

Kompilering

Kompileringsprocessen består egentligen av flera steg. Man skriver sitt program i form av källkodsfiler (.c och .h), som sedan kompileras till objektfiler .o och slutligen länkas ihop till en körbar (binär) fil.

Konceptet är inte beroende av programmeringsspråk, så mycket av det som står här gäller även för andra språk än C. Många av flaggorna som anges här är någorlunda standard, medan andra är specifika för gcc, som används i exemplen.

Utan flaggor kommer gcc att både kompilera och länka. Resultatet är ett körbart program **a.out**. Med flaggan **-o** kan man bestämma namnet på programmet:

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

När källkoden består av flera .c-filer kan man dela upp stegen, så att varje .c-fil kompileras för sig, och de resulterande .o-filerna länkas ihop.

Skapa objektfiler **funcs.o** och **program.o**:

```
$ gcc -c funcs.c
$ gcc -c program.c
$ gcc -c main.c
```

Länka ihop objektfilerna till ett körbart program **myprog**:

```
$ gcc -o myprog funcs.o main.o program.o
```

Observera att man inte behöver kompilera och länka i separata steg, utan stegen kan kombineras enligt följande, som både kompilerar och länkar:

```
$ gcc -o myprog funcs.c main.c program.c
```

Om man gjort ändringar i endast en eller några av källkodsfilerna och har kvar de övriga objektfilerna, går kompileringen fortare om man bara kompilerar om de filer som faktiskt ändrats. Man kan alltså ange en blandad lista av .c- och .o-filer så här:

```
$ gcc -o myprog funcs.o main.c program.o
```

Ibland behövs särskilda funktionsbibliotek vid kompilering, i så fall anges dessa med flaggan **-l**. Vanligast är när man använt matematiska funktioner i sin kod och behöver länka med mattembiblioteket. Flaggan **-l** anges alltid sist:

```
$ gcc -o myprog myprog.c -lm
```

-lm säger att filen **libm.a** ska länkas med resten av programmet. Observera att **gcc** endast letar efter funktionsbibliotek bara på särskilda ställen, t ex **/usr/lib**. Om du har ett funktionsbibliotek som ligger någon annanstans, måste du ange explicit för **gcc** var den ska leta. Det görs med flaggan **-L**:

```
$ gcc -o myprog myprog.c -L/stud/docs/kurs/os/lib -lm -los
```

Detta exempel visar hur man skriver när funktionsbiblioteket ligger "där du står nu":

```
$ gcc -o myprog myprog.c -L. -lm -los
```

Felsökning

Det är sällan man lyckas skriva ett helt korrekt program första gången. Det är i stället vanligt med fel, både *syntaktiska* och *logiska*.

Syntaktiska fel

Syntaktiska fel är sådana som hindrar kompilatorn från att generera ett program utifrån din källkod, pga att koden avviker i någon bemärkelse från programspråkets regler.

Kompilatorn skiljer mellan *varningar* och *fel* vid kompileringen.

Syntaktiska fel gör att kompileringen misslyckats, och dessa indikeras alltid med texten **error** följt av en (mer eller mindre) hjälpsam beskrivning av felet, samt radnumret där felet upptäcktes. Observera att kompilatorn ofta upptäcker fel först på nästa rad eller senare, och att vissa sorters fel kan ge upphov till ett stort antal *följdfel* i programkod som i själva verket är korrekt.

Varningar beror på saker som inte är direkt felaktiga, men som kompilatorn tycker är konstiga. Då kompilatorn oftast vet bäst kan det vara bra att notera även varningar och försöka få bort källan till dem. `gcc` brukar inte skriva ut varningar om inte man explicit begär det, det görs med flaggan `-Wall` och jag rekommenderar att ni använder den. T ex:

```
$ gcc -Wall -D__USE_FIXED_PROTOTYPES__ ...
```

Normalt ska inte `-D__USE_FIXED_PROTOTYPES__` behövas, men `-Wall` ger ibland en del varningar som inte beror på den egna koden, och denna flagga kan användas för att slippa dem.

Logiska fel

När man lyckats kompilera och har fått ett program som går att köra, är det inte säkert att det fungerar korrekt. Det finns lite olika metoder för att hitta logiska fel som gör att du fått ett annat resultat än det väntade.

Vanligt är att man använder en *debugger*, som tillåter att man kör programmet en eller flera rader i taget, samtidigt som man kan titta på värden som lagrats i variabler osv. För att kunna använda en debugger krävs att kompilatorn bevarar en del symbolisk information i filen vid kompilering och länkning. Detta görs med flaggan `-g`, dvs:

```
$ gcc -g -o myprog myprog.c ...
```

Programmet kan sedan köras i debuggern, som då styr dess körning. Ex:

```
$ gdb myprog
```

eller

```
$ /usr/hacks/bin/xxgdb myprog
```

Väl inne i debuggern finns det flera funktioner för att följa körningen och titta i programminnet osv. Manualsidan för debuggern ifråga kan erbjuda mer information.

En annan metod för att hitta logiska fel är felutskrifter. Man lägger in ett antal utskrifter på särskilda ställen i koden, så att man ser var programmet befinner sig och kan observera värdet på intressanta variabler.

För att det ska vara enkelt att ta bort felutskrifterna efteråt (och även lägga tillbaka dem vid behov) används en enkel teknik. Genom att omge felutskrifterna med `#ifdef...#endif`-satser kan man styra vid kompilering huruvida dessa rader ska vara med i programmet eller inte:

```
#ifdef DEBUG
    fprintf(stderr,"i is now %d, foo is %s",i,foo);
#endif
```

När man kompilerar skriver man så här för att få med utskrifterna:

```
$ gcc -Wall -DDEBUG ...
```

För att kompilera utan felutskrifterna tar man helt enkelt bort `-DDEBUG` och kompilerar om.

En mycket användbar metod för att hitta logiska fel är att använda `assert()`. `assert()` är ett makro, vars argument är ett påstående som alltid måste vara sant. `assert()` testar påståendet och ifall det visar sig vara falskt, avbryter programmet med ett meddelande.

```
#include <assert.h>

[...]

assert( i>0 && i<MAX );
```

Man använder `assert()` när man gjort ett antagande i koden (t ex att ett värde ligger inom ett visst intervall) men inte explicit testat antagandet. Om man någonsin får "assertion failure" när man kör programmet, betyder det att antagandet var fel. Normalt märks inte `assert()` i fungerande kod.

Man behöver aldrig ta bort `assert()`-satser ur sin kod, men när man (tror man har) debuggat färdigt kan man få bort dem ur programmet och därmed slippa den utökade exekveringstiden de medför. Det görs genom att kompilera med flaggan `-DNDEBUG`.

Kort sammanfattning

Det finns massvis med standardflaggor vid kompilering av C-program, följande har diskuterats i detta dokument:

- c endast kompilering (skapa objektfil)
- o bestäm namnet på utfilen
- l länka med angivet funktionsbibliotek
- L leta efter funktionsbibliotek även på angivet ställe
- g bevara debug-information i programmet
- D definiera konstant

Följande är specifik för gcc:

- Wall ger även varningsmeddelanden

Konstanterna som diskuterats:

- D__USE_FIXED_PROTOTYPES__ behövs tillsammans med -Wall vid kompilering på SunOS 4, för att slippa vilseledande varningar angående standardfunktioner
- DDEBUG kan användas för att villkorligt kompilera debug-satser i programmet
- DNDEBUG tar bort `assert()` ur programmet