Jazyk C: Proces překladu

2

V tomto cvičení se zaměříme na praktické aspekty procesu překladu programů v jazyce C do spustitelné podoby. Jazyk C bereme pouze jako vzorový jazyk a zde nastíněné principy lze aplikovat i na další kompilované programovací jazyky.[1] Značně zjednodušeně řečeno, cílem je ukázat, co se skrývá pod zeleným trojúhelníčkem *Compile & Run* známým z vývojových prostředí. Pro demonstraci budeme používat překladač gcc a nástroje z operačního systému GNU/Linux, avšak podobné principy lze aplikovat i na další překladače, např. Clang/LLVM, MSVC.[2]

1 Překlad programu

Uvažujme jednoduchý příklad typu „Hello World".

```
// hello.c
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World!\n");
    return 0;
}
```

V nejjednodušší variantě program přeložíme a spustíme následovně:

```
gcc hello.c -o hello
./hello
```

V tomto případě název `hello.c` odpovídá zdrojovému kódu programu a `hello` výslednému binárnímu (spustitelnému) programu.

1.1 Fáze překladu

Ač to nemusí být na první pohled zjevné, zdrojový kód při svém překladu prochází několika postupnými transformacemi, které jsou skryty pod jedno spuštění překladače příkazem `gcc`.

[1] Jednotlivé jazyky mohou mít své odlišnosti.
[2] Jednotlivé nástroje mohou mít své odlišnosti.

1. Nejdříve je spouštěn preprocesor (příkaz `cpp`), který vloží hlavičkové soubory (direktiva `#include`), expanduje makra (direktiva `#define`), odstraní části kódu nesplňující danou podmínku (direktiva `#ifdef`), odstraní komentáře apod . Výstupem preprocesoru je kód programu, který obsahuje pouze kód v jazyce C.

2. V druhé fázi překladač přeloží zdrojový kód v jazyce C do jazyka symbolických adres, tj. vytvoří zápis programu v podobě jednotlivých instrukcí procesoru. Tyto instrukce jsou zapsány v textové podobě.

3. Kód v jazyce symbolických adres je nástrojem, který *assembler*[3] (příkaz `as`) přeložen do podoby objektového souboru.[4] Objektové soubory obsahují program přeložený do strojového kódu a další související informace jako jsou konstanty, informace o poskytovaných symbolech (funkcích, proměnných), ladící informace apod.

4. V poslední fázi překladu jsou objektové soubory sloučeny (příkaz `ld`) a spojeny s knihovnami[5] a je vytvořen výsledný binární soubor.

Jak vypadá kód v jednotlivých fázích překladu, můžeme zjistit pomocí přepínačů příkazu `gcc`.

1.1.1 Preprocessor

Výsledek zpracování zdrojového souboru preprocessorem můžeme získat příkazem:

```
gcc -E hello.c
```

Na výstupu z preprocessoru si všimněme jednak absence komentářů a toho jaký kód byl vložen z hlavičkového souboru `#include <stdio.h>`. Když si například dohledáme funkci `printf`, vidíme, že je ve zdrojovém kódu přítomen jen její prototyp, nikoliv celá funkce.

1.1.2 Překlad do jazyka symbolických adres

Jak vypadá program přeložený do jazyka symbolických adres, zjistíme s pomocí přepínače `-S`.

```
gcc -S hello.c
```

V tomto případě překladač vygeneruje soubor `hello.s`, který obsahuje jednotlivé instrukce procesoru zapsané v jazyce symbolických adres a další dodatečné informace. Podrobněji se jazyku symbolických adres budeme věnovat v následujících cvičeních.

[3] Původně slovo *assembler* označovalo pouze nástroj, který vzal program v tzv. *jazyce symbolických adres* a přeložil jej do strojového kódu. Postupně se označení assembler přeneslo i na jazyk symbolických adres a dnes naprosto běžně pojem assembler označuje jak nástroj, tak i jazyk popisující program na úrovni jednotlivých instrukcí.

[4] Název objektový soubor nijak nesouvisí s objektově orientovaným programováním.

[5] Minimálně standardní knihovnou jazyka C.

1.1.3 Překlad do objektového souboru

V předchozí fázi překladu jsme získali program, který již nabyl podoby jednotlivých instrukcí procesoru, ale je nutné jej přeložit do strojového kódu. To obstará assembler, který vygeneruje objektový kód. Objektový kód získáme přepínačem `-c`.

```
gcc -c hello.c
```

Překladač vygeneruje soubor `hello.o`, který již obsahuje přeložený strojový kód. Jelikož se jedná o binární formát dat, není možné[6] jej studovat prostým zobrazením. Můžeme to posoudit zobrazením pomocí nástroje `hexdump`.

```
hexdump -C hello.o
```

Chceme-li prozkoumat obsah objektového souboru, musíme použít vhodný nástroj, kterým je například `objdump`. Nástroj `objdump` umožňuje zobrazit jednotlivé logické části objektového souboru, tzv. sekce.

```
objdump -s hello.o
```

Ve výpisu bychom měli minimálně vidět sekci `.text`, která obsahuje program přeložený do strojového kódu, a sekci `.rodata`, která obsahuje data programu, která jsou jen pro čtení, v našem případě řetězec `Hello World!`.

Další užitečnou funkcí, kterou nástroj `objdump` nabízí, je tzv. disassembling, opačný proces k assembleru, kdy je strojový kód přeložen zpět na svou textovou reprezentaci ve formě jazyka symbolických adres. Tato funkcionalita se skrývá pod přepínačem `-d`, případně ve spojení s přepínačem `-M intel`, který zajistí zobrazení kódu v syntaxi, jak byla použita v průběhu přednášky.

Pokud provedeme následující příkaz:

```
objdump -d -M intel hello.o
```

Uvidíme, že objektový soubor opravdu obsahuje jen námi vytvořenou funkci a neobsahuje kód funkce `printf`.

1.1.4 Vytvoření spustitelného souboru

Při vytvoření spustitelného souboru dochází k tomu, že jednotlivé objektové soubory, ze kterých se program skládá,[7] jsou sloučeny společně s knihovnami. K vytvoření výsledného souboru můžeme použít buď příkaz `gcc` nebo zavolat tzv. *linker* (příkaz `ld`).

```
gcc -o hello hello.o
```

Což odpovídá přibližně:

[6] Nebo minimálně pohodlné.

[7] V ukázkovém příkladu se program skládá z právě jednoho objektového souboru.

```
ld -o hello hello.o /lib64/crt1.o --dynamic-linker /lib64/ld-linux-x86-64.so.2 -lc
```

V tomto případě linker spojí vstupní objektové soubor se souborem `crt1.o` (C runtime), který obsahuje funkce nutné pro běh programu v C, společně se standardní knihovnou jazyka C (přepínač `-lc`). Přepínač `--dynamic-linker` zajišťuje, že knihovna jazyka C bude připojena dynamicky. Podrobněji to bude vysvětleno na přednášce.

Ze zápisu obou variant je zjevné, že první varianta vytvoření binárního souboru je komfortnější.

1.2 Nástroje pro překlad programu

Rozdělení překladu do jednotlivých fází, zejména generování objektových souborů, umožňuje rozdělit překlad do logických celků, tj. mít související funkce v oddělených souborech a v případě změny překládat jen změněné soubory. Aby se daly větší programy pohodlně překládat, vznikl nástroj `make`, který sestaví program dle zadaných pravidel. Pravidla pro překlad se zapisují do souboru, který se jmenuje `Makefile`, a jednotlivá pravidla mají následující tvar:

```
cil: zavislosti
<TAB!>prikaz pro sestaveni cile
<TAB!>prikaz pro sestaveni cile
```

Pro náš ukázkový příklad by `Makefile` mohl vypadat následovně.

```
hello: hello.o
	gcc -o hello hello.o

hello.o: hello.c
	gcc -c hello.c
```

První pravidlo udává, že pro sestavení programu `hello` potřebujeme mít soubor `hello.o`, a pokud jej máme, program se přeloží pomocí `gcc -o hello hello.o`. Druhé pravidlo říká, že pro vytvoření programu `hello.o` potřebujeme `hello.c` a výsledný soubor získáme příkazem `gcc -c hello.c`.

Program přeložíme pomocí příkazu `make`. Je důležité, že nástroj `make` automaticky vyřeší všechny závislosti a spustí příkazy ve správném pořadí.[8] Současně nástroj `make` zajistí, že se překládají jen změněné části programu a ty, které na nich závisí.

1.2.1 Sestavení většího programu

V tomto jednoduchém příkladu je použití nástroje `make` příslovečný kanón na vrabce, ale již u mírně větších projektů se ukazuje jeho užitečnost.

Předpokládejme, že budeme chtít mít nějaké funkce[9] v odděleném souboru se zdrojovými kódy a ty volat z programu `hello.c`.

V takovém případě budeme postupovat následovně.

[8] S pomocí přepínače `-j` je možné překlad spustit i paralelně.

[9] Pro jednoduchost budeme uvažovat jen jednu funkci pro výpočet faktoriálu

1.2.2 Hlavičkový soubor

Nejdříve vytvoříme hlavičkový soubor, nazvěme jej `myfuncs.h`. Tento hlavičkový soubor by měl obsahovat deklaraci prototypu funkce.

```
// myfuncs.h
#ifndef MYFUNCS_H
#define MYFUNCS_H

/* funkce pro vypocet faktorialu */
unsigned int fact(unsigned int n);

#endif
```

Všimněme si, že v hlavičkovém souboru deklarujeme jen název funkce, typy argumentů a typ návratové hodnoty. V hlavičkovém souboru se nenachází tělo funkce. Dále stojí za povšimnutí direktivy preprocesu na začátku souboru. Ty zajišťují, že hlavičkový soubor je do kódu vložen nanejvýš jednou.

Kód funkce je pak uveden v samostatném souboru `myfuncs.c`.

```
// myfuncs.c
#include "myfuncs.h"

unsigned int fact(unsigned int n)
{
        if (n == 0) return 1;
        return n * fact(n - 1);
}
```

Použití funkce pro výpočet faktoriálu se neliší od použití funkcí, např. ze standardní knihovny, tj. vložíme hlavičkový soubor a voláme funkci běžným způsobem.

```
// hello.c
#include <stdio.h>
#include "myfuncs.h"

int main(int argc, char **argv)
{
        printf("5! = %i\n", fact(5));
        return 0;
}
```

Při překladu doplníme do `Makefile` pravidlo pro překlad `myfuncs.o` a doplníme jej jako závislost pro sestavení programu `hello`.

```
hello: hello.o myfuncs.o
	gcc -o hello hello.o myfuncs.o

hello.o: hello.c
	gcc -c hello.c

myfuncs.o: myfuncs.c myfuncs.h
	gcc -c myfuncs.c
```

Při sestavování programu bývá zvykem nastavit přepínače udávající chování překladače, např. míru optimalizace (přepínač `-O1` až `-O3`), standard jazyka (např. `-std=c99`), zobrazení upozornění (přepínač `-W`) nebo generování ladicích informací (přepínač `-g`). Abychom tyto volby mohli nastavit jednotně, podporuje nástroj `make` proměnné podobně jako například unixový shell. Ukázkový `Makefile` bychom mohli vylepšit následovně.

```
CFLAGS=-O1 -Wall -std=c99 -g

hello: hello.o myfuncs.o
	gcc $(CFLAGS) -o hello hello.o myfuncs.o

hello.o: hello.c
	gcc $(CFLAGS) -c hello.c

myfuncs.o: myfuncs.c myfuncs.h
	gcc $(CFLAGS) -c myfuncs.c
```

2 Programování v assembleru

Způsob překladu popsaný v předchozí kapitole má ještě jeden zásadní důsledek a tím je možnost vytvářet programy, jejichž jednotlivé části jsou napsány v různých jazycích. Jelikož objektové soubory obsahují přeložený strojový kód ve standardizovaném formátu, je možné je vytvářet v různých jazycích a následně je linkerem nechat spojit do spustitelného formátu. Pro nás je to zajímavé z toho důvodu, že jedním z těch jazyků může být jazyk symbolickcýh adres.

2.1 Program v jazyce symbolických adres a jeho překlad

Pro překlad programu v jazyce symbolických adres budeme používat assembler `nasm`, který je mírně přívětivější než nástroj `as`.[10] Ukážeme si to na jednoduchém příkladu funkce vracející hodnotu `42`.

```
; soubor demo.asm
global foo
```

[10] Tento nástroj vzniknul primárně pro potřeby překladačů a nemusí být úplně uživatelsky přívětivý.

```
section .text
foo:
        mov eax, 42
        ret
```

Ukázkový soubor obsahuje několik základních částí.

Na začátku máme komentář vyznačený znakem středník. Následuje direktiva `global`, která označuje jaké symboly (v terminologii C: funkce a proměnné) daný zdrojový soubor poskytuje, v našem případě to bude funkce `foo`. Následuje vyznačení sekce `.text`, která obsahuje kód programu zapsaný v jazyce symbolických adres. Naše funkce je vyznačena pomocí návěstí `foo:` (identifikátor + dvojtečka) a následuje kód samotné funkce.

Funkce se skládá ze dvou instrukcí. První instrukce uloží do registru `eax` návratovou hodnotu. Jedná se o konvenci, kdy celočíselné návratové hodnoty jsou předávány registrem `rax`, resp. `eax`, dle velikosti typu návratové hodnoty. Návrat z funkce je realizován instrukcí `ret`. Podrobněji se volání funkcí budeme věnovat na přednášce a v následujících cvičeních.

Překlad program v jazyce symbolických adres zajistíme příkazem `nasm`.

```
nasm -f elf64 demo.asm
```

Překladač v tomto případě vygeneruje soubor `demo.o`, který je ve formátu `elf64`, což je formát, který implicitně používá i překladač `gcc`.

2.2 Spojení s programem v jazyce C

Na straně jazyka C použijeme prostředky, které již známe. Nejdříve deklarujeme protyp funkce[11] a tuto funkci zavoláme.

```
#include <stdio.h>

/* prototyp funkce napsane v assembleru */
int foo();

int main()
{
        printf("Answer to life, etc.: %i\n", foo());
        return 0;
}
```

Samotné provázání funkcí je realizováno v linkovací fázi a pro pohodlné sestavení můžeme použít nástroj `make`, kdy do `Makefile` vložíme pravidlo pro překlad souboru v jazyce symbolických adres:

```
hello: hello.o demo.o
        gcc -o hello hello.o demo.o
```

[11] Mohli bychom jej umístit do samostatného hlavičkového souboru.

```
hello.o: hello.c
        gcc -c hello.c

demo.o: demo.asm
        nasm -f elf64 demo.asm
```

3 Úkoly k procvičení

1. Vezměte funkce `int2bits` a `bits2int` a umístěte je do samostného souboru `bits.c` a vytvořte odpovídající hlavičkový soubor `bits.h`.
2. To samé proveďte pro funkce `encode_date` a `decode_date` a ty umístěte do souborů `dates.[ch]`.
3. Vytvořte program, který výše popsané funkce bude používat.
4. Pro překlad programu vytvořte vhodný *makefile* a program přeložte.
5. S pomocí nástrojů a přepínačů překladače popsaných v přiloženém textu se podívejte na jednotlivé fáze překladu.
6. Na základě vzoru v přiloženém textu napište v jazyce symbolických adres funkci, která do registru `edi` uloží hodnotu 10 představující jednu stranu obdélníka, do registru `esi` uloží hodnotu 17 představující druhou stranu obdélníka a vrátí obvod obdélníka s těmito rozměry.
7. Výše popsanou funkci vyzkoušejte.