LINKÖPINGS UNIVERSITET

TDP005 Projekt: Objektorienterat system

Introduktion till Make

Författare

Filip Strömbäck; Dag Jönsson

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2026
Version 1.1

29 oktober 2026

Introduktion

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

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 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 kan hjälpa oss med kommer vi fortsätta med ert SFML demo från tidigare.

1 Kompilera manuellt

För att bättre se vad verktygen hjälper till med börjar vi med att kompilera ert SFML-demo från tidigare lab manuellt. Det enklaste sättet att göra detta på är genom att skriva:

```
g++ -g -std=c++17 -Wall -Wextra main.cpp game.cpp \
  -lsfml-graphics -lsfml-window -lsfml-system -o sfml
```

Som ni kanske har börjat inse så är det ganska tröttsamt att skriva hela den raden (eller snarare att trycka uppil tills ni hittar den i historiken). Fundera nu på hur hållbart detta är om ni börjar lägga till fler och fler filer. Detta arbetssätt har också problemet med att alla era filer kommer kompileras om, även om ni inte har ändrat dem. Detta är kanske inte ett så stort problem just nu, men senare i ert projekt kan detta börja ta mer och mer tid.

För att avhjälpa det senare problemet skulle det vara bättre om vi kunde kompilera alla filer en gång och sen bara kompilera om 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 ert SFML-demo manuellt på detta vis kan vi göra så här:

```
g++ -c -g -std=c++17 -Wall -Wextra main.cpp
g++ -c -g -std=c++17 -Wall -Wextra game.cpp

#Länka ihop alla objektfiler:
g++ -g -std=c++17 -lsfml-graphics -lsfml-window -lsfml-system *.o -o sfml
```

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. Det är även i detta steg vi specificerar att det behöver länkas med externa bibliotek.

Om vi nu skulle ändra i exempelvis filen `game.cpp` behöver vi bara kompilera om `game.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` i den nuvarande mappen 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 `game.o` så kommer Make leta efter en regel som berättar hur detta ska göras. Vi kan skriva den regeln så här:

```
game.o:
  <tab>  g++ -c -g -std=c++17 -Wall -Wextra game.cpp
```

Var noga med formateringen, 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 `game.o` eftersom det är den första regeln som finns i Makefilen (man kan också köra `make game.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 `game.o` ha skapats. Testa då att köra `make` igen. Denna gång ser `make` att filen `game.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 `game.o` när `game.cpp` har ändrats. Vi kan testa detta genom att ändra något i `game.cpp` (det räcker att göra `touch game.cpp`) och sedan köra `make` igen. Tyvärr kommer Make att svara med: `make: `game.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 `game.o`, så Make antar att inget behöver göras så länge målfilen finns.

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

```
game.o: game.cpp
  <tab>  g++ -c -std=c++17 -g -Wall -Wextra game.cpp
```

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

2.3 Variabler

Nu har ni bara 2 filer som ni vill kompilera, men i ert projekt kommer ni ha många fler, och ni kommer vilja använda samma flaggor på flera olika ställen. För att inte behöva upprepa dessa för varje fil kan vi spara dessa i en variabel. Detta gör det också lätt att ändra flaggorna om ni inser att ni vill göra det, och er Makefile blir även lite lättare att läsa.

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++17 -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:

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

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, `sfml`. 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:

```
sfml: game.o main.o
  <tab>  g++ $(CXXFLAGS) $^ $(LDFLAGS) -o $@
```

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

Om detta mål läggs först i makefilen bör ert SFML-demo kunna kompileras genom att köra `make`. Testa även att ändra i en `.cpp`-fil och kompilera igen. Make bör inse att `sfml` 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. Vad händer om vi kör `make` nu? Vilka filer borde byggas om?

För att fixa detta måste vi uppdatera beroendena för våra objektfiler. Exempelvis `game.o` beror just nu bara på `game.cpp`, men egentligen stämmer inte det. Den beror även på `game.hpp`. Vi måste alltså lägga till denna fil som beroenden till vårt 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 headerfilen som beroende och testa ändra i headerfilen, och se att `game.o` kompileras om när header filen ändras.

Eftersom detta är aningen omständigt att göra manuellt kan man också generera dessa beroenden genom att låta `g++` generera dem med hjälp av flaggorna `-MMD -MP`. 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 sfml
```

Detta fungerar 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 sfml
```

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 Slutmål

Make är ett extremt kraftfullt verktyg som är värt att lära sig att använda. I detta fall nöjer vi oss med att ni har landat i en Makefile som ni känner er bekväma med, och ni vet hur ni lägger till nya filer och kopplar ihop de olika beroenden ni kommer skapa i ert projekt. Målet är alltså att ni bara ska behöva skapa *.o-målet för varje enskild fil, och lägga till detta mål som beroende för den körbara filen.

4 Ytterligare läsning

Om ni är intresserade av att lära er mer om Make så kan ni börja med att studera den officiella manualen här: `https://www.gnu.org/software/make/manual/make.html`. Den går igenom allt som Make kan göra, vilket är ganska mycket. För kursens skull behöver ni inte studera denna manual, det som har behandlats i denna introduktion är tillräckligt.

5 Överkurs: CMake

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

CMake är ett verktyg för att bygga C- och C++-projekt. 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(sfml)

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++17")

set(SOURCE_FILES
  <tab>  main.cpp
  <tab>  game.cpp
  <tab>  )

add_executable(sfml ${SOURCE_FILES})
```

Här säger vi till CMake att vi har ett projekt som heter `sfml`, sen lägger vi till flaggan `-std=c++17` 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 `sfml` av dem.

För att kompilera projektet, kör nu `cmake .` för att generera en Makefil. Kör sedan `make` för att bygga ert SFML-demo. 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. I detta fallet kommer det inte riktigt funka då CMake har inte fått förklarat för sig att den behöver länka in SFML också.

5.1 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(sfml ${SOURCE_FILES})
target_link_libraries(sfml 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.

```
cmake_minimum_required(VERSION 3.10)
project(sfml)

# Använd C++ 17.
set(CMAKE_CXX_STANDARD 17)
```

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

# Lägg till lämpliga bibliotek till kompileringen. Man kan använda 'find_package',
# men det kräver att SFML är installerat på ett annat sätt.
foreach(i ${SFML_MODULES})
  list(APPEND SFML_LIBRARIES "sfml-${i}")
endforeach(i)

# Ange källfiler, lägg till de filer som ni behöver!
set(SOURCE_FILES main.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 ${SFML_LIBRARIES} ${SFML_DEPENDENCIES})
```

Raden som anger `SOURCE_FILES` är densamma som tidigare.