Introduktion till `make`

Innehållsförteckning

Inledning

När man utvecklar programsystem bestående av flera moduler, med källkodsfiler, headerfiler, objektkodsfiler och rutinbibliotek så börjar det bli besvärligt att kompilera och länka. Det besvärliga består av två saker: dels blir kommandon långa och jobbiga att skriva, dels måste man komma ihåg vilka moduler som måste kompileras om när man har gjort ändringar i källkods- eller headerfiler.

Man kan få hjälp med båda dessa saker genom att använda programmet `make`. För att avända `make` skall man först ha skapat en fil med namnet `makefile` (eller `Makefile`, eller ett annat filnamn som i så fall anges på kommandoraden med flaggan `-f`), på denna fil anger man för varje fil som skall framställas t.ex. genom kompilering eller länkning dels från vilka filer framställningen skall ske, dels med vilket kommando det skall ske. Om man t.ex. har ett program `prog.c` som man vill kompilera och länka ihop med en modul `modul.o` som ligger i mappen (säg) `/usr/local/bib/c/58an/modules` så kan `makefile` se ut så här:

```
prog: prog.c
      gcc -g -Wall prog.c /usr/local/bib/c/58an/modules/modul.o -o prog
```

På första raden anger man normalt namnet på filen som skall framställas (målfilen, *target*) med ett kolon efter och en blankseparerad lista av namn på filer som behövs för framställningen (som målfilen är beroende utav, *dependencies*). Andra raden skall börja med ett <TAB>-tecken och innehålla därefter det kommando som skall utföras för att framställa målfilen.
Har man gjort en sådan fil så kompilerar och länkar man programmet med kommandot
`make prog`

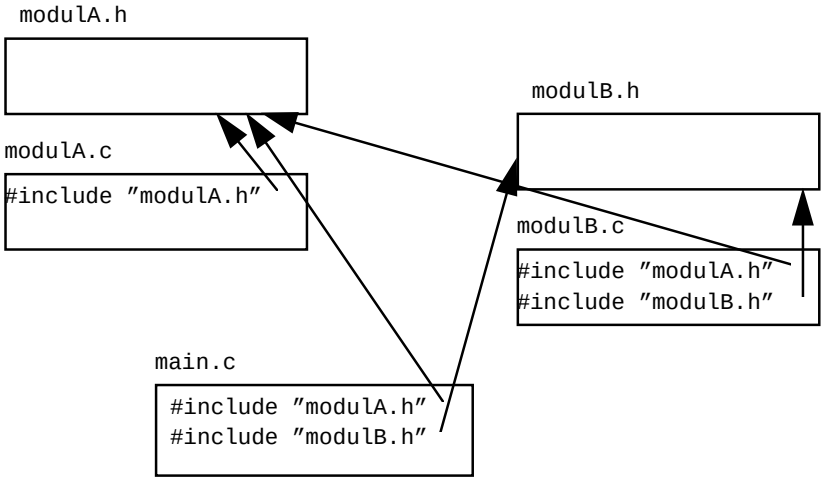
`make` letar upp målet `prog` i `makefile` och kontrollerar om målfilen `prog` finns i aktuell katalog och om den i så fall är äldre än de filer den är beroende utav (i exemplet är den endast beroende av `prog.c`). Om `prog` inte finns eller är äldre än `prog.c` så utför `make` kommandot. Om `prog` finns och är yngre än `prog.c` så anser `make` att det inte behövs någon ny kompilering eller länkning och skriver ut meddelandet[*]:
`` `prog´ is up to date. ``

Med hjälp av `make` slipper man alltså skriva det långa och komplicerade kommandot efter att ha matat in det på `makefile`.

[*] Man kan lura `make` genom att ge Unixkommandot `touch prog.c`. Kommandot `touch` ändrar filens senaste ändringsdatum till nu. Om filen inte finns så skapas en tom fil med detta namn vilket gör att detta kommando utgör ett lätt sätt att skapa en tom fil.

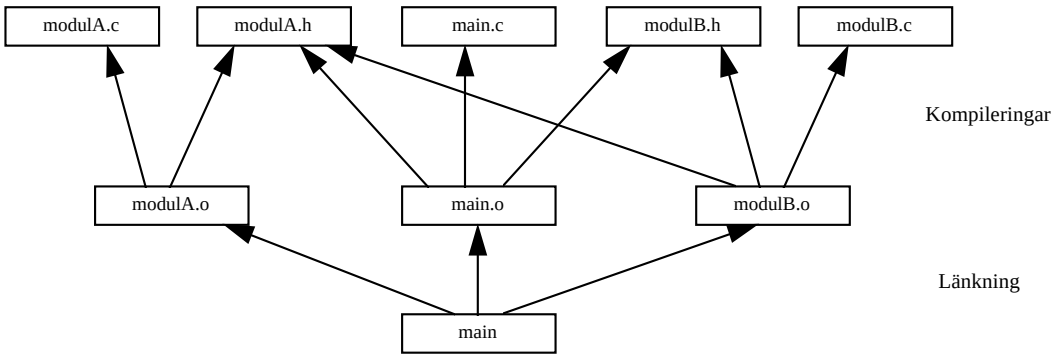
Exemplet ovan är lite väl enkelt: `prog` är beroende av endast en fil, vi har slagit ihop kompilering och länkning till ett enda steg o.s.v..

För att illustrera hur `make` kan användas med system bestående av flera moduler får vi titta på ett större exempel. Antag att vi skall bygga (kompilera och länka) ett system bestående av källkods- och header-filer enligt följande figur:



Systemet består av två moduler och ett huvudprogram. Varje modul består av två filer: en header-fil och en källkodsfil. Pilarna i figuren visar filberoenden: varje modul är beroende av sin headerfil, `modulB` är dessutom beroende av `modulA`:s header-fil. Huvudprogrammet är beroende av båda modulers header-filer.
Framställning av ett exekverbart program från dessa filer består av flera steg: man kompilerar först varje modul för sig och framställer objektfilerna `modulA.o` och `modulB.o`, man kan även separatkompilera huvudprogrammet och framställa `main.o` innan länkningen. Därefter länkar man ihop `modulA.o`, `modulB.o` och `main.o` till ett exekverbart program med namnet (säg) `main`.

I figuren nedan visas vilka filer som behövs vid framställning av de olika objektmodulerna resp. vid framställning av det exekverbara programmet:



En `makefile` för detta system skulle se ut så här:

```
main: main.o modulA.o modulB.o
      gcc -g main.o modulA.o modulB.o -o main
main.o: main.c modulA.h modulB.h
      gcc -c -g -Wall main.c
modulA.o: modulA.c modulA.h
      gcc -c -g -Wall modulA.c
modulB.o: modulB.c modulA.h modulB.h
      gcc -c -g -Wall modulB.c
```

Om alla källkods- och header-filer finns inmatade så kan nu det exekverbara programmet `main` framställas med ett enda kommando:
`make main`

`make` tittar i `makefile` och ser att `main` skall framställas av `main.o`, `modulA.o` och `modulB.o`. `make` söker då upp dessa i tur och ordning och kollar om de är "up to date", dvs. om de finns och i så fall om de är yngre än de filer de i sin tur är beroende utav. Om de inte finns eller är äldre än någon av dessa filer så framställer `make` dem genom att utföra motsvarande kommando. Därefter kommer `make` tillbaka till `main` och utför kommandot för framställning av det.

`make` skriver ut de kommandon den utför på skärmen.
Om något mål inte kan framställas (t.ex. därför att det har blivit kompileringsfel vid kompilering) så avbryter `make` arbetet.

Har man framställt `main`, testkört det och upptäckt ett fel i t.ex.modulB så ändrar man i `modulB.c`. Därefter ger man kommandot `make main` igen. `make` kommer att göra som förut: den ser att `main` är beroende av `main.o`, `modulA.o` och `modulB.o`. Den tittar då på `main.o` och ser att denna är beroende av `main.c`, `modulA.h` och `modulB.h`. Ingen av dessa har förändrats (de är alla äldre än `main.o`) så den kompileringen görs inte. Samma gäller `modulA.o`: den är yngre än de filer den är beroende utav och alltså "up to date". Men `modulB.o` är äldre än `modulB.c` (som vi just har ändrat) och alltså "out of date". `make` utför därför kompileringskommandot för `modulB.o`. Därefter kommer `make` tillbaka till `main`. Men nu har `modulB.o` blivit yngre än `main` (den har ju just framställts genom kompilering) så `make` utför länkningskommandot för att framställa `main` på nytt.

Genom att använda `make` på detta sätt behöver vi alltså inte komma ihåg vilka omkompileringar som måste göras efter ändringar av källkods- eller headerfiler, `make` gör vad som behöver göras med ledning av `makefile`.

Samma `makefile` kan användas för att framställa olika filer vid olika tilfällen. Argumentet till `make`-kommandot är det målet man vill framställa, vill man endast göra en omkompilering av `modulB.c` i exemplet ovan kan man ge kommandot `make modulB.o`. `make` söker då upp målet `modulB.o`, undersöker vilka filer det är beroende utav osv., men struntar i allt som inte behövs för att framställa `modulB.o`. Om man ger kommandot `make` utan argument så gör `make` det första målet i `makefile` (och vad som behövs för att framställa detta mål).

Raddelning, kommentarer, macros och inkludering

Om en rad i `makefile` är så lång att man vill fortsätta den på nästa rad så skall man skriva teckent `'\'` (backslash) sist på raden. Om man vill skriva kommentarer i en `makefile` så skall de påbörjas med `'#'` och sträcker sig då över resten av raden:

```
#Denna makefile illustrerar långa rader och kommentarer
prog: prog.c /usr/local/bib/c/58an/modules/module.h /usr/local/bib\
/c/58an/l2/l2lists.h     #Detta är en fortsättning av föregående rad
      gcc -g -c -Wall prog.c
```

Man kan även definiera sk. *macros* i en makefile. Ett macro är helt enkelt ett namn på en sträng, efter definitionen av macrot kan man i `makefile` använda namnet och `make` kommer att ersätta detta namn med strängen. Macrodefinitionen ser så här:

`namn = sträng`

där `sträng` sträcker sig över resten av raden eller fram till ett kommentarstecken om det finns på raden. När man sedan skall använda namnet så skall det omges av parenteser och föregås av ett `$` (dollartecken):

`$(namn)`

Det är brukligt att macronamn stavas med versaler (`make` skiljer på versaler och gemena), så att risken för namnkollisioner minskar.

Exempel: antag att programmet `prog` ska kompileras och länkas ihop med en modul `modul.o` som ligger på katalogen `/usr/local/bib/c/58an/modules` och att denna moduls header fil modul.h ligger på katalogen `/usr/local/bib/c/58an/include`. Makefilen skulle kunna se ut så här:

```
BIB=/usr/local/bib/c/58an
MOD=$(BIB)/modules
INC=$(BIB)/include

prog: prog.o $(MOD)/modul.o
      gcc -g -o prog prog.o $(MOD)/modul.o
prog.o: prog.c $(INC)/modul.h
      gcc -c -g -Wall -I$(INC) prog.c
```

Flaggan `-I` till kompileringskommandot anger var preprocessorn skall söka headerfiler som inkluderas i den kompilerade källkoden.

Vid mer professionell programutveckling i Unix-miljön är det brukligt att definiera macros för kompileringskommandon, kompileringsflaggor, länkningsflaggor och grupper av filer - t.ex. alla källkoder och alla objektkoder. Dels gör det möjligt att enkelt ändra och lägga till filer och flaggor eller byta kompilator, dessutom kan man då utnyttja viss automatik i vissa situationer (som faller utanför ramen för detta häfte).

Exempel på en sådan "proffsig" `makefile` (se nästa sida):

```
# Makefil för exempelsystemet av Jozef Swiatycki

CC= gcc               # Kompilerings- och länkningskommandot
CFLAGS= -g -c -Wall   # Flaggor för kompilering
LDFLAGS= -g           # Flaggor för länkning
OBJECTS= main.o modulA.o modulB.o  # Objektkodsfilerna

main: $(OBJECTS)
      $(CC) $(LDFLAGS) $(OBJECTS) -o main
modulB.o: modulB.c modulB.h modulA.h
      $(CC) $(CFLAGS) modulB.c
modulA.o: modulA.c modulA.h
      $(CC) $(CFLAGS) modulA.c
main.o: main.c modulA.h modulB.h
      $(CC) $(CFLAGS) main.c
```

Macros kan även definieras från kommandoraden. Detta brukar t.ex. användas om man vill förbereda makefilen att vid olika tillfällen använda filer från olika kataloger. Antag t.ex. att modulen `modul.o` finns kompilerad för olika operativsystem, t.ex. Linux, Windows och Solaris. Ett vanligt sätt att göra är att lägga objektkoderna för de olika versionerna av modulen i var sin mapp, t.ex. `/usr/local/bib/c/58an/modules/linux/modul.o`, `/usr/local/bib/c/58an/modules/windows/modul.o` o.s.v. Säg dessutom att headerfilen `modul.h` ligger på `/usr/local/bib/c/58an/include` (samma för alla system).

Följande skulle kunna vara en makefile för ett program som använder sig av modulen:

```
BIB=/usr/local/bib/c/58an
MOD=$(BIB)/modules
INC=$(BIB)/include

prog: prog.o $(MOD)/$(SYSTEM)/modul.o
      gcc -g prog.o $(MOD)/$(SYSTEM)/modul.o -o prog
prog.o: prog.c $(INC)/module.h
      gcc -c -g -I$(INC) prog.c
```

Macrot `SYSTEM` är inte definierat i makefilen utan skall definieras på kommandoraden. Genom att definiera macrot `SYSTEM` som `linux`, `windows` eller `solaris` beroende på vilket system vi vill bygga programmet på kan samma `makefile` användas. Kommandoexempel (på en Linux-maskin):

```
make prog SYSTEM=linux
```

Detta kan göras ännu bättre genom att definiera macrot `SYSTEM=linux` i makefilen, macros på kommandoraden omdefinierar nämligen eventuella macros i makefilen. På det sättet behöver man inte ange något speciellt om man bygger systemet på en Linux-maskin, men kan använda makefilen och ange `SYSTEM=windows` om man vill bygga det för MS Windows och motsvarande för Solaris.

Om man har ett macro betecknande en viss sträng så kan man ur denna sträng framställa en annan sträng genom att byta ut delar av den ursprungliga strängen - det finns många funktioner för strängmanipulering i makefiler. Bl.a. finns möjligheten att ersätta ändelsen i varje ord som förekommer i macro-strängen med en annan ändelse. Syntaxen är följande `$(namn:`*`old`*`=`*`new`*`)` med betydelsen att här läggs strängen från macrot `namn` in, men ändelsen *`old`* i varje ord ersätts med ändelsen *`new`*.

Om vi t.ex. har ett macro med en lista av källkodsfiler:

`SOURCES=modulA.c modulB.c main.c`
så kan vi skapa ett macro med en lista av motsvarande objektkodsfiler genom att skriva:
`OBJECTS=$(SOURCES:.c=.o)`
Värdet av macrot `OBJECTS` blir alltså `modulA.o modulB.o main.o`

Vi kommer att behöva denna möjlighet senare.

Med direktiven `include` *`filnamn`* eller `-include` *`filnamn`* i makefilen kan man få make att läsa från en annan fil (som alltså ska innehålla samma konstruktioner som en makefil: macrodefinitioner, filberoenden, kommandon...). Meningen med detta är att kunna dela upp makefilen för stora projekt i flera filer eller att kunna ha macrodefinitioner som ska gälla för flera projekt i en global fil som inkluderas av flera makefiler.

Dessa direktiv måste börja först på sin rad, ordet `include` måste efterföljas av ett blanktecken.

Skillnaden mellan `include`-direktivet och `-include`-direktivet är att varianten utan bindestreck ger ett felmeddelande och avbryter make-processen om den angivna filen inte kan öppnas, medan varianten med bindestreck ignorerar inkluderingen om angiven fil inte hittas men fortsätter. Vi kommer att se användning av detta direktiv i avsnittet om automatisk generering av filberoenden.

Undertryckning av utskrifter från `make`

Ibland vill man undertrycka utskrifter från `make`. Detta är framför allt fallet då `make`-körningen utgör en del av en större programgenereringsprocess, där man inte tror att användaren är intresserad av en massa utskrifter på skärmen som hon inte förstår.

Undertryckning av utskrifter från `make` kan göras på tre sätt:
– undertryckning av utskrift av enstaka kommandon görs genom att man skriver ett @ (snabel-a) framför kommandot:

```
main: main.o modulA.o modulB.o
      @gcc -g main.o modulA.o modulB.o –o main
```

– undertryckning av alla utskrifter kan göras genom att man skriver in det speciella målet `.SILENT:` någonstans i makefilen
– undertryckning av alla utskrifter kan även göras vid kommandot `make` med flaggan `-s`:
`make -s main`

Om man har begärt undertryckning av utskrifter kan man ändå skriva ut meddelanden till användaren genom att använda Unix-kommandot `echo`, som skriver ut sitt argument på terminalen:
`echo Nu är det klart!`

Se exemplet i nästa avsnitt.

Flera kommandon per mål

Man kan ge fler än ett kommando för ett visst mål, dessa kommandon kan dessutom vara godtyckliga kommandon. Man kan t.ex. vilja ändra filskydd för den framställda målfilen, kopiera den framställda filen till någon annan katalog eller skapa subkataloger där de olika filerna kommer att läggas. Det viktiga att komma ihåg är att kommandorader skall börja med ett <TAB>-tecken.

Dessutom behöver inte målen i en `makefile` vara filer som skall framställas. Man kan hitta på ett målnamn bara för att få de kommandon som man anger på efterföljande rader utförda genom att man ger motsvarande `make`-kommando. T.ex. kan man vilja ha kommandon som tar bort objektfilerna efter att det exekverbara programmet är framställt samlade under namnet `clean`, som

i exemplet nedan. För att få dessa kommandon utförda ger man kommandot `make clean`. Att man kan vilja lägga detta i makefilen beror på att man vill samla alla kommandon som har att göra med framställningen av `main` i en och samma fil.
Listan av filer som behövs för målet `clean` blir givetvis tom.
Det finns dock en situation då detta inte fungerar, nämligen om det råkar finnas en fil med namnet `clean` på den aktuella katalogen. För att se till att `clean` och andra sådana ”låtsasmål” fungerar även i denna situation kan man räkna upp dessa ”låtsasmål” efter specialmålet `.PHONY:`

Bland annat kan kommandon för ett mål utgöras av `make`-kommandot (rekursivt)!

Exempel:

```
PROJ=~kalle/projektet           # Katalogen där vi vill ha main

.SILENT:
main: main.o modulA.o modulB.o
      gcc -g main.o modulA.o modulB.o -o main
      chmod 770 main
      mkdir $(PROJ)
      cp main $(PROJ)
      make clean
      echo Nu är det klart!
main.o: main.c modulA.h modulB.h
      gcc -c -g main.c
      echo Klar med kompilering av main.c
modulB.o: modulB.c modulB.h modulA.h
      gcc -c -g modulB.c
      echo Klar med kompilering av modulB.c
modulA.o: modulA.c modulA.h
      gcc -c -g modulA.c
      echo Klar med kompilering av modulA.c
.PHONY: clean
clean:
      rm main.o
      rm modulA.o
      rm modulB.o
      echo Klar med städning av objektfilerna
```

Obs dock att det inte är så smart att anropa målet `clean` från main-målet: alla objektkodsfiler tas nu bort vid varje länkning och en del av vitsen med make går förlorad eftersom alla moduler kommer att kompileras om vid varje förändring.

Generella mål och automatisk generering av beroenden

Om man vet att framställning av en viss typ av filer från en annan typ av filer alltid görs med samma kommando kan man skapa en generell regel (*inference rule*) för detta. Om vi t.ex. vet att vi alltid vill framställa objektkoder (vilkas filnamn slutar med `.o`) från C-källkoder (vilkas namn slutar med `.c`) med kommandot
`gcc -c -g` *`källkodsfilens namn`* så kan det uttryckas en gång för alla i en makefil med hjälp av målet `.c.o:` vilket står för "hur man ur en `.c`-fil framställer en `.o`-fil". Vi måste dock kunna uttrycka *källkodsfilens namn* i `gcc`-kommandot. `make` har för sådana situationer några inbyggda macros, av vilka vi här kan nämna följande:

- `$<` står för namnet på den fil som har utlöst en generell regel, alltså just källkodsfilens namn
- `$*` står för filnamnet utan suffix (i vårt fall utan `.c`)
- `$@` står för målets fulla namn, kan endast användas i specifika regler
- `$?` står för de filer som utlöste en specifik regel

För vårt exempel skulle vi t.ex. kunna skapa följande makefil:

```
main: main.o modulA.o modulB.o
      gcc -g  $? -o $@
.c.o:
      gcc -c -g  $<
```

När vi nu ger kommandot `make main` upptäcker `make` att `main` är beroende av `main.o` . `make` hittar inte `main.o` i katalogen, men väl `main.c`. Nu hittar `make` den generella regeln för framställning av `.o`-filer från `.c`-filer, så den kör kommandot
`gcc -c -g  main.c`
där macrot `$<` i den generella regeln har ersatts med den filen som utlöste regeln, nämligen just `main.c`
Detta upprepas för `modulA.o` och `modulB.o`. Därefter kör `make` kommandot
`gcc -g  main.o modulA.o modulB.o -o main`
där `$?` har ersatts med de filer som utlöste regeln, medan `$@` har ersatts med målets namn.

Denna makefil har flera grova svagheter. För det första ersätts macrot $? med namnen på endast de filer som behövde göras om. Om t.ex. `modulB.o` redan finns och är "up to date" så kommer den inte att behöva göras om och kommer inte med i länkningskommandot. Detta är givetvis fel, macrot `$?` kan endast användas på detta sätt då man med säkerhet vet att ingen av filerna finns kompilerad på katalogen.
För det andra så har vi inte dokumenterat att ändringar i headerfilerna också skall utlösa den generella regeln (vi har inte angivt filberoenden). Detta kan vi lösa genom att lägga in rader med filberoenden men utan kommandon - kommandot kan fortfarande tas från den generella regeln:

```
main: main.o modulA.o modulB.o
      gcc -g  main.o modulA.o modulB.o -o $@
.c.o:
      gcc -c -g  $<
main.o: main.c modulA.h modulB.h
modulB.o: modulB.c modulB.h modulA.h
modulA.o: modulA.c modulA.h
```

Dessa rader med filberoenden kan genereras automatiskt på lite olika sätt. Det rekommenderade sättet om man jobbar med GCC är att låta kompilatorn framställa filer med dessa beroenden och inkludera dessa filer med beroenden i sin makefil. I GCC får man kompilatorn att skapa beroende-rader genom att ge flaggan `-MD` eller `-MMD` vid kompileringen. Dessa flaggor gör att GCC förutom

att framställa objektkoden på en fil med filtypen `.o` även framställer en liten fil med filtypen `.d` med en rad med källkodsfilens filberoenden. Skillanden mellan `-MD` och `-MMD` är att den första skriver ut alla beroenderader medan den andra skriver ut bara egna beroenden (alltså inte beroenden av standardheader-filer, som ju inte förändras).

Om vi t.ex. ger kommandot
```
gcc -c -g -Wall -MMD main.c
```

så kommer GCC dels att framställa filen `main.o`, dels filen `main.d` med följande utseende:

```
main.o: main.c modulA.h modulB.h
```

Om man ger denna flagga till kompileringskommandot i den generella regeln i makefilen så kommer GCC att framställa en sådan `.d`-fil för varje källkodsfil den kompilerar, dessa `.d`-filer kan vi då inkludera i makefilen:

```
# Makefile för exempelsystemet
CC= gcc
CFLAGS= -g -c -MMD
LDFLAGS= -g
SOURCES= main.c modulA.c modulB.c
OBJECTS= $(SOURCES:.c=.o)

main: $(OBJECTS)
     $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.c.o:
     $(CC) $(CFLAGS) $<

-include $(SOURCES:.c=.d)

.PHONY: clean
clean:
     rm *.o *.d
```

Eftersom vi har flaggan `-MMD` bland kompileringsflaggor kommer kompileringskommandot att framställa en `.d`-fil med filberoenden för varje källkodsfil som kompileras. Dessa `.d`-filer inkluderar vi med ett `-include`-direktiv - obs att det måste vara bindestrecksvarianten av detta direktiv eftersom `.d`-filer inte finns vid första kompileringen.

Några ord till användare av MinGW

I den normala distributionen av MinGW ingår programmet make under namnet `mingw32-make` (har man även installerat terminal- och `bash`-emulatorn `MSYS` så finns där en annan version av make med det vanliga namnet). Man kan använda programmet enligt beskrivningen i detta häfte men under detta långa namn eller döpa om filen `mingw32-make.exe` till `make.exe` (programmet ligger i mappen `MinGW\bin\`).
Ett litet problem är att exekverbara filer under Windows har filtypen `.exe`, vilket gör att det slutliga målet måste ha denna typ för att kännas igen, t.ex. i exemplet ovan:
`main.exe: $(OBJECTS)`
Detta kommer dock inte att fungera om makefilen flyttas till Linux. Man kan då ändra detta i makefilen vid flyttningen till Linux eller använda ett macro:
```
SUFFIX=.exe
main$(SUFFIX): $(OBJECTS)
```
som man får definiera om vid flytten till Linux (i makefilen eller från kommandoraden).

Flaggor till make-kommandot

Man kan ändra `make`s beteende genom att ange flaggor till `make`-kommandot. Följande är de mest användbara:

`-f` *`filnamn`* betyder att `make` skall använda filen med det givna namnet istället för en fil med namnet `makefile` (eller `Makefile`)

`-k` betyder att om `make` upptäcker fel vid framställning av ett av målen så skall den ändå försöka framställa andra mål istället för att avbryta jobbet helt

`-n` betyder att `make` inte skall utföra kommandon utan bara skriva ut dem på terminalen, kan användas för att kontrollera att `makefile` är rätt skriven

`-s` silent, betyder att `make` inte skall skriva ut utförda kommandon på skärmen

`namn``=`*`text`* macrot med namnet *`namn`* definieras och ges värdet *`text`*

Läsa mer om `make` och liknande verktyg

På WWW, t.ex. `http://www.gnu.org/software/make/manual/make.html` finns fullständiga manualer för GNU `make` i olika format, till ett av dessa finns en länk från kursens webbsidor.

Kommandot `man make` skriver ut manual-sidorna med information om `make` på skärmen.

I bokform finns ”*Managing projects with* GNU `make`”, 3dje uppl. av R. Mecklenburg, O’Reilly & Associates 2005.

Det finns andra liknande verktyg, mest kända är:

- `jam`, se `http://www.perforce.com/jam/jam.html`
- `nmake`, `se http://www.bell-labs.com/project/nmake/` för Lucent-variant eller `http://msdn.microsoft.com/library/en-us/vcug98/html/_asug_overview.3a_.nmake_reference.asp` för Microsofts variant

Det finns ingeting som hindrar att man använder `make` för att bygga Java-program, men Java-utvecklare verkar föredra ett verktyg med liknande funktionalitet gjort speciellt för Java. Verktyget heter `Ant`, kommer från Apache Software Foundation (`http://www.apche.org`), är skrivet i Java och använder XML som notation för sin motsvarighet till makefiler. Information om `Ant` kan man hitta på `http://ant.apache.org/manual/index.html`.