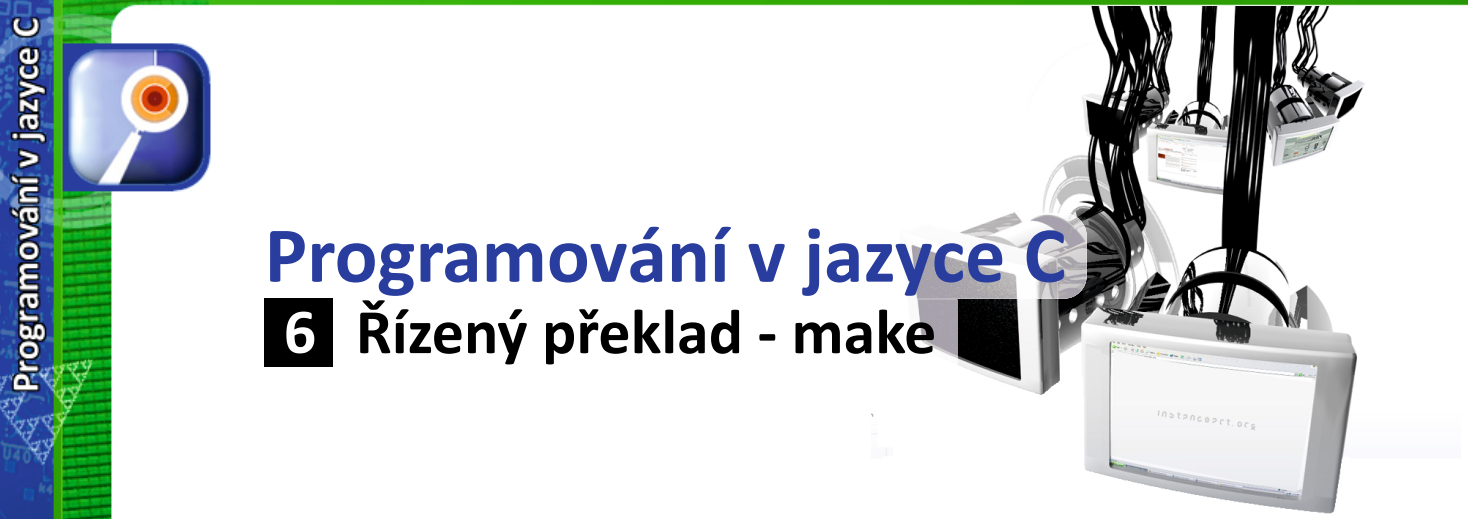

Programování v jazyce C

6 Řízený překlad - make

- Lexikální konvence make
- Konstrukce `makefile`
- Pravidla, cíle
- Symbolické cíle
- Makra
- Návaznost na operační systém

Nástroj make
Skriptem řízený překlad a sestavení projektu

- **`make`** je původně UNIXový pomocný vývojový nástroj, vytvořený v AT&T kolem roku 1975.
- Účelem je zjednodušit a zautomatizovat překlad a sestavení (*linking*) větších projektů, kde existují závislosti, tj. je nutné zdrojový kód překládat v určitém pořadí, které však není bez hlubšího zkoumání zjevné.
- Zároveň částečně dokumentuje mechanismus sestavení projektu (závislosti modulů, atp.) a umožňuje provést např. i migraci či instalaci
- Programovat v jazyce C bez využití nástroje **`make`** lze, ale **není to dobrý nápad!**
- Překlad pomocí **`make`** je řízen souborem **`makefile`**.
- **`makefile`** je prostý textový soubor (ASCII), který obsahuje (i) **komentáře**, (ii) **pravidla** (*Rules*), (iii) **cíle** (*Targets*), (iv) **makra** a (v) **příkazy** (*Commands*) zapsané danou formou.

Komentáře
Komentování obsahu `makefile`

- Komentář začíná znakem '**#**' (hashmark) kdekoliv na řádce a končí na konci řádky (tj. nelze vkládat mezi příkazy).

```
# The BIN and LIB macros define the
# output directories for binaries and
# compiled units.
DCC = dcc32 -q # set the C compiler
TASM = tasm32
BRCC = brcc32
...
```

- Zvláště složitější **`makefile`** je **nutné dobře komentovat**, aby plnil i svojí „dokumentační" funkci.

Pravidla (*Rules*)
Zápis postupu vedoucího ke splnění cíle

- **Pravidlo definuje**, co je potřeba (**závislosti**) a jakým postupem dojde k **vytvoření** daného **cíle** (např `.exe` souboru).

```
myprog.exe: myprog.c module.c
⊢TAB⇥ gcc myprog.c module.c -o myprog.exe
↵
```

- Obecný tvar pravidla je:

cíl : **řádka závislostí**

⊢TAB⇥ **příkaz**
⊢TAB⇥ **příkaz**
...
⊢TAB⇥ **příkaz**
↵

Zde **musí** být tabulátor!

jaké soubory se budou předávat příkazům

posloupnost příkazů končí prázdnou řádkou

Je nutné použít vhodný editor, který nenahrazuje tabulátory mezerami

Pravidla (*Rules*)
Zápis postupu vedoucího ke splnění cíle

- Pro každý cíl by mělo být definováno právě jedno pravidlo, ale existují výjimky:

```
target1: file1.c file2.c file3.c file4.c
↵
target1: file5.c file6.c file7.c
⊢TAB⇥ gcc -c $^
↵
```

tzv. **automatická proměnná**, tato konkrétní znamená „celý seznam závislostí s mezerami mezi jednotlivými jmény"

- Pokud je stejný cíl definován více pravidly, pak **jen jediné může obsahovat příkazovou část** (ostatní jen závislosti, které se při vykonání spojí do jediného seznamu).

Pravidla (*Rules*)
Zápis postupu vedoucího ke splnění cíle

- Pro každý cíl by mělo být definováno právě jedno pravidlo, ale existují výjimky:

```
mylib.lib:: part1.c part2.c
⊢TAB⊣ gcc -c $^
⊢TAB⊣ ar rv mylib.lib part1.o part2.o
↵
mylib.lib:: part3.c part4.c
⊢TAB⊣ gcc -c $^
⊢TAB⊣ ar rv mylib.lib part3.o part4.o
↵
```

- Technika ***Double Colon*** dovoluje mít více pravidel (i s příkazy) pro stejný cíl → **nezáleží na pořadí** jejich vykonání, nejsou na sobě závislé.
- Typické použití je (↑) přidávání modulů do nějaké knihovny.

Explicitní pravidla
Pravidla, která **`make`** používá při plnění cílů

- Pravidla mohou být (i) **explicitní** nebo (ii) **implicitní** (dále).
- **Explicitní** jsou běžná pravidla, která definují splnění **jediného konkrétního (vyjádřeného) cíle**:

```
main.exe: frontend.c engine.c control.c\
      model.c output.c
⊢TAB⊣ gcc $^ -o $@
↵
support.exe: support.c model.c output.c
⊢TAB⊣ gcc $^ -o $@
↵
```

automatická proměnná s významem
„úplné jméno (včetně přípony) cíle”

zpětné lomítko značí pokračování příslušné části pravidla (zde seznamu závislostí) na další řádce

Implicitní pravidla
Pravidla, která **`make`** používá při plnění cílů

- **Implicitní pravidla** jsou obecná pravidla, která definují, jak vznikají **celé množiny cílů** → soubory určitého typu (daného příponou):

přípona zdrojového souboru
přípona cílového souboru

```
.c.o:
	gcc -c $<
.o.exe:
	gcc $< -o $@
```

automatická proměnná s významem „první položka ze seznamu závislostí"

Toto implicitní pravidlo není zcela bezpečné: Ne z každého objektového souboru lze udělat spustitelnou aplikaci!

Cíle (*Targets*)
Výsledné zkompilované soubory

- **Cíl** (není-li tzv. *symbolický*) určuje, co je výsledkem příslušného pravidla, tj. co vznikne, je-li toto pravidlo vykonáno.
- V **`makefile`** může být definováno mnoho cílů, **nesmí být pojmenovány shodně**.

```
myapp.exe: myapp.o gui.o engine.o
⊢TAB⊣ gcc $^ -o myapp.exe
↵

...
myapp.o: myapp.c
⊢TAB⊣ gcc -c $^
↵
```

pravidlo vlastně definuje **cíl** (viz dříve) – zde je cílem vytvoření spust. souboru **`myapp.exe`**

- Je-li v seznamu závislostí **soubor** (tj. cíl jiného pravidla), **který** v souborovém systému **neexistuje**, hledá se pravidlo, jež ho vytvoří (pokud není nalezeno, hlásí **`make`** chybu).

Symbolické cíle (*Symbolic Targets*)
Cílem nemusí být nutně jen soubor

- **Symbolický cíl** umožňuje vykonat několik akcí současně, např. vytvořit několik spustitelných souborů.
- V **`makefile`** může být definováno více symbolických cílů – **`make`** vykoná pravidlo pro ten, který je **první**, nebo ten, který určíme na příkazové řádce jako parametr.

```
all: myapp.exe install.exe
↵
…
myapp.exe: myapp.o gui.o engine.o
⊢TAB⊣ gcc $^ -o $@ -lm
↵
install.exe: install.o diskio.o
⊢TAB⊣ gcc $^ -o $@
↵
```

pravidlem nebude vytvořen soubor **`all`**, ale soubory **`myapp.exe`** a **`install.exe`**

Symbolické cíle (*Symbolic Targets*)
Určení symbolického cíle, je-li jich více

- **Symbolický cíl** musí mít jedinečné jméno (soubor takového jména nesmí ve složce s projektem existovat), které **musí vyhovovat požadavkům OS na pojmenovávání souborů**.

```
all: myapp.exe install.exe
↵
…
clean:
⊢TAB⊣ del *.o
⊢TAB⊣ del myapp.exe
⊢TAB⊣ del install.exe
↵
```

žádné závislosti – smazat soubory lze vždy

```
>make clean
```

parametrem **make** lze určit, který symbolický cíl se má splnit…

- Uživatel **nemusí parametr** na příkazové řádce **uvést** → na prvním místě tedy musí být platný (proveditelný) symbolický cíl (zde cíl **all**).

```
>make all = >make
```

Pořadí cílů
Na pořadí záleží!

```
test.o: test.c
     gcc -c test.c

test.exe: test.o
     gcc test.o -o test.exe
```

```
test.exe: test.o
     gcc test.o -o test.exe

test.o: test.c
     gcc -c test.c
```

- **Záludná otázka:** Který z těchto 2 **`makefile`** vytvoří spustitelnou aplikaci **`test.exe`**, vzniklou překladem ZK **`test.c`**?

Pořadí cílů
Na pořadí záleží!

```
test.o: test.c
     gcc -c test.c

test.exe: test.o
     gcc test.o -o test.exe
```

soubor **`test.c`** je k dispozici, není tedy třeba plnit další cíle...

```
test.exe: test.o
     gcc test.o -o test.exe

test.o: test.c
     gcc -c test.c
```

soubor **`test.o`** neexistuje – make tedy hledá pravidlo k jeho vytvoření...

- **Odpověď:** Samozřejmě, že ten **druhý** – neurčí-li uživatel cíl, plní se první nalezený cíl, což je v tomto případě pouze překlad do objektového kódu...

Makra v `makefile`
Možnost definovat „konstanty”

okolo znaku ‘=’ nesmí být mezery

```
SRCEXT=.c
SOURCES=module1.c module2.c module3.c
OPTIONS=-lm -O2
```

- **`make`** je **case-sensitive** → **`myconst`** a **`MYCONST`** jsou 2 různá makra. Dodržuje se C-úzus „konstanty velkými písmeny”.
- Jméno makra max. 512 znaků, tělo makra max. 4096 znaků.
- Makro lze nadefinovat i „zvenčí”, parametrem na příkazové řádce:

```
>make -DSOURCES="main.c gui.c"
```

- Je-li makro stejného jména definováno i v **`makefile`**, použije se to z **`makefile`** (je to proto, aby případnou chybou na příkazovém řádku nedocházelo ke znemožnění sestavení celého projektu).

Vestavěná makra

„Konstanty”, které **`make`** poskytuje uživateli

POZOR! V různých verzích **`make`** se tato makra mohou významně lišit...

Makro	Význam
`AS`	assembler
`CC`	překladač C
`MAKE`	příkaz, kterým byl spuštěn make
`MAKEFLAGS`	parametry make z příkazové řádky

Makro	Význam
`$@`	jméno cíle
`$%`	jméno cíle (jako součást archivu)
`$<`	jméno prvního souboru ze seznamu závislostí
`$?`	celý seznam závislostí (novějších než cíl)
`$^`	všechny závislosti oddělené mezerami

- Existuje i řada dalších vestavěných maker a automatických proměnných – liší se podle výrobce **`make`** (a příp. verze).

Ukázka použití maker

```
CFLAGS=-O2 -lm
INCL=defs.h consts.h
…
.c.o:
     $(CC) -c $(CFLAGS) $(INCL) $<
```

- Přesnou podobu maker a automatických proměnných je nutné dohledat v dokumentaci použité verze **`make`**.
- Zásadní rozdíly panují zejména mezi **GNU `make`** a **Microsoft `nmake`** nebo třeba **Watcom `wmake`**.

- Některé verze **`make`** se spouští i jiným příkazem, např. v případě **MinGW**: **`mingw32-make`**

Příkazy v `makefile`
Volání příkazů operačního systému

```
clean:
      @echo Cleaning...
      @echo Y | del *.*

install:
      @echo Installing
      @copy myapp.exe $(TARGETDIR)
```

Příkaz uvedený znakem ‘@’ se provede, ale nevypíše do konzole.

Lze použít libovolný příkaz OS, ale vede to pochopitelně k zanesení závislosti na konkrétním OS do procesu překladu...

Budoucnost řízeného překladu
Nástupci nástroje **make**

- Pro rozsáhlé projekty je klasický **make** už **nepřehledný**, `makefile` může mít tisíce řádek.
- Většina moderních nástrojů (např. Qt Creator) stále **make** využívá, ale `makefile` nepíše programátor – generuje jej vývojové prostředí z přehlednějšího a jednoduššího zápisu (např. projektové soubory Qt).

- Moderní nástroj pro řízený překlad projektů v C/C++ (obdoba ANTu/Mavenu) je

CMake ⋯▸ https://cmake.org

- Rozsáhlá dokumentace, mnohem širší možnosti a schopnosti řízení překladu než make, ale také výrazně komplikovanější použití.