LINKÖPINGS UNIVERSITET

TDP005 Projekt: Objektorienterat system

Laboration i Make och CMake

Författare

Filip Strömbäck

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2016
Version 1.0

2016-10-04

Introduktion

I denna lab kommer vi titta närmare på två verktyg som gör det enklare att kompilera program skrivna i C och C++: Make och CMake.

För program med bara en handfull filer går det bra att kompilera med hjälp av `g++` i terminalen, men så snart programmet växer utöver det blir det snabbt ohållbart. Antingen kör man `g++ *.cpp -o program` varje gång, vilket ofta tar onödigt mycket tid, eller så måste man hålla reda på vilka filer man ändrat i och endast kompilera om dem. Make och CMake listar ut vilka filer som ska kompileras om baserat på vilka filer som har ändrats så att kompileringen går så snabbt som möjligt.

För att se vad Make och CMake kan hjälpa oss med kommer vi använda miniräknaren från CLion-labben. Se därför till att ha källkoden från `https://gitlab.ida.liu.se/filst04/tdp005-lab.git` tillgänglig under labben.

1 Kompilera manuellt

För att bättre se vad verktygen hjälper till med börjar vi med att kompilera miniräknaren manuellt. Det enklaste sättet att göra detta på är genom att skriva:

```
g++ -g -std=c++11 -Wall -Wextra *.cpp -o calc
```

Som ni kanske märker så tar det tid att kompilera all kod på detta sätt. Varje gång vi kompilerar så kommer kompilatorn att kompilera om alla filer, trots att vi bara har ändrat i en eller ett fåtal filer sedan förra gången vi kompilerade programmet. På grund av detta blir denna metod snabbt ohållbar då programmet växer. Det tar helt enkelt för lång tid att kompilera all kod varje gång.

Det skulle vara bättre om vi kunde kompilera alla filer en gång och sen bara kompilera de filer som har ändrats sedan förra kompileringen. För att göra detta kan vi kompilera vår kod i två steg. Först *kompilerar* vi alla källkodsfiler (`.cpp`-filer) till objektfiler, sen *länkar* vi ihop dessa objektfiler till en körbar fil. För att kompilera miniräknaren manuellt på detta vis kan vi göra så här:

```
g++ -g -c -std=c++11 -Wall -Wextra constant.cpp -o constant.o
g++ -g -c -std=c++11 -Wall -Wextra expression.cpp -o expression.o
g++ -g -c -std=c++11 -Wall -Wextra main.cpp -o main.o
g++ -g -c -std=c++11 -Wall -Wextra operator.cpp -o operator.o
g++ -g -c -std=c++11 -Wall -Wextra parse.cpp -o parse.o
g++ -g -c -std=c++11 -Wall -Wextra tokenize.cpp -o tokenize.o

#Länka ihop alla objektfiler:
g++ -g -std=c++11 *.o -o calc
```

Först kompilerar vi alla `.cpp`-filer till objektfiler med hjälp av `-c`-flaggan (som står för *compile*). Sedan länkar vi ihop dem till en körbar fil genom att köra `g++` som vanligt, men i stället för att ge den `.cpp`-filer ger vi den `.o`-filer.

Om vi nu skulle ändra i exempelvis filen `expression.cpp` behöver vi bara kompilera om `expression.o` och sedan länka ihop programmet igen. Alla andra filer behöver inte kompileras om, eftersom en kompilerad version av de filerna redan finns i motsvarande `.o`-fil. Fördelen med att göra detta är som sagt att det går mycket fortare att kompilera då man bara ändrat i ett fåtal filer. Däremot är det inte särskilt praktiskt att göra detta manuellt eftersom det är mycket att hålla reda på och mer än ett kommando som måste köras. Detta skulle man kunna halvautomatisera med hjälp av en `.sh`-fil, men det blir snabbt ohållbart det med.

2 Make

Som vi just såg är det inte särskilt praktiskt att göra kompileringen i två steg manuellt. Det är på tok för mycket att hålla reda på för att det ska vara värt den tidsvinst det innebär. Däremot kan vi låta Make göra det åt oss. Då kan vi ”lära” make hur vårt program ska byggas en gång och sedan behöver vi inte bry oss om det längre. Alltså får vi både tidsvinsten från att inte kompilera om alla filer varje gång utan att det blir svårare att kompilera programmet.

2.1 Regler

När man kör kommandot `make` så kommer Make läsa instruktioner från en fil som heter `Makefile` för att ta reda på vad som ska göras. `Makefile` innehåller en uppsättning *regler* som beskriver vad som ska byggas och hur det ska byggas. En regel är en beskrivning av hur Make ska producera ett visst mål. Om Make inser att den behöver skapa filen `expression.o` så kommer Make leta efter en regel som berättar hur detta ska göras. Vi kan skriva den regeln så här:

```
expression.o:
  <tab>  g++ -g -c -std=c++11 -Wall -Wextra expression.cpp -o expression.o
```

Var noga med formatteringen, Make är ganska petig! `<tab>` ska vara ett tab-tecken, inte fyra eller åtta mellanrum. De flesta texteditorer har koll på detta och hjälper till så gott de kan. Syntaxen är alltså: `fil som ska skapas` följt av ett kolon, sen alla kommandon som ska köras för att producera filen på nästkommande rader, indenterade med ett tab-tecken.

Se nu till att ta bort alla `.o`-filer, så kan vi testa vår regel. Kör sedan `make`, så försöker Make skapa filen `expression.o` eftersom det är den första regeln som finns i makefilen (man kan också köra `make expression.o` för att explicit ange vad make ska skapa). Make kommer då att hitta vår regel och köra det kommandot vi skrev där. När Make kör kommandon så skriver den som standard ut dem så att vi ser vad Make gör. Om allt gått rätt bör filen `expression.o` ha skapats. Testa då att köra `make` igen. Denna gång ser `make` att filen `expression.o` redan finns och beslutar därför att inget behöver göras.

2.2 Beroenden

För att uppnå vårt mål vill vi att Make ska bygga om `expression.o` när `expression.cpp` har ändrats. Vi kan testa detta genom att ändra något i `expression.cpp` (det räcker att göra `touch expression.cpp`) och sedan köra `make` igen. Tyvärr kommer Make att svara med: `make: `expression.o' is up to date`, vilket innebär att Make tycker att inget behöver göras. Detta beror på att vi inte har berättat för Make vilka filer vi använder för att bygga `expression.o`, så Make antar att inget behöver göras så länge filen finns.

För att berätta för Make att vi skapar `expression.o` baserat på `expression.cpp` så vill vi säga åt Make att `expression.o` *beror på* `expression.cpp`. Det innebär att Make kommer undersöka vilken av `expression.o` och dess beroenden som är modifierad senast. Om `expression.o` är modifierad senast kommer Make besluta att inget har ändrats sedan sist och inte göra något, annars kommer Make att besluta att `expression.o` måste byggas om eftersom någon av beroendena har ändrats sedan förra gången Make kördes. Detta skrivs på följande sätt:

```
expression.o: expression.cpp
  <tab>  g++ -g -c -std=c++11 -Wall -Wextra expression.cpp -o expression.o
```

Vi kan nu verifiera att det fungerar som vi vill genom att återigen ändra `expression.cpp` och köra `make` igen. Om allt gått som det ska borde Make nu bygga om `expression.o`, precis som vi ville.

2.3 Variabler

Nu när vi har lyckats kompilera en objektfil korrekt kan vi skapa liknande regler för alla `.cpp`-filer i miniräknaren. Det vi snabbt kan se är att vi upprepar flaggorna vi skickar till kompilatorn väldigt många gånger, så om vi senare inser att vi behöver lägga till en kompilatorflagga blir det ganska jobbigt. För att underlätta detta och dessutom göra vår makefil mer lättläslig kan vi använda variabler.

Alla variabler i Make är strängar. Att skapa en variabel i Make ser ut som en tilldelning i många andra språk, och det fungerar som en `#define` i C/C++. För att skapa variabeln för de flaggor vi vill ha till kompilatorn (den brukar heta `CXXFLAGS`), kan man göra så här:

```
CXXFLAGS = -g -std=c++11 -Wall -Wextra
```

För att sedan använda en variabel skriver man `$(variabelnamn)`. Man kan självklart använda variabler när man definierar andra variabler, så vi skulle kunna skapa en separat variabel som innehåller alla varningsflaggor och använda denna som en del för `CXXFLAGS`. Med hjälp av detta kan vår regel förenklas till följande:

```
expression.o: expression.cpp
  <tab>  g++ -c $(CXXFLAGS) expression.cpp -o expression.o
```

Det kan också vara vettigt att använda variablerna `$@` och `$^`. `$@` innehåller målfilen för regeln, och `$^` innehåller alla filer målet beror på.

Nu kan det vara lämpligt att skriva målet för den körbara filen, `calc`. Utöver `CXXFLAGS` brukar också variabeln `LDFLAGS` skickas till kompilatorn i detta mål eftersom vissa kompilatorflaggor endast ska skickas med till kompilatorn under länkningsfasen (exempelvis om vi vill använda några externa bibliotek). Det kan exempelvis se ut så här:

```
calc: expression.o constant.o main.o operator.o parse.o tokenize.o
  <tab>  g++ $(CXXFLAGS) $(LDFLAGS) $^ -o $@
```

I detta fall kommer alltså `$@` innehålla `calc`.

Om detta mål läggs först i makefilen bör miniräknaren kunna kompileras genom att köra `make`. Testa även att ändra i en `.cpp`-fil och kompilera igen. Make bör inse att `calc` beror på en objektfil som i sin tur beror på en `.cpp`-fil och därmed bygga om denna kedja.

2.4 Headerfiler

Testa nu att ändra i en headerfil, exempelvis `expression.h`. Vad händer om vi kör `make` nu? Vilka filer borde byggas om?

För att fixa detta måste vi uppdatera beroendena för våra objektfiler. Exempelvis `constant.o` beror just nu bara på `constant.cpp`, men egentligen stämmer inte det. Den beror även på `constant.h` och `expression.h` eftersom den inkluderar dessa filer. Vi måste alltså lägga till dessa filer som beroenden till våra mål. Om vi gör detta kan vi inte längre använda `$^` för att skicka alla beroenden till kompilatorn. Eftersom vi bara vill skicka med `.cpp`-filen, vilket borde vara det första beroendet, kan vi använda `$<` i stället. Lägg till headerfiler som beroenden och testa vad som kompileras om när olika headerfiler ändras. Det kan underlätta att rita upp en beroendegraf på papper för att enklare se vad som beror på vad.

Eftersom detta är aningen omständigt att göra manuellt kan man också generera dessa beroenden genom att låta `g++` generera dem med hjälp av flaggan `-M`. Vi kommer inte titta närmare på detta i den här labben.

2.5 Andra mål

Utöver att bygga programmet brukar en makefil också innehålla ett annat mål: `clean`. Om vi kör målet `clean` med kommandot `make clean` förväntar vi oss att Make tar bort alla temporära filer som har skapats under tidigare kompilationer. Detta kan vara bra om man vill bygga om hela programmet från noll, vilket man ibland vill göra om saker börjar bete sig konstigt eller om man börjar få slut på diskutrymme.

Målet `clean` ska alltså bara se till att ta bort alla körbara filer och objektfiler. Det kan se ut som följer:

```
clean:
  <tab>  rm -f *.o calc
```

Detta fungarar bra så länge det inte finns en fil som heter `clean`. Finns `clean` så kommer Make, som vi tidigare såg, konstatera att målet redan är byggt och att inget behöver göras. I detta fall förväntar vi oss inte att `clean` är en fil, så vi kan berätta detta för Make genom att märka `clean` som ett *Phony target*:

```
.PHONY: clean
clean:
  <tab>  rm -f *.o calc
```

Ibland är det också användbart att ha ett mål som heter `zap` som också tar bort eventuella temporärfiler från Emacs eller liknande. Detta mål kan bero på `clean` så behöver man inte repetera logiken för det målet i `zap`.

3 CMake

Som ni märkt är det tidskrävande att skriva en bra makefil från noll varje gång eftersom Make inte känner hur C++ fungerar särskilt väl. Eftersom fler har tyckt att detta är omständigt finns CMake.

CMake är ett verktyg för att kompilera C- och C++-kod. Den gör det genom att läsa instruktioner från filen `CMakeLists.txt` och generera exempelvis en makefil (eller projektfiler för Visual Studio etc.). Sen kan vi använda `make` för att kompilera projektet som tidigare.

Filen `CMakeLists.txt` ser ut ungefär som följer:

```
project(calc)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14")

set(SOURCE_FILES
    constant.cpp
    constant.h
    expression.cpp
    expression.h
    main.cpp
    operator.cpp
    operator.h
    parse.cpp
    parse.h
    tokenize.cpp
    tokenize.h)

add_executable(calc ${SOURCE_FILES})
```

Här säger vi till CMake att vi har ett projekt som heter `calc`, sen lägger vi till flaggan `-std=c++14` för C++-kompilatorn. Sen skapar vi en lista med alla filer vi vill kompilera (som vi kallar för `SOURCE_FILES`) och säger åt CMake att den ska producera en körbar fil som heter `calc` av dem.

För att kompilera projektet, kör nu `cmake .` för att generera en makefil. Kör sedan `make` för att bygga miniräknaren. Det räcker med att köra `cmake` då ni ändrat i `CMakeLists.txt` (exempelvis då ni lagt till en fil). Så i normala fall räcker `make` för att kompilera programmet.

Projekt i CLion beskrivs av en `CMakeLists.txt`-fil. Så när ni kompilerar kod i CLion körs CMake i bakgrunden.

3.1 Använda bibliotek i CMake

När man använder ett bibliotek i C++ måste man berätta det för kompilatorn, så att den kan länka programmet korrekt. Kompilerar man sitt program via kommandoraden så gör man det genom att använda flaggan `-l` när man länkar programmet. Vill man exempelvis använda biblioteket `libpng` (för att avkoda PNG-filer) lägger man till flaggan `-lpng`.

För att göra motsvarande i CMake kan man lägga till kommandot `target_link_libraries` efter `add_executable` på följande sätt:

```
# ... samma som förut
add_executable(calc ${SOURCE_FILES})
target_link_libraries(calc png)
```

Det här enkla sättet fungerar bra för bibliotek som är installerade globalt i systemet. Ibland räcker inte det enkla sättet eftersom man kanske har flera versioner av ett bibliotek installerat, eller så kanske man inte kan (eller vill) installera ett visst bibliotek i hela systemet. Det är fallet med SFML på skolans Linux-datorer, där det finns en ”vanlig” version av SFML och en version som är kompilerad för GCC 6.1.0. I dessa fall behöver man instruera CMake om att leta efter biblioteken på egen hand. För SFML görs det på följande sätt:

```
project(sfml)

# Använd C++ 14.
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++14")

# På skolans Linux-system finns två versioner av SFML. Se vilken som används:
if (DEFINED ENV{LOADEDMODULES} AND ("$ENV{LOADEDMODULES}" MATCHES ".*prog/gcc/6.1.0.*"))
    set(CMAKE_CXX_COMPILER /sw/gcc-6.1.0/bin/g++)

    if (DEFINED ENV{SFML_ROOT})
        list(APPEND CMAKE_MODULE_PATH "$ENV{SFML_ROOT}/share/SFML/cmake/Modules")
    endif()
endif()

# Välj vilka delar av SFML som ska användas.
set(SFML_MODULES network graphics window system)

# Försök att hitta SFML genom att anropa 'FindSFML.cmake'.
find_package(SFML 2 COMPONENTS ${SFML_MODULES})
include_directories(${SFML_INCLUDE_DIR})

# Om det inte gick hoppas vi att det är installerat i alla fall.
if (NOT (${SFML_FOUND} STREQUAL "TRUE"))
```

```
    set(SFML_LIBRARIES, "")
    foreach(i ${SFML_MODULES})
        list(APPEND SFML_LIBRARIES "sfml-${i}")
    endforeach(i)
endif()

# Ange källfiler, lägg till de filer som ni behöver!
set(SOURCE_FILES sfml.cpp)

# Säg till CMake att skapa en körbar fil som heter 'sfml'.
add_executable(sfml ${SOURCE_FILES})

# Länka med biblioteken som vi hittade tidigare.
target_link_libraries(sfml_test ${SFML_LIBRARIES} ${SFML_DEPENDENCIES})
```

Raden som anger `SOURCE_FILES` är densamma som tidigare.

När ni kompilerar på skolans datorer vill ni först lägga till modulen med den senaste C++-kompilatorn genom att köra:

```
module add prog/gcc/6.1.0
module initadd prog/gcc/6.1.0
module add prog/sfml-2-gcc-6
module initadd prog/sfml-2-gcc-6
```

Om det inte fungerar, testa att ta bort filen `CMakeCache.txt` och mappen `CMakeFiles` och kör CMake igen.