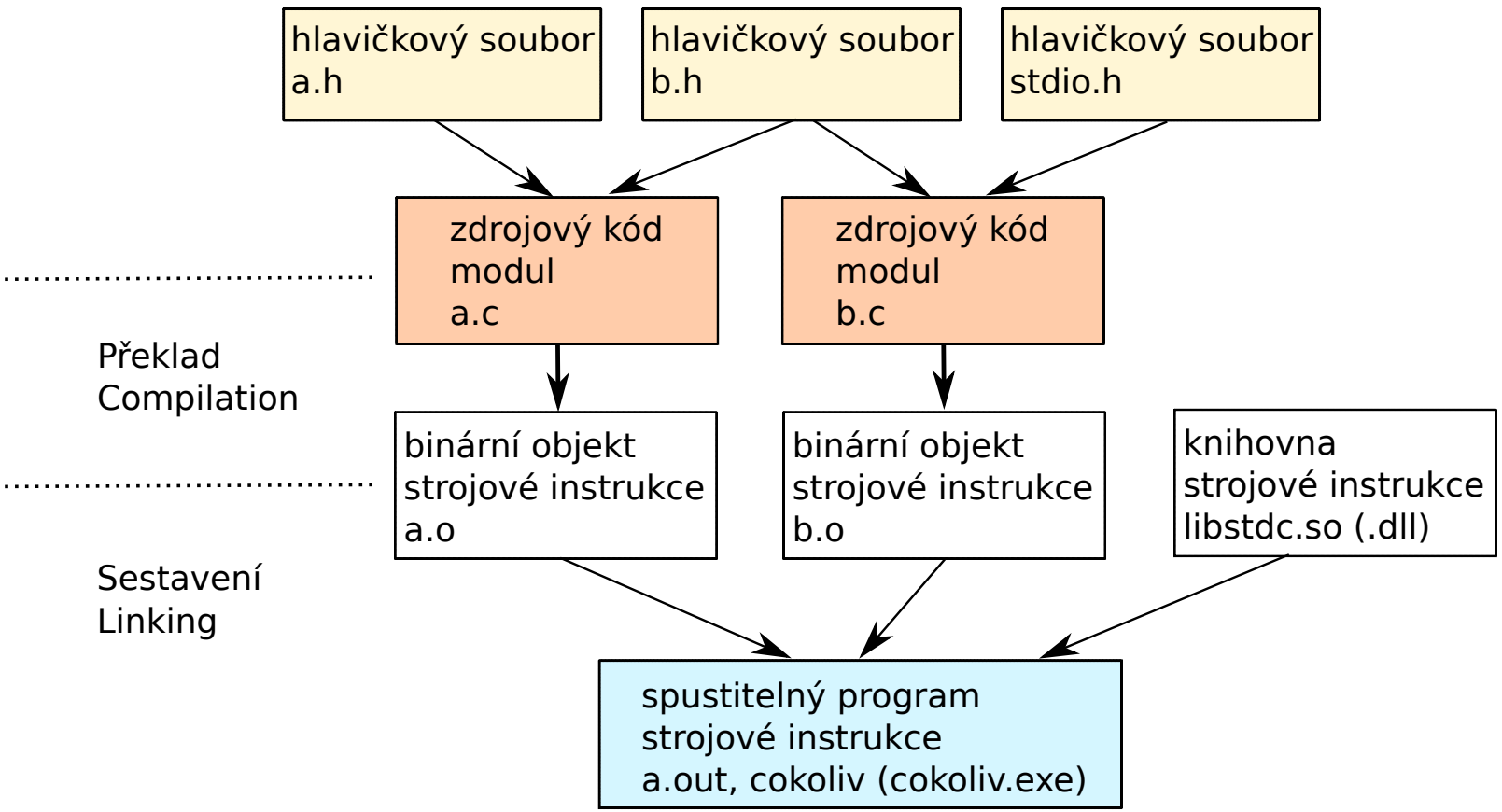
17. října, 2025

Obsah

Motivace

- V procedurálním programování PRP jste zatím měli programy, které byly složeny pouze z jednoho modulu, z jednoho zdrojového souboru.
- V této lekci se podíváme, co je nutné zajistit, pokud se Váš program skládá z více modulů
- Vysvětlíme si, co jsou hlavičkové soubory (header files), jaký mají význam a jak je správně psát
- Ukážeme si, základní problémy sestavování výsledného programu z více modulů a připojení knihoven k Vašemu programu
- Podíváme se na nástroj make a jak Vám může tento nástroj zpříjemnit překlad programů.

Překlad


hlavičkový soubor
a.h
hlavičkový soubor
b.h
hlavičkový soubor
stdio.h
zdrojový kód
modul
a.c
zdrojový kód
modul
b.c
Překlad
Compilation
binární objekt
strojové instrukce
a.o
binární objekt
strojové instrukce
b.o
knihovna
strojové instrukce
libstdc.so (.dll)
Sestavení
Linking
spustitelný program
strojové instrukce
a.out, cokoliv (cokoliv.exe)

Fáze vytváření programu

Jak je zobrazeno na předchozím obrázku, fáze vytváření programu jsou dvě:

- Překlad (Compilation) jednotlivých modulů na binární objektové soubory
  - Při překladu není nutné znát definici všech použitých funkcí a proměnných, ale je nutné vědět o jejich typech a velikostech.
  - Překlad vygeneruje strojové instrukce, které jsou definovány Vaším programem a tyto strojové instrukce se pak již nemění, pouze se mohou měnit jejich parametry.
    - Pro přístup k proměnné typu int, strojové instrukce musí nahrát do registrů procesoru hodnotu proměnné z konkrétní adresy v paměti.
    - Správná adresa proměnné není při překladu známá, protože se určí až při sestavování programu
    - Překladač tedy vygeneruje instrukce pro načtení proměnné, ale adresu proměnné nechá nulovou a ta se doplní až při sestavování
    - Překladač vygeneruje tabulku všech adres a jim odpovídajících symbolů, které se při sestavování musí nastavit na správné hodnoty

Fáze vytváření programu

a druhá fáze je:

- Sestavení (Linking) všech modulů a knihoven do výsledného programu
  - Každý modul má seznam symbolů, které jsou v něm definovány (export) a symbolů, které potřebuje najít v ostatních modulech nebo knihovnách (import)
  - Použité symboly jsou jména proměnných nebo funkcí, často se zakódovanými informacemi o vstupech a výstupech funkce.
  - Sestavovač (Linker) neví nic o typech symbolů, on hledá stejné názvy exportů a importů (vlastně se nestará ani o to, zda jde o funkci nebo proměnnou, vše jsou pro něj pouze adresy v paměti).
  - Linker spojí všechny programy (funkce) do jednoho bloku a všechna data do dalšího bloku a přiřadí jim adresy v paměti.
  - Tyto adresy vyplní na místa podle tabulek vygenerovaných při překladu.

Vyvolání překladu

Základní přepínače se neliší pro různé překladače (gcc, clang) :

- -c zajistí pouze překlad do .o souboru
- -o pojmenuje výstup překladače, buď .o soubor nebo celý program
- -I cesty pro nalezení hlavičkových souborů (navíc k systémovým hlavičkovým souborům)
- -L cesta pro nalezení knihoven (navíc k systémovým knihovnám)
- -l jméno knihovny pro připojení k programu

Některé další důležité přepínače si vysvětlíme později.

Vyvolání překladu

Překlad bez vytvoření .o souboru:

- gcc modul1.c -o jmeno_programu
- gcc modul1.c modul2.c -o jmeno_programu
- gcc -lm -lpthread modul1.c modul2.c -o jmeno_programu

Překlad s vytvořením .o souborů. Nejdříve vytvoříme pro každý modul jeho .o soubor:

- gcc -c modul1.c -o modul1.o

Pak vytvoříme program sestavením všech modulů:

- gcc -lm -lpthread modul1.o modul2.o -o jmeno_programu

Vytvoření .o souborů umožní, že se překládají pouze moduly, které se změnily. Nezměněné moduly použijí dříve vytvořené soubory .o.

Obsah

Hlavičkové soubory

Význam hlavičkového souboru:

- Umožnit překlad (tedy vytvoření programu ze strojových instrukcí), který bude správně provádět zapsaný program v jazyce C.
- Informace v hlavičkových souborech musí poskytnout veškerou informaci potřebnou pro "porozumění" programu.
  - Pro proměnné je nutné uvést jejich typ a jméno
    - Pro jednoduché proměnné lze uvést např.
      ```
      extern double koeficient_1;
      ```
    - Klíčové slovo `extern` je důležité, protože říká překladači, že dané proměnné bude v jiném modulu.
    - Pokud je typ struktura/třída/výčet, je potřeba uvést celou definici (plný obsah) všech položek struktury/třídy/výčet.
  - Pro funkce je nutné uvést typ návratové hodnoty a typy všech argumentů:
    - Pro funkce není nutné používat klíčové slovo `extern`, tedy
      ```
      double power(double, int);
      ```
    - Při deklaraci funkce není nutné uvádět názvy argumentů, ale z hlediska čitelnosti to může pomoci k rychlému porozumění významu argumentů
      ```
      double power(double x, int pow);
      ```

Hlavičkové soubory

Příklad hlavičkového souboru struktura_a.h:

```
#ifndef __struktura_a_h__
#define __struktura_a_h__

struct structura_a {
  int cele_cislo;
  int pole_celych_cisel[4];
  float realne_cislo;
}
extern struct struktura_a promenna_struct_a;
int kontrola(struktura_a test);

#endif
```

Hlavičkový soubor definuje strukturu_a, dále deklaruje existenci proměnné typu struktura a a deklaruje funkci, která jako argumnet používá struktura_a a výsledek je celé číslo.

Hlavičkové soubory

K čemu je dobrý začátek a konec hlavičkového souboru?

```
#ifndef __struktura_a_h__
#define __struktura_a_h__

...

#endif
```

Tato konstrukce zabraňuje zacyklení vkládání souboru struktura_a.h, pokud soubor v sobě zahrnuje vložení jiného souboru.
Příklad? Jak definovat strukturu a, která v sobě obsahuje ukazatel na strukturu b, když struktura b obsahuje ukazatel na strukturu a?

Hlavičkové soubory

Příklad cyklického vnoření hlavičkových souborů

```c
#ifndef __struktura_a_h__
#define __struktura_a_h__

struct struktura_a;
#include "struktura_b.h"

struct structura_a {
  int cele_cislo;
  struct struktura_b *b;
}
extern struct struktura_a a1;
#endif
```

```c
#ifndef __struktura_b_h__
#define __struktura_b_h__

struct struktura_b;
#include "struktura_a.h"

struct struktura_b {
  float realne_cislo;
  struct struktura_a *a;
}
extern struct struktura_b b1;
#endif
```

Při prvním vložení např. struktura_a.h se definuje symbol ___struktura_a_h___, dále se vloží se soubor struktura_b.h a definuje se symbol ___struktura_b_h___.

Nyní při vkládání souboru struktura_a.h se zjistí, že již symbol ___struktura_a_h___ je definován a vkládání se přeskočí až k *#endif*

Hlavičkové soubory - pravidla

Zde jsou pravidla, jak hlavičkové soubory používat:

- Každý modul by měl mít jeden svůj hlavičkový soubor
- Každý modul by měl vkládat svůj vlastní hlavičkový soubor, aby překladač provedl kontrolu konzistence deklarace a definice.
- Všechny globální proměnné a všechny funkce, které nejsou exportované v hlavičkovém souboru by měly být deklarovány a definovány v modulu jako static.
  - Všechny proměnné a funkce, které nejsou static se automaticky exportují v modulu a mohou způsobit nechtěné interakce.
- Nepoužívejte definice proměnných v hlavičkových souborech, i když je to možné obejít přepínačem -fcommon (viz následující slajd)

Společné symboly - Common symbols

- Při správném použití extern ve všech souborech a definici pouze v jednom modulu nehrozí žádné možnosti.
- Nastavení -fno-common otestuje, že je proměnná definována pouze v jednom modulu.
  - gcc od verze 10 používá nastavení -fno-common standardně, pro jeho vynutí je nutné nastavit -fcommon
  - gcc před verzí 10 je pro jednoznačnost symbolů nutné nastavit přepínač -fno-common
  - clang má sandardně nastaveno pravidlo jedné definice (One Definition Rule - ODR)
- Při chybné definici proměnné ve dvou modulech pak při sestavování programu linker nahlásí chybu multiple definition.
- Při chybějící definici proměnné pak při sestavování programu linker nahlásí chybu undefined reference.

Obsah

Make tool

Nástroj make byl vytvořen pro zefektivnění překladu:

- Na základě definovaných pravidel se provede aktualizace výsledného programu, tedy překlad pouze těch modulů, které byly aktualizovány
- Zda je soubor aktuální se řeší pouze na základě časů modifikace souborů
  - pokud byste použili touch, pak byste mohli nástroj make poplést a pak by nepracoval správně.

Make tool

Nástroj make pracuje na základě pravidel definovaných standardně v souboru Makefile (pokud soubor nazveme jinak, tak pak musíme zadat make -f nazev_souboru).

```
cíl: závislost závislost
<-TAB->příkaz
<-TAB->příkaz
<-TAB->příkaz
```

Pozor, znak tabulátor je povinný být před příkazem.
Některé editory nahrazují znak tabulátor definovaným počtem mezer. Pro vytvoření souboru Makefile je nutné editor nastavit, aby ponechával znaky TAB nezměněné v souboru.

Make tool

Příklad:

```
hello:
<-TAB->echo "Hello world"
<-TAB->echo "File hello does not exist"
```

Po zadání příkazu make, systém vytiskne Hello world a File hello does not exist.
Pokud soubor hello existuje, pak vytiskne make: ĥelloís up date.

Make tool

Příklad více cílů:

```
hello:
<-TAB->echo "Hello world"
<-TAB->touch hello

bye:
<-TAB->echo "Bye bye"
<-TAB->rm -f hello
```

Po zadání příkazu make bez parametru, systém se snaží splnit první uvedený cíl. Pokud hello neexistuje, pak vytiskne Hello world a vytvoří soubor hello.
Pokus soubor hello existuje, pak vytiskne make: 'hello' is up to date.
Pokud spustíme make bye, vytiskne se Bye bye a smaže se soubor hello.

Make tool

Příklad se závislostmi:

```
hello: bye
<-TAB->echo "Hello world"
<-TAB->touch hello

bye:
<-TAB->echo "Bye bye"
<-TAB->rm -f hello
```

Protože soubor bye neexistuje, použije pravidlo pro vytvoření "souboru" bye, které smaže soubor hello, tedy vždy se provedou obě pravidla bye i hello.

Make tool

Cíl bye z předchozího příkladu byl vlastně virtuální cíl, nechtěli jsme vytvářet soubor bye, pouze provést pravidla s cílem bye. Pokud by náhodou existoval soubor bye, tak by zablokoval provádění cíle bye.

```
hello: bye
<-TAB->echo "Hello world"
<-TAB->touch hello

.PHONY: bye clean

bye:
<-TAB->echo "Bye bye"
<-TAB->rm -f hello

clean:
<-TAB->rm *.o
<-TAB->rm program
```

Klíčové slovo .PHONY znamená, že makefile nebude hledat soubor bye, ale pokud bude bye v nějakém pravidle, tak se vždy provede.

Nejběžnějším .PHONY pravidlem je clean, který smaže všechny soubory, které se v make tvořily.

Make tool

Příklad s proměnnými:

```
CC:=gcc
CFLAGS:=-Wall -fno-common
soubory := a.c b.c
hello: $(soubory)
<-TAB->$(CC) $(CFLAGS) $^ -o $@
```

Pokud program hello nexistuje spustí se program gcc -Wall -fno=common a.c b.c -o hello, který pokud jsou moduly a.c a b.c bez chyby vytvoří program – soubor hello.
I když je přístup k proměnným jiný než v Bashi, tak v make zná bashové proměnné.

Make tool

Přiřazení proměnným:

`CC:=gcc`
Okamžité vyhodnocení, neboli vyhodnotím na místě přiřazení

`OBJS=$(mask)`
Líné vyhodnocení, hodnota OBJS se určí, až když se bude OBJS používat a to pokaždé znovu.

`CC?=clang`
Pokud není CC definován (ani v bashi), pak nastav CC=clang.

`CC+=-Wall`
Přidej k uvedenou hodnotu k proměnné. Pokud proměnná nebyla ještě nastavena, tak se nastaví na zadanou hodnotu.

Implicitní pravidla

Implicitní pravidla (pravidla, která nemusíte napsat a přesto existují):

```
CC:=gcc
CFLAGS:=-Wall -fno-common
LDFLAGS:=-lm
objs := a.o b.o
hello: $(objs)
```

Proměnné používané implicitními pravidly:

- CC program pro překlad C programů; defaultně cc
- CXX program pro překlad C++ programů; defaultně g++
- CFLAGS extra příznaky pro překladač C
- CXXFLAGS extra příznaky pro překladač C++
- CPPFLAGS: extra příznaky pro předzpracování C
- LDFLAGS: extra příznaky pro linker (tedy hlavně přidání knihoven, případně cest ke knihovnám)

Implicitní pravidla

Implicitní pravidla jsou tři a mají tento tvar:
Překlad modulů v jazyce C:

```
%.o: %.c
<-TAB->$(CC) -c $(CPPFLAGS) $(CFLAGS) $^ -o $@
```

Překlad modulů v jazyce C++:

```
%.o: %.cc %.cpp
<-TAB->$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $^ -o $@
```

Spojení modulů do programu (linking):

```
PRG: OBJS
<-TAB->$(CC) $(LDFLAGS) $^ $(LOADLIBES) $(LDLIBS) -o $@
```

Závislosti souborů

V běžném projektu chceme většinou zkompilovat všechny .c a .cc nebo .cpp soubory na .o soubory a následně všechny .o soubory spojit do jednoho programu.

- K tomu lze použít implicitní pravidla, případně si nadefinovat svoje překládací a linkovací pravidla.
- Jak ale vyřešit závislost na hlavičkových souborech?
  - Závislosti se mohou v průběhu vývoje programu měnit
  - Můžete přidat hlavičkový soubor nového modulu nebo nově použité funkce jiného modulu
  - Někdy můžete i odebrat hlavičkový soubor, protože činnost bude dělat jiný modul.

Závislosti souborů - pšotrosí přístup

- Závislosti na hlavičkových souborech nebudete řešit v Makefile.
- Vždy při provedení libovolné změny v hlavičkových souborech, dáte make clean all. Tím vytvoříte všechny moduly znovu.
- Nevýhody:
  - Pro velké projekty může změna hlavičkového souboru znamenat dlouhý překlad, klidně i hodiny.
  - Pokud zapomenete dát make clean, může to vést k nekonzistentnímu programu, tedy programu, který je sice správně napsaný, ale špatně přeložený a tím pádem nefunkční.
- Výhody:
  - Velmi jednoduchý makefile

Závislosti souborů - ruční přístup

- Závislosti na hlavičkových souborech nadefinujete ručně v Makefile.

```
a.o: a.c a.h c.h d.h
<-TAB->$(CC) -c $(CPPFLAGS) $(CFLAGS) $^ -o $@
```

- Nevýhody:
  - Pro velké projekty je ruční definice pracná a můžete udělat chybu.
  - Při změně používání hlavičkových souborů musíte změnit makefile.
- Výhody:
  - Makefile obsahuje správné závislosti a překládá pouze nutné změny.

Závislosti souborů - makedpend

- Použijeme nástroj makedepend, který vloží závislosti do Makefile.
- Je nutné zadat všechny zdrojové soubory.

```
CC:=gcc
CFLAGS:=-Wall -fno-common
SRCS := $(wildcard *.c)
OBJS := $(SRCS:.c=.o)
hello: $(OBJS)
<-TAB->$(CC) $(CFLAGS) $^ -o $@
depend:
<-TAB->makedepend $(SRCS)
# DO NOT DELETE THIS LINE -- make depend depends on it.
```

Závislosti souborů - využití překladače

- Použijeme nástroj makedepend, který vloží závislosti do Makefile.
- Je nutné zadat všechny zdrojové soubory.

```
CC:=gcc
CFLAGS:=-Wall -fno-common
SRCS := $(wildcard *.c)
OBJS := $(SRCS:.c=.o)
DEPS := $(OBJS:.o=.d)
CPPFLAGS += -MMD -MP
hello: $(OBJS)
<-TAB->$(CC) $(CFLAGS) $^ -o $@
# DO NOT DELETE FOLLOWING LINE
-include $(DEPS)
```

Testovací kvíz

Kolik jste toho zatím pochopili?

A Vše bez problému.

B Téměř vše.

C Téměř nic.

D Vůbec nic.