Programmering i C++
Kompilering från kommandoraden

Sven Gestegård Robertz
Datavetenskap, LTH

9 november 2015

Sammanfattning

Ibland vill man, av olika anledningar, inte använda en stor integrerad utvecklingsmiljö (IDE), med den extra komplexitet det innebär, utan kompilera sin källkod direkt från kommandoraden. Detta kan ha fördelen att man enkelt kan ge olika flaggor direkt till kompilatorn, i stället för att behöva ändra inställningar i projekt, etc. Vad som är smidigast är ytterst en smaksak, och beror även på vilken storlek projektet har, och i vilken fas av utvecklingen man är.

Detta dokument försöker ge en introduktion till hur man kan kompilera sin källkod direkt från kommandoraden genom att dels anropa kompilatorn direkt, och dels använda bygg-verktyget `make`. Dokumentet förutsätter en *unix*-liknande miljö, vilket inkluderar Linux och MacOS. En liknande miljö kan fås under Windows med t ex *Cygwin*.

1 Inledning

När man programmerar i en integrerad utvecklingsmiljö (*IDE, Integrated Development Environment*) som *Eclipse* eller *Visual Studio* så är alla verktyg man använder när man skriver, bygger, testar och felsöker program samlade i ett användargränssnitt. Vi ska nu i stället se hur man kan använda dessa delar var för sig. Detta betyder att vi kommer att

skriva program i en valfri text-editor,

kompilera källkoden med en kompilator, direkt från kommandoraden, och

exekvera programmet från kommandoraden.

Därutöver kan man använda

byggverktyg (som t ex `make`) för att hantera beroenden mellan komponenter.

Detta dokument ger en första introduktion, med små exempel, till hur man kan arbeta med C++-program utan att använda en IDE. De exempel som ges ska ses som axplock för att hjälpa läsaren över den första tröskeln, men många detaljer utelämnas. För närmare information hänvisas till dokumentationen för respektive verktyg.

Vad gäller text-editorer finns det olika sorters text-editorer avsedda för programmering, och som passar olika människor. Det går inte att ge några generella rekommendationer, utan var och en får prova sig fram och bilda sig en egen uppfattning om vad som fungerar bäst.

2 Att anropa kompilatorn direkt från kommandoraden

I detta avsnitt kommer *gcc* att användas som exempel. Andra kompilatorer, som t ex *clang* fungerar på liknande sätt.

Ett minimalt exempel

Om vi antar at vi har en källkodsfil `hello.cpp`, så kan vi kompilera den till en exekverbar fil genom att ge kommandot [1]

```
> g++ hello.cpp
```

Om kompileringen och länkningen går bra kommer den exekverbara filen att skapas med namnet `a.out` (eller `a.exe` under Windows/Cygwin). För att ange namnet på den exekverbara filen som ska skapas använder man optionen `-o <filnamn>`, t ex

```
> g++ -o hello hello.cpp
```

För att exekvera den genererade filen (`hello`) ger man kommandot

```
> ./hello
```

Kompileringsflaggor

För att ändra hur kompilatorn arbetar ger man flaggor och argument på kommandoraden. Några exempel:

-std anger vilken standard av språket som ska användas. För att använda C++11 skriver man `-std=c++11`.

-W anger vilka varningar som ska användas. För att slå på alla varningar skriver man `-Wall`

-g anger att kompilatorn ska generera *debug-symboler*, så att man kan få mer information om källkoden när man kör sitt program i en debugger.

-O anger vilken grad av optimering kompilatorn ska använda. `-O0` betyder utan optimering. Det normala är `-O2` eller `-O3`.

En normal kommandorad för att kompilera sitt program är

```
> g++ -std=c++11 -g -Wall -o hello hello.cpp
```

[1] I de exempel som ges kommer följande notation att användas: Kommandoprompten som används är >, vilket betyder att detta betyder att användaren skriver `g++ hello.cpp` följt av `<ENTER>`.

3 byggverktyget `make`

Verktyget `make` är ett mycket kraftfullt verktyg för att bygga program och hantera beroenden mellan källkodsfiler, så att endast de filer som ändrats kompileras om. Ett byggverktyg är nödvändigt för att bygga stora system, men det är även smidigt för att beskriva kompilering av enskilda filer eller små system, för att slippa skriva långa kommandorader för varje kompilering.

`make` fungerar så att reglerna för hur bygget ska göras finns i filen `Makefile`, i samma katalog som källkodsfilerna. En `Makefile` innehåller en uppsättning regler. Exemplet i figur 1 betyder att filen `target` beror av filerna `dependency_1`, `dependency_2` etc. Om filen `target` inte finns, eller om någon av `dependency_k` är nyare än `target`, så kommer `make` att köra sekvensen av kommandon `command_1`, `command_2`, etc.

```
target: dependency_1 dependency_2 ... dependency_n
      command_1
      command_2
      ...
      command_n
```

Figur 1: Syntax för regler i en Makefile. OBS! rader som innehåller de kommandon, som hör till en regel, inleds med ett *tab*-tecken, och inte åtta *space*. I Exemplet här betyder det att de indragna rader med `command_k` inleds med ett *tab*-tecken. Att råka göra indraget med *space* i stället för *tab* är ett väldigt vanligt fel när man skriver Makefiler, så var nogrann med det.

Exempel

Till exempel, kan en Makefile som beskriver hur man bygger programmen `hello` och `test` se ut som i figur 2.

```
CXXFLAGS=-g -std=c++11 -Wall -pedantic-errors

hello : hello.cpp
        g++ $(CXXFLAGS) -o hello hello.cpp

test : test.cpp test.h my_lib.h
        g++ $(CXXFLAGS) -o test test.cpp
```

Figur 2: En enkel Makefile. Notera hur variabler (t ex `CXXFLAGS`) kan användas, samt hur `test` även har beroenden till två header-filer.

Generella regler, *wildcards*

För att slippa skriva väldigt många nästan likadana regler, kan man skriva generella regler genom att använda mönstermatchning. En enkel standard-makefile, som kompilerar varje `filnamn.cpp` till en exekverbar fil `filnamn` kan se ut som i figur 3.

```
CXXFLAGS=-g -std=c++11 -Wall -pedantic-errors

% : %.cpp
        g++ $(CXXFLAGS) -o $@ $<
```

Figur 3: Makefile som använder mönstermatchning. Variabeln `CXXFLAGS` innehåller de flaggor som ska ges till c++-kompilatorn, och de automatiska variablerna `$@` och `$<` som innehåller resultatet (*target*) respektive det *första* beroendet. Tecknet `%` är ett jokertecken (*wildcard*) som matchar ett godtyckligt mönster.

Med en Makefile enligt figur 3 betyder det att om man ger kommandot

```
> make hello
```

så kommer jokern `%` att matcha strängen `hello`, och alltså kommer `$@` resp `$<` att innehålla strängarna `hello` resp `hello.cpp`. Detta betyder att `make` ska bygga filen `hello`, som beror av filen `hello.cpp`. Som resultat kommer kommandoraden `g++ $(CXXFLAGS) -o hello hello.cpp` att exekveras (om filen `hello` inte finns, eller om `hello.cpp` är nyare än `hello`.

Notera att i figur 3 så beror den exekverbara filen (t ex `hello`) bara av källkodsfilen (t ex `hello.cpp`) men inte av några header-filer. Det betyder att om man gör en ändring i t ex `hello.h` och därefter gör `make` så kommer `hello` *inte* att byggas om. Det finns flera olika system för att hantera sådana beroenden, men de är för komplicerade för att beskrivas här. För små system är det enklaste att helt enkelt skriva explicita regler för varje kommando.

Inbyggda regler

Make har även inbyggda regler för många programspråk, och variabler som styr vilken kompilator som ska användas och vilka flaggor den få.

Vår generella `Makefile` i figur 3 har regler för C++ som täcks av de inbyggda reglerna. Då räcker det att definiera variablerna `CXX` och `CXXFLAGS`. Några regler behöver inte anges (figur 4).

På samma sätt som i figur 2, så vill man ibland ange fler beroenden för ett program, än enbart källkodsfilen med samma namn och ändelsen `.cpp`. För att bara ange beroendena, men använda standard-reglerna för hur filen ska byggas, så skriver man bara beroendena men inte några kommandon, som i figur 5.

```
CXXFLAGS=-g -std=c++11 -Wall -pedantic-errors
CXX=g++
```

Figur 4: Makefile utan några regler. Variablerna (vars namn är standardiserade) styr hur de inbyggda reglerna arbetar. I detta exemplet sätts `CXX`, som anger vilket kommando som ska användas för att kompilera C++-filer, och `CXXFLAGS`, som anger vilka flaggor som ska ges på kommandoraden till `$CXX`.

```
CXXFLAGS=-g -std=c++11 -Wall -pedantic-errors
CXX=g++

test : test.cpp test.h my_lib.h
```

Figur 5: Makefile som anger beroenden men använder standard-reglerna för att bygga filerna.

Med en `Makefile` enligt figur 5 kommer kommandot

```
> make hello
```

bara att bygga `hello` om filen saknas eller `hello.cpp` är nyare.

Däremot kommer kommandot

```
> make test
```

även att bygga `test` även om någon av filerna `test.h` och `my_lib.h` är nyare.

A Kort om att installera kompilator

MacOS

På en nyare MacOS är det enklaste att installera *XCode*. Alternativt kan man installera gcc (och ev. make), t ex via MacPorts, som är en pakethanterare för att installera open-source program under MacOS (`www.macports.org`).

Linux

Många linux-distributioner inkluderar gcc som standard Annars får man installera paketen som innehåller `gcc` och `make`. I debian-baserade distributioner (Debian, Ubuntu, etc.) heter paketen `gcc` respektive `make`. I redhat-baserade distributioner (t ex Fedora) behöver man paketen `gcc` och `gcc-c++` samt ev `make`. I andra borde de heta något liknande.

Windows

Under Windows är Cygwin (`www.cygwin.com/`) ett bra alternativ för att få en kommandoradsmiljö, kompilator [2] och verktygsprogram som liknar POSIX-/Unix. Cygwin är en pakethanterare, ungefär som MacPorts.

[2] Tyvärr verkar den kompilator som finns som "officiellt" paket i Cygwin vara en gammal version av `gcc` (4.6).