make

make är ett verktyg som främst används för att underhålla, uppdatera och återskapa program och filer. Det är dock ett generellt verktyg som kan användas även i många andra sammanhang. En avancerad sida av make är att make har en hel del inbyggd kunskap om hur kommandon ska ges för att översätta program skrivna i olika programspråk till körbara program. Normal skriver man dock s.k. *make-filer*, i vilka man anger hur **make** ska utföra olika slags bearbetningar.

make-filer

En make-fil är en textfil som **make** läser. Då **make** används för att generera ett körbart program från källkod, innehåller make-filen typiskt kompileringskommandon för att skapa objektkodsmoduler och länkkommando för att sätta samman objektkodsmodulerna det körbara programmet. Make-filen kan även innehålla kommandon för hur programmet och eventuell tillhörande dokumentation ska installeras, liksom kommandon för att städa upp efter översättning och installation.

Det finns två standardnamn för make-filer, makefile och Makefile. En make-fil kan ha ett godtyckligt namn, men det underlättar att använder dessa standardnamn.

Köra make

make körs i det enklaste fallet med följande kommando i ett terminalfönster:

```
% make
```

make söker då i arbetsmappen efter i första hand `makefile`, och om ingen sådan fil finns, i andra hand efter `Makefile`. Finns ingen av dessa filer avslutas **make**. Man kan köra **make** och med växeln `-f` för att uttryckligen ange namnet på den make-fil som **make** ska använda:

```
% make -f makefil
```

I detta fall kan make-filen ha ett godtyckligt namn. Det finns många fler växlar som kan ges till **make**, se manualsidan för make(1).

Konstruktion av make-filer

En make-fil kan vara mycket avancerad och **make** har en hel del inbyggd kunskap, till exempel hur källkodsfiler för ett programspråk ska kompileras och hur körbara program ska genereras. Här finns endast utrymme för att översiktligt beskriva hur enkla make-filer kan konstrueras. Några avancerade möjligheter beskrivs dock kort avslutningsvis.

Det finns tre grundläggande begrepp då det gäller make-filer: *mål* (target), *beroende* (dependency) och *regel* (rule). Dessutom är *makron* (*macros*) något som är mycket användbart och det finns en stor mängd fördefinierade makron i **make**. Dessa fyra begrepp beskrivs översiktligt nedan.

Mål

Ett *mål* (*target*) är ofta en fil (*module*) som ska konstrueras av **make**. Det kan vara en objektkodsmodul, som då är ett delmål (*sekundärt* mål) eller ett körbart program som är slutmålet. Om man har ett program som är uppdelat på flera källkodsfiler, av vilka en fil innehåller huvudprogrammet, kan delmålen vara att kompilera varje enskild källkodsfil till objektkod, och slutmålet att länka dess objektkodsmoduler till det körbara programmet.

Vissa mål kan innebära att en följd av kommandon utförs utan att någon modul eller annat skapas. Ett exempel på ett sådant mål kan vara att radera alla objektkodsmoduler som skapats vid kompilering av ett program, efter det att programmet genererats och installerats.

I en make-fil anges ett mål först på en rad och efter målet skrivs ett kolon. Om det finns beroenden anges dessa efter kolonet, annars ingenting ytterligare på målraden. Exempel ges nedan.

Beroende

Ett *beroende* (*dependency*) beskriver vilken eller vilka andra moduler eller delmål som ett mål är beroende av. En objektkodsfil är till exempel beroende av den motsvarande källkodsfilen. Om ett mål är att skapa objektkodsfilen för en källkodsfil main.cc, ser målraden ut enligt följande:

```
main.o: main.cc
```

Målet är objektkodsfilen `main.o` och denna beror av källkodsfilen `main.cc`.

Regler

En *regel* (*rule*) är ett eller flera kommandon som ska utföras för att skapa ett mål. För det exempel som påbörjades ovan kan den tillhörande regeln vara:

```
main.o: main.cc
        g++ -c main.cc
```

För att generera målet, objektkodsfilen main.o, ska således källkodsfilen main.cc kompileras med kommandot ”g++ -c main.cc” (regeln). Väljaren -c anger att kompileringen ska avbrytas innan länkningen, vilket medför att objektkod skrivs på filen main.o.

Om flera kommandon behöver utföras för att skapa ett mål, skrivs varje kommando på en egen rad.

Observera, varje regelrad måste inledas med ett tabulertecken!

Makron

Ett *makro* är en variabel som kan användas i beroenden och i regler. Makron definieras på formen:

```
MAKRONAMN = sträng
```

Ett *makronamn* skrivs typiskt med versaler. *Strängen* som makronamnet representerar är en godtycklig sträng, som kan innehålla även blanktecken och som avslutas då den rad ett makro skrivs på avslutas eller då ett kommentartecken (#) skrivs. Exempel på två enkelt makron, avslutade med kommentar:

```
    CCC = g++               # C++-kompilator (GCC)
CCFLAGS = -g -std=c++11     # Kompilera för avlusning och C++11
```

Då ett makronamn används skrivs det inom parenteser och med ett inledande dollartecken. I det exempel som använts ovan skulle ovan definierade makron kunna användas enligt följande:

```
main.o: main.cc
        $(CCC) $(CCFLAGS) -c main.cc
```

Makron definieras vanligtvis inledningsvis i make-filer och används i flera syften. Makron kan användas för att parametrisera innehållet i en make-fil och därigenom förenkla ändringar man kan vilja göra. Om man till exempel vill använda `CC` i stället för `g++`, ändrar man `g++` till `CC` i makrot `CCC`. Om man *ej* önskar kompilera för avlusning, tar man bara bort väljaren `-g` på den rad makrot `CCFLAGS` definieras.

En annan användning av makron är att ge enkla namn på kanske långa strängar man behöver skriva. Om man till exempel har många objektkodsmoduler man vill ange, kan man definiera ett makro i stil med:

```
OBJECTS = qsort.o bsearch.o hash.o interface.o main.o
```

Då objektkodsmodulerna ska refereras i något sammanhang, behöver man endast ange $(OBJECTS).

Ett mer komplett exempel

För det exempel som använts ovan behövs ytterligare ett mål för att skapa det körbara programmet `main` från objektkodsmodulen `main.o`. Dessutom kan man vilja ha ett mål för att städa upp efter att programmet `main` genererats och installerats på annan plats (det är i och för inte så realistiskt...).

En enkel make-fil för att göra detta kan se ut enligt följande (med LDFLAGS för att fixa länkproblem på IDA:s solarissystem).

```
#
# Makefile för att kompilera programmet main med g++
#
    CCC = g++
CCFLAGS = -g -std=c++11 -pedantic -Wall -Wextra -Werror
LDFLAGS = -L/sw/gcc-${GCC4_V}/lib -static-libstdc++

main: main.o
        $(CCC) $(CCFLAGS) $(LDFLAGS) main.o -o main

main.o: main.cc
        $(CCC) $(CCFLAGS) -c main.cc

# Städa arbetsmappen

clean:
        @ \rm -f *.o
```

Om man ger kommandot **make** kommer, under förutsättning att objektkodsmodulen main.o ej existerar och att den den ej är föråldrad, först målet main.o att utföras, dvs motsvarande kommandot

```
g++ -g -std=c++11 -pedantic -Wall -Wextra -Werror -c main.cc
```

och sedan målet main, där både flaggorna i CCCFLAGS och LDFLAGS ingår.

Om man endast vill utföra målet `main.o` kan man göra det med kommandot:

```
% make main.o
```

Om man därefter, utan att ändra i källkodsfilen `main.cc`, ger kommandot **make** kommer endast huvudmålet att behöva utföras; **make** noterar i detta fall att en aktuell objektkodsmodul redan finns. Detta är en klar fördel med **make**, endast de filer som behöver kompileras om, kompileras om.

Målet `clean` städar bort objektkodsfilerna och används enligt:

```
% make clean
```

Tecknet @ medför att kommandot ej skrivs ut. Tecknet \ före kommandot rm innebär att ett eventuellt alias som införts för rm (till exempel rm -i) ej ska användas, utan rm-kommandot i dess egentliga definition (vilket är att radera angivna filer utan begära bekräftelse på att så ska ske).

Anm. Om man har ett program på en enda källkodsfil, till exempel main.cc, kan **make** kompilera ett sådant program, *utan* att en make-fil alls existerar, om man ger kommandot:

```
% make main
```

Detta fungerar därför att make har fördefinierade makron, se nedan, för vanliga filnamnssuffix, till exempel .cc, och olika kompilatorer, till exempel g++, samt kompileringskommandon som ska användas i olika fall. Argumentet main till **make** medför att **make** letar efter en fil med namnet main. Då en sådan fil hittas, main.cc, bestämmer suffixet .cc, att ett förutbestämt kompileringskommando ska utföras.

Flera källkodsfiler

Antag att vi har ett program som består av källkodsfilerna main.cc, a.cc, b.cc och c.cc, med tillhörande inkluderingsfiler. En enkel make-fil för att kompilera det programmet kan vara:

```
     CCC = g++
CCFLAGS = -g -std=c++11 -pedantic -Wall -Wextra
LDFLAGS = -L/sw/gcc-${GCC4_V}/lib -static-libstdc++

main: main.o a.o b.o c.o
        $(CCC) $(CCFLAGS) $(LDFLAGS) main.o a.o b.o c.o -o main

main.o: main.cc
        $(CCC) $(CCFLAGS) -c main.cc

a.o: a.cc
        $(CCC) $(CCFLAGS) -c a.cc

b.o: b.cc
        $(CCC) $(CCFLAGS) -c b.cc

c.o: c.cc
        $(CCC) $(CCFLAGS) -c c.cc
```

Varje implementeringsfil har ett eget mål för att kompilera källkoden till objektkod. Huvudmålet main är ett i grunden länkkommando för att sätta ihop objektkodsmodulerna till det körbara programmet.

Fördefinierade makron

För ett flertal programspråk och en del verktyg finns föredefinierade makron som anger hur olika saker ska utföras. För C++ finns fördefinierade makron som bestämmer kommandona för kompilering och länkning:

`CCC` Kompilator (förvalt värde är `CC`).

`C++C` Alternativ till `CCC`, se ovan.

`CCFLAGS` Väljare till kompilerigskommandot (inga förvalda).

`C++FLAGS` Alternativ till `CCFLAGS` (förvalt värde `-O`, för optimering).

`CCPFLAGS` Väljare till preprocessorn (inga förvalda).

`LDFLAGS` Väljare till länkaren, `ld` (inga förvalda).

`COMPILE.cc = $(CCC) $(CCFLAGS) $(CPPFLAGS) -c`
Kommandoraden för kompilering, sammansatt av värdena för `CCC`, `CCFLAGS`, `CPPFLAGS`, samt väljaren `-c`.

`COMPILE.C = $(C++C) $(C++FLAGS) $(CPPFLAGS) -c`
Alternativ till `COMPILE.cc`, se ovan.

`LINK.cc = $(CCC) $(CCFLAGS) $(CPPFLAGS) $(LDFLAGS)`
Kommandot för länkning, sammansatt av värdena för `CCC`, `CCFLAGS`, `CPPFLAGS` och `LDFLAGS`.

`LINK.C = $(C++C) $(C++FLAGS) $(CPPFLAGS) $(LDFLAGS) -target`
Alternativ till `LINK.cc`, se ovan.

Fördefinierade makron kan användas i egna make-filer enligt följande exempel:

```
C++FLAGS += -g -std=c++11 -Wpedantic -Wall -Wextra -Werror

main: main.o a.o b.o c.o
        $(LINK.cc) main.o a.o b.o c.o -o main

main.o: main.cc
        $(COMPILE.cc) main.cc

a.o: a.cc
        $(COMPILE.cc) a.cc

b.o: b.cc
        $(COMPILE.cc) b.cc

c.o: c.cc
        $(COMPILE.cc) c.cc
```

Implicita regler

Kommandoraderna i exemplet ovan för att kompilera main.o, a.o, b.o och .c.o från motsvarande källkodsfiler är identiska med den implicita *suffixregel*n .cc.o, som anger hur källkodsfiler med suffix .cc ska kompileras till objektkod, .o. Det finns också en suffixregel .cc, som anger hur källkodsfiler ska kompileras och länkas till ett körbart program. Dessa regler använder dessutom *dynamiska makron* (t$@ och $<, se nedan):

```
.cc:
$(LINK.cc) -o $@ $<

.cc.o:
$(COMPILE.cc) $(OUTPUT_OPTION) $<
```

Dessa implicita regler innebär att delmålen i exemplet ovan, eftersom de är identiska med de implicita suffixreglerna, inte behöver anges. make-filen kan alltså skrivas:

```
C++FLAGS += -g -std=c++11 -Wpedantic -Wall -Wextra -Werror

main: main.o a.o b.o c.o
        $(LINK.cc) main.o a.o b.o c.o -o main
```

Dynamiska makron

Dynamiska makron får sina värden dynamisk, dvs då **make** utför make-filen, och det sker mål för mål. Dessa makron används i hög grad i implicita regler men är också mycket användbara för att själv konstruera regler. Här är några av de vanligaste dynamiska makrona:

$@ Namnet på det aktuella målet.

$? Listan av beroenden.

$< Namnet på beroendefilen, som valts av **make** för användning i en implicit regel.

$* Basnamnet för det aktuella målet (dvs namnet utan suffix).

I exempelvis suffixregeln .cc.o ovan, ersätts $< med namnet på den beroendefil, i detta fall en .cc.fil för det aktuella målet (till exempel main.cc om det aktuella målet valt av **make** är main.o).