



Mittuniversitetet

MID SWEDEN UNIVERSITY

Metoder och verktyg i mjukvaruprojekt

Lektion 3 – Byggverktyg

- *Installation av Cmake*
- *Uppsättning och installation av utvecklingsmiljön*
- *Konfiguration av Qt Creator / Clion*
- *Använda CMake och Git i relation till Qt Creator / Clion*
- *Använda Git Submodule*

Innehållsförteckning

1 Installera verktyg.....	3
1.1 CMake.....	3
1.1.1 Linux.....	3
1.1.2 Windows.....	4
1.1.3 Mac OSX.....	4
1.2 Utvecklingsmiljö.....	4
1.3 Övriga utvecklingsbibliotek.....	5
2 Konfiguration av utvecklingsmiljö.....	6
2.1 Konfigurera CLion.....	6
2.2 Konfigurera Qt Creator.....	7
3 Testprojekt.....	10
3.1 Debug / Release Mode.....	12
3.1.1 CLion.....	15
3.1.2 Qt Creator.....	16
4 Git Submodule.....	17
Referenser.....	20
Bilaga A: Cmake Referenser.....	21
Bilaga B: Skapa klasser & filer med Qt Creator.....	22

1 Installera verktyg

1.1 CMake

I denna kurs kommer vi använda **CMake** som är ett s.k. *cross platform* verktyg för att bygga C/C++ program. Fördelen med *CMake* är att den låter dig skapa en struktur på hur ett projekt ska byggas oberoende av underliggande utvecklingsmiljö och plattform. Tanken är att man i *Git* endast lagrar **byggskriptet** för *CMake* samt projektets källkod. De filer som är ett resultat av själva kompileringen, eller genereringen, skall alltså inte versionshanteras. Detta eftersom det blir redundant i och med att de skapas utifrån befintliga filer. På så sätt slipper man problem som uppstår när de genererade filerna i *Git* inte är uppdaterade samt eventuella konflikter med plattformsberoende filer.

För att klargöra vilka filer som ej skall ingå i versionshanteringen så anger man dem i ett särskilt manifestdokument **.gitignore** [11] som läggs direkt under projektroten. I detta dokument skriver man helt enkelt vilka konkreta filer / mappar man vill utesluta, men även mer avancerade mönster kan användas för att ange omfånget av filändelser samt särskilda undantag. Du förväntas fördjupa dig i **.gitignore** och använda det på lämpligt vis alltigenom kursen!

Det finns två sätt att installera *CMake*, antingen laddar du hem ett installationspaket och installerar det på din dator eller så lägger du till det manuellt med anvisningarna nedan.

Viktigt är att du installerar version 3.2 eller senare (vi kommer använda funktioner som kräver denna version för att göra det lättare för oss).

1.1.1 Linux

Om det inte finns en nyare version än 3.2 att installera via pakethanteraren måste du göra det manuellt genom att följa stegen nedan.

1. Gå till följande [sida](#) [1] och ladda ned **cmake-X.X.X-Linux-x86_64.tar.gz** där **X.X.X** är versionsnumret och **x86_64** är din processorarkitektur.
2. Packa upp innehållet till **~/Dev/Tools/cmake-X.X.X** eller välj en annan katalog som passar bättre för din miljö.
3. Lägg till **cmake-X.X.X/bin** till **PATH**:
 1. Redigera **.bashrc** (**nano ~/.bashrc**)
 2. Lägg till **export PATH=~/Dev/Tools/cmake-X.X.X/bin:\$PATH**
4. Starta om terminalen eller läs in *bash*-filen igen (**source ~/.bashrc**)
5. Verifiera att allt fungerar genom att köra kommandot **cmake -version** i terminalen, om allt gått som det skulle kommer cmake-versionen skrivas ut. Om detta inte fungerar gå baklänges genom listan tills du hittar ett fel.

1.1.2 Windows

1. Gå till följande [sida](#) [1] och ladda ned **cmake-X.X.X-win32-x86.zip** där **X.X.X** är versionsnumret och **x86** är din processorarkitektur.
2. Packa upp innehållet till **C:/Dev/Tools/cmake-X.X.X** eller välj en annan katalog som passar bättre för din miljö.
3. Lägg till **cmake-X.X.X/bin** till **PATH**:
 1. Kontrollpanelen → System → Avancerade systeminställningar
 2. Avancerat → Miljövariabler
 3. **Systemvariabler**; redigera **path** och lägg till sökvägen till bin-katalogen i cmake (sökvägarna separeras med semikolon).
4. Starta om kommandotolken eller *Git Bash*.
5. Verifiera att allt fungerar genom att köra kommandot **cmake -version** i kommandotolken / *Git Bash*. Om allt gått som det skall kommer cmake-versionen skrivas ut. Om detta inte fungerar, gå baklänges genom listan tills du finner ett fel.

1.1.3 Mac OSX

Gå till följande [sida](#) [1] och ladda ned **cmake-X.X.X-Darwin-x86_64.tar.gz** där **X.X.X** är versionsnumret och **x86_64** är din processorarkitektur. Följ sedan anvisningarna för installation under Linux, med start på steg 2.

1.2 Utvecklingsmiljö

CMake kan generera projektfiler för mängder av *utvecklingsmiljöer* (*IDE's*), plattformsspecifika såsom **XCode** och **Visual Studio** såväl som plattformsberoende som **CLion** och **Qt Creator**. Ni har därför möjlighet att använda den *IDE* som ni själva trivs bäst med, men detta lektionsmaterial behandlar enbart konfigurationer av de två sistnämnda alternativen.

Både **CLion** och **Qt Creator** finns som sagt tillgängligt för alla plattformar (*Windows, Linux, OS X*) och använder CMake som byggsystem. **Qt Creator** är fritt tillgängligt som *open source* medan **CLion** kräver en licens för att användas, utöver sin 30-dagars testperiod. Lyckligtvis erbjuder **JetBrains** studentlicenser för alla sina produkter och registrerar ni er med er studentemail så får ni utöver **CLion** även tillgång till fullversioner av deras övriga högt ansedda utvecklingsmiljöer, såsom **IntelliJ IDEA** (*Java / JVM*) samt **PyCharm** (*Python*). De som använder **Visual Studio** som *IDE* går heller inte lottlösa utan kan glädjas över deras verktyg **ReSharper**, som avser underlätta utvecklingen av C++ projekt i Windowsmiljö. Ni kan läsa mer om **JetBrains** produkter samt registrera er för gratislicens på deras [hemsida](#) [7]!

Om valet av *IDE* faller på alternativen **Qt Creator** och **CLion** så kan det vara värt att notera att den sistnämnda är en något mer modern *IDE* som utvecklas/underhålls av en organisation med mycket stor erfarenhet kring utformning av utvecklingsmiljöer.

Ni installerar **CLion** antingen genom att först installera [Toolbox App](#) [8], från vilken sedan **CLion** och **JetBrains** övriga verktyg kan installeras och uppdateras, eller så laddar ni ned och installerar **CLion** som separat applikation.

När det gäller **Qt** så har ni också olika möjligheter för installation. Antingen tar ni hem hela **QT SDK** med **Qt Creator** (OBS! vi kommer inte att använda **Qt**-biblioteket i denna kurs) eller så tar ni endast hem **Qt Creator**. Det lättaste är att ta hem hela **SDK**, då det blir mindre konfiguration. Nedladdning finner ni på följande [sida](#) [2].

Windowsanvändare rekommenderas att installera MinGW-kompilatorn. För Qt Creator finns installation som inkluderar detta;

[Qt 5.6.2 for Windows 32-bit \(MinGW 4.9.2, 1.0 GB\)](#)

För CLion är rekommendationen att installera [mingw-w64](#) [9] som inkluderar full support för kursens behov!

1.3 Övriga utvecklingsbibliotek

För att göra utvecklingen både roligare och lättare så har vi valt att inkludera ett par tredjepartsbibliotek. Skapa en katalogstruktur som heter exempelvis **Dev/Tools/include**, skapa sedan en miljövariabel som heter **TOOLS_INCLUDE** som pekar på denna katalog (det viktiga är inte var katalogen finns men att variabeln med det exakta namnet finns och att den innehåller sökvägen till katalogen).

*Genom att använda samma miljövariabel när vi sedan bygger kan samma projekt enkelt byggas på olika datorer / operativsystem utan att man behöver gå in och uppdatera alla sökvägar i byggsriptet. Det kan tyckas vara en enkel sak att byta namn på en sökväg i ett skript, men när man arbetar i ett projekt tillsammans med andra utvecklare där det används olika sökvägar så måste alla uppdatera denna varje gång någon gör en uppdatering av den aktiva **branchen** i Git.*

Memstat & Terminal

Mikael Nilsson är en av kursens tidigare lärare och utformade betydande delar av momentet för byggverktyg. Han har bland annat skapat ett par programvaror för att underlätta projektutvecklingen. **Memstat** är ett litet bibliotek som används i debuggingsyfte för att identifiera minnesläckor medan **Terminal** gör det enkelt att skriva till terminalen i olika färger, samt flytta utskriftsmarkören. Eftersom **Terminal** inte är något **header only** bibliotek så kommer vi att lägga till det som en s.k. **submodule** till projektet, men mer om det senare i lektionen.

Catch

[Catch](#) [3] är ett testramverk som från början är skrivet av Phil Nash, men som nu drivs som ett *community open source* projekt. Detta ramverk kommer vi att använda senare i kursen när vi går igenom **Test Driven Development** (TDD).

För att göra det enkelt att installera så finns alla *header-filer* på *Bitbucket* och ni installerar dem genom att gå in i katalogen **TOOLS_INCLUDE** och skriva följande *Gitkommando* (glöm inte punkten på slutet).

```
$ git clone git@bitbucket.org:miun_dt042g/tools-include.git .
```

2 Konfiguration av utvecklingsmiljö

Tillvägagångssätten för konfigurationen av **Qt Creator** jämte **CLion** skiljer sig åt, såväl vad gäller programvarornas olika versioner som anpassning för er egen systemmiljö. Instruktionerna nedan tar inte hänsyn till alla möjliga problem som kan uppstå och ni förväntas lösa eventuella tekniska utmaningar mycket på egen hand och med hjälp av era medstudenter.

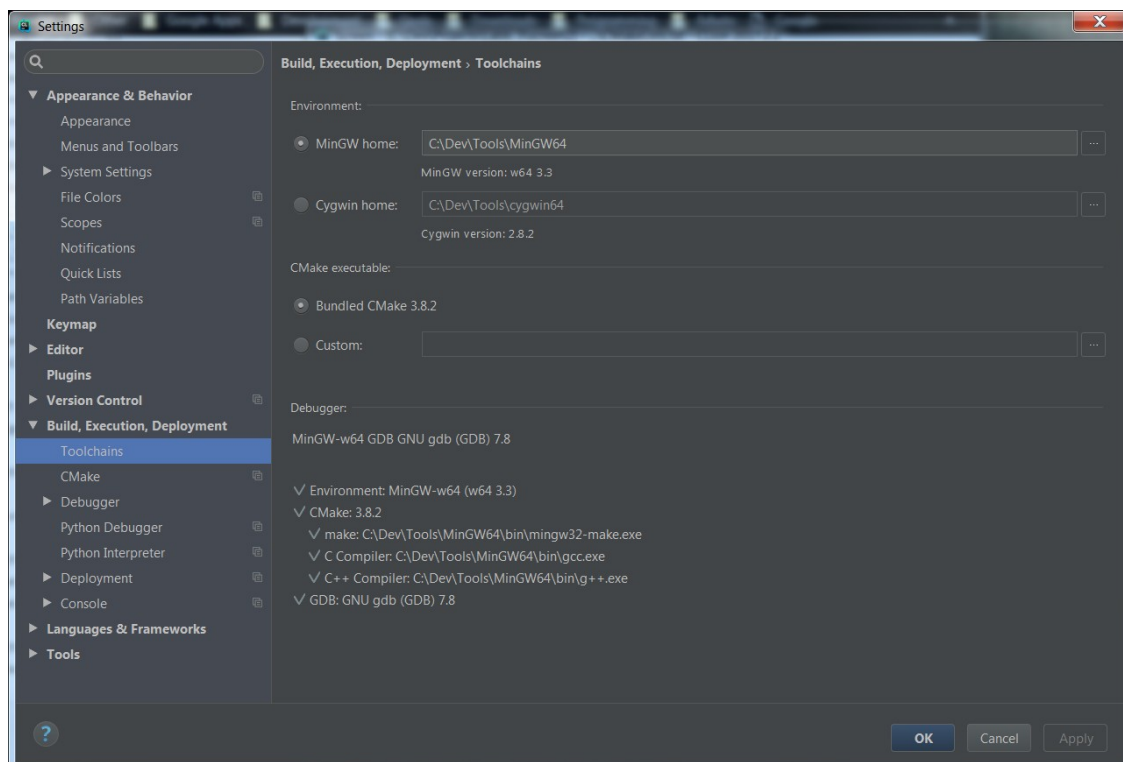
***CLion** är nytillkommet tillskott för kursen och testad för Windows 7 64-bit med **mingw-w64** kompilatorn. Informationen för **Qt Creator** är en kvarleva från tidigare kursomgångar och har inte uppdaterats.*

2.1 Konfigurera CLion

För att kunna bygga ditt projekt måste du ange en kompilator med stöd för C++11. För Windows faller detta huvudsakligen ned på de två alternativen **Cygwin** och **MinGW**. Vilken av dessa kompilatorer som väljs spelar mindre roll men vissa distributioner kan medföra en del manuellt arbete för att kunna stödja bland annat POSIX-trådar. Viktigt är också att en **debugger** medföljer installationen, varav [The GNU Project Debugger \(GDB\)](#) [10] är ett lämpligt alternativ. Den som följde rekommendationen i sektion [1.2 Utvecklingsmiljö](#) och installerade [mingw-w64](#) har allt nödvändigt stöd samt **GDB**.

Du anger vilka verktyg som skall användas för *kompilering*, *debugging* samt *byggsystem* i programmets inställningar;

File → Settings → Build, Execution, Deployment → Toolchains



För att aktivera integrationen av Git i ett befintligt projekt;

- **VCS** → **Enable Version Control Integration...** och välj **Git** i listan för alternativ.
- Om **git.exe** inte kan hittas så måste du ange dess path; **File** → **Settings** → **Version Control** → **Git** och för **Path to Git executable** anger du sökvägen till den exekverbara filen (e.x. **C:\Dev\Tools\Git\bin\git.exe**).
- Om **CLion** ger felmeddelandet **Invalid VCS root mapping...** så måste du koppla projektroten till Git; **VCS** → **Import into Version Control** → **Create Git Repository...** vilket skapar mappen **.git** i roten till projektet.

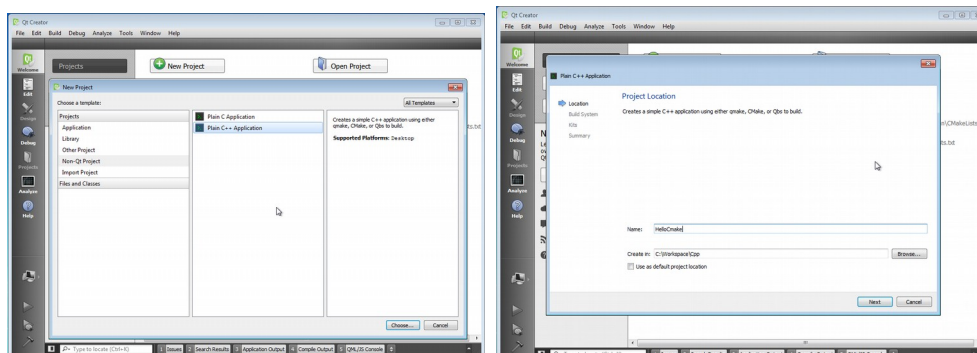
Om felmeddelandet kvarstår behöver du starta om **CLion**!

*All information relaterad till CLion för din specifika utvecklingsmiljö finns lagrad under katalogen **.idea** i projektroten. Tänk på att denna information antagligen inte kommer vara relevant för andra deltagare. För att slippa se denna katalog under **untracked files** så inkluderar du ***.idea/** i **.gitignore**.*

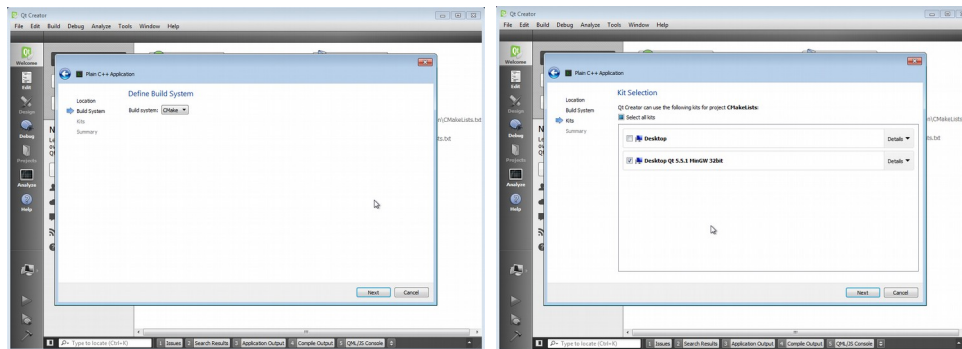
*Många Git-kommandon fungerar direkt i CLion och finns under menyn **VCS** → **Git***

2.2 Konfigurera Qt Creator

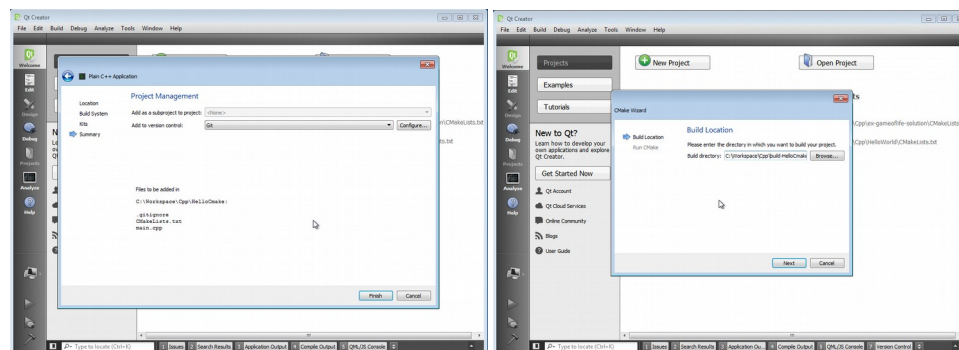
1. Skapa ett nytt projekt; **New Project** → **Non-Qt Project** → **Plain C++ Application**.
2. Ge projektet ett lämpligt namn, exempelvis **HelloCMake**.



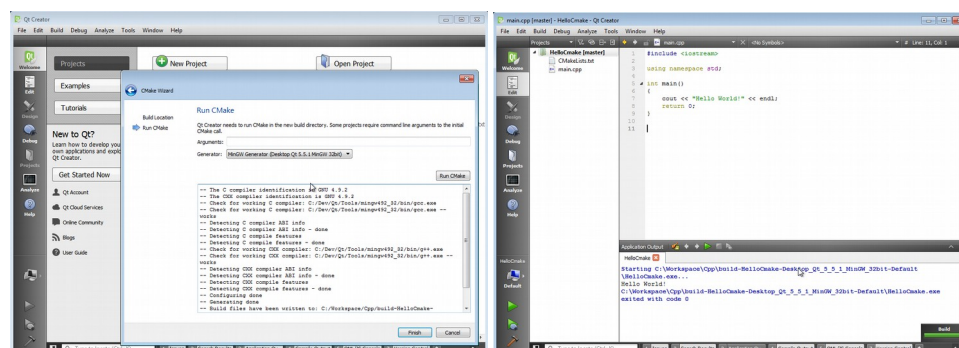
3. Välj **CMake** som byggsystem. Om den inte hittar **CMake** så kommer du behöva lägga till sökväg till den exekverbara filen (**cmake**, **cmake.exe**).
4. Välj ett **Kit** (kompilator och debugger). Om det inte finns några **Kit** tillgängliga så måste du skapa ett och då lägga till sökväg till kompilator och debugger.



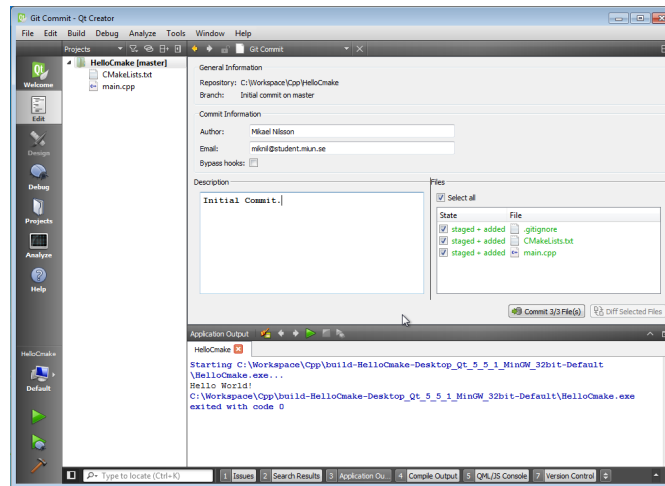
- Välj **Git** under **Add version control**. Om Git inte finns att välja i listan så trycker du på **Configure** och lägger till katalogen **bin** som ligger i installationskatalogen för Git.
- Välj en katalog där programmet skall byggas, men observera att denna måste vara avskild från den katalog som innehåller källkoden!



- Välj **Run CMake** och **Finish** för att bygga projektet!
- Kompilera och kör programmet. Nu bör du se texten **Hello World!** i terminalfönstret!



- Qt Creator har skapat ett Git-repository med en **.gitignore**-fil, däremot är filerna inte sparade i någon **branch**. Kör kommandot **git commit** från Qt Creator genom **Tools → Git → Local Repository → Commit**. Markera de filer du vill lägga till samt skriv ett meddelande och klicka på knappen **Commit** så kommer versionskontrollprompten upp och **Gitkommandot** exekveras.



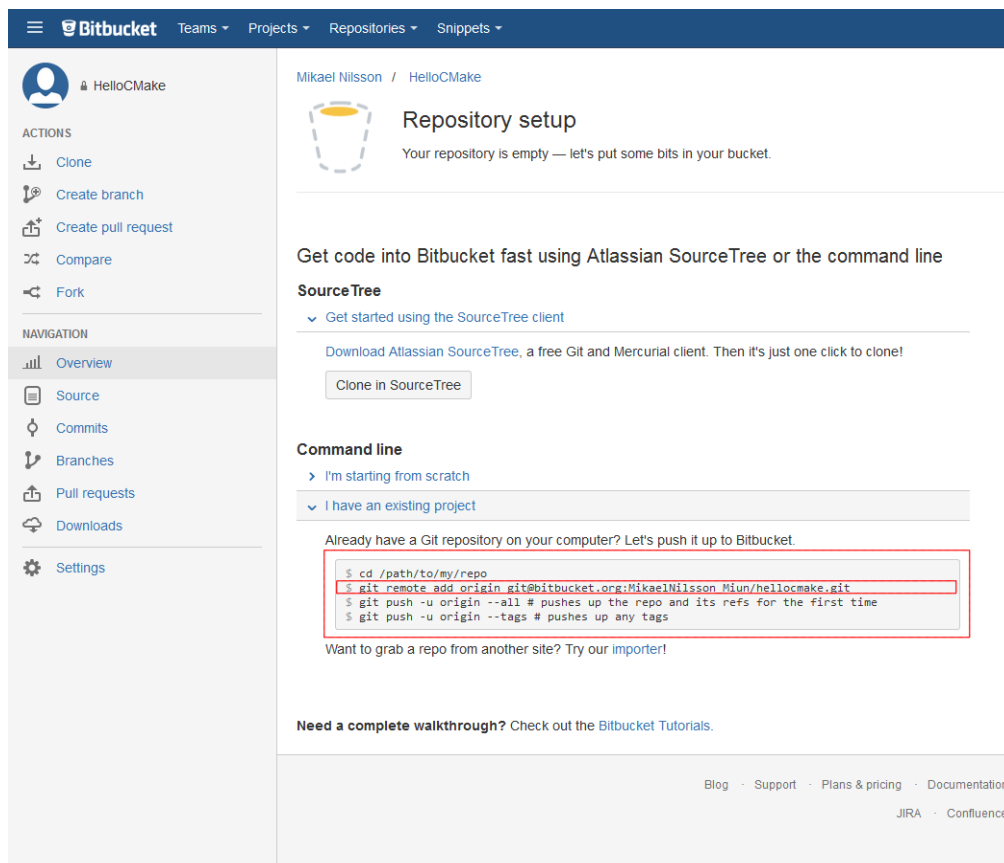
Många Git-kommandon fungerar direkt i Qt Creator och finns under menyn **Tools → Git**

3 Testprojekt

Källkoden för detta avsnitt finns tillgängligt på Bitbucket och ni kan kлона det med följande kommando;

```
$ git clone git@bitbucket.org:miun_dt042g/hellocmake.git
```

Nu är det hög tid att skapa ett nytt *repository* på *Bitbucket*. Välj **Repositories** → **Create repository** och fyll i dess namn, exempelvis **HelloCMake**.



Kopiera **remote** adressen för ditt *repository* på *Bitbucket*s sida och koppla ihop den med din nyskapade **branch** genom att i terminalen stå i projektkatalogen och skriva följande kommandon (ersätt `ssh://git@bitbucket/...` med adressen från *Bitbucket*);

```
$ git remote add origin ssh://git@bitbucket/...  
$ git push --set-upstream origin master
```

Det går även att lägga till **remote** via Qt Creator;
Tools → Git → Remote Repository → Manage Remotes

... och i CLion;
VCS → Git → Remotes...

Återvänd till din *IDE* och redigera filen **CMakeLists.txt**. Detta är skriptet som används för att bygga C++-projektet. En kommentar i *CMake* börjar med tecknet **#**. Uppdatera filen med koden nedan, förklaring för vad de olika delarna gör står i kommentarerna!

```
# Projektets namn
project(HelloCMake)

# Minsta tillåtna CMake version
cmake_minimum_required(VERSION 3.2)

# Kompilera med stöd för C++ 11
set (CMAKE_CXX_STANDARD 11)

# Lägg till katalogen med våra bibliotek till INCLUDE path
# (i dessa kataloger kommer kompilatorn att leta efter de
# header-filer som inkluderas i koden)
include_directories($ENV{TOOLS_INCLUDE})

# Ta med alla källkodsfiler i den aktuella katalogen och
# lägg dem i listan SRC_LIST
aux_source_directory(. SRC_LIST)

# Skapa en exekverbar fil med källkodsfilerna från
# SRC_LIST. Första parameteren är namnet på målet (Target)
add_executable(${PROJECT_NAME} ${SRC_LIST})
```

Varje gång man uppdaterat **CMakeLists.txt** måste man bygga om projektet. Detta gör man i Qt Creator genom; **Build → Run CMake** och sedan **Run CMake** och **Finish**.

... och i CLion; **Tools → CMake → Reload CMake Project**

Nu är det bara att börja skriva programmet. Exempel på ett simpelt testprogram för att testa **Memstat**;

```
#include <memstat.hpp>
#include <iostream>
using namespace std;

int main() {
    cout << "Hello CMake!" << endl;
    int *tal = new int;
    int *array = new int[10];
    delete array;
    return 0;
    delete tal;
    return 0;
}
```

Denna exempelkod demonstrerar på ett enkelt sätt hur man använder **Memstat** för att finna olika typer av minnesläckor. Allt man behöver är att lägga till raden `#include <memstat.hpp>` längst upp i filen som innehåller main-funktionen. Om filen inte hittas kan det bero på att miljövariabeln `TOOLS_INCLUDE` inte är skapad. Vill du inte sätta det som en global miljövariabel så kan man i Qt Creator göra så här; Under **Projects** klickar du på **Build** på det **Kit** som är aktivt. Skrolla sedan ned till **Build Environment** och expandera **Details** där du kan lägga till variabler som enbart blir kända för Qt Creator. Värt att notera är att om man gör så här kan man inte bygga via terminalen!

Kompilera och kör programmet. Först skrivs **Hello CMake!** ut och därefter allokeras minne för ett heltal samt en heltalsarray. Sedan frigör den minnet för arrayen genom att använda den felaktiga **delete**-operatoren och sedan avslutas programmet i förtid (resterande kommandon kommer alltså aldrig att exekveras).

I utskriften ser man att det allokerats minne två gånger som aldrig frigörs, samt att ett försök med icke matchande **new** och **delete** operator har använts en gång. Denna utskrift syns efter att programmet avslutats och allt ni behöver göra är att inkludera *header*-filen. Hur **Memstat** fungerar behöver ni inte sätta er in i eller känna till i den här kursen, men kortfattat fungerar det så att den överlagrar de globala operatorerna **new** och **delete** samt håller koll på allt som allokeras och avallokeras med dessa.

3.1 Debug / Release Mode

Memstat kräver mer minne än ett program som inte använder det och det är därför ingen mening att använda det annat än i debuggningssyfte. Vi skall därför sätta upp så att *Memstat* endast körs då vi kör en **debug build** och inte när vi gör en **release**. Detta gör vi genom att modifiera `CMakeLists.txt` så här (ändringar visas i blå färg);

```
# Projektets namn
project(HelloCMake)

# Minsta tillåtna CMake version
cmake_minimum_required(VERSION 3.2)

# Kompilera med stöd för C++ 11
set (CMAKE_CXX_STANDARD 11)

# Lägg till katalogen med våra bibliotek till INCLUDE path
# (i dessa kataloger kommer kompilatorn att leta efter de
# header-filer som inkluderas i koden)
include_directories($ENV{TOOLS_INCLUDE})

# Ta med alla källkodsfiler i den aktuella katalogen och
# lägg dem i listan SRC_LIST
aux_source_directory(. SRC_LIST)

# Om Debug mode
if (CMAKE_BUILD_TYPE STREQUAL "Debug")
    # skriv ut meddelandet Debug mode samt sätt kompilatordirektivet
    # DEBUG
    message("Debug mode")
    add_definitions(-DDEBUG)
else(CMAKE_BUILD_TYPE STREQUAL "Debug")
    # skriv ut meddelandet Release mode
    message("Release mode")
endif(CMAKE_BUILD_TYPE STREQUAL "Debug")

# Skapa en exekverbar fil med källkodsfilerna från
# SRC_LIST. Första parameteren är namnet på målet (Target)
add_executable(${PROJECT_NAME} ${SRC_LIST})
```

Här har vi lagt till en **IF**-sats som gör en strängjämförelse (**STREQUAL**) för att se om innehållet i variabeln **CMAKE_BUILD_TYPE** är lika med strängen **"Debug"**. Den skriver sedan ut ett meddelande och sätter **kompilatordirektivet** **DEBUG**. Om strängarna inte är lika anses det istället vara **Release Mode**. Utskriften är till för att man skall få feedback från CMake vilket typ av bygge den utfört, vilket är bra i felsökningssyfte. Nu kan vi med hjälp av **kompilatordirektivet** ta med **Memstat** under tiden för utveckling och stänga av det när vi bygger för produktion.

Viktigt att poängtera är att variabeln **CMAKE_BUILD_TYPE** används internt av CMake och om ni använder Debuggern i Qt Creator måste den vara satt till **Debug** annars finns det ingen debuginformation inkluderat i den körbara filen och det går inte att sätta **breakpoints**!

Öppna **main.cpp** och ändra inledningen till följande;

```
#if DEBUG
#include <memstat.hpp>
#endif
```

Detta kontrollerar om *kompiletordirektivet* **DEBUG** är angivet och inkluderar isåfall *Memstat*. Innan ni kan exekvera detta måste ni köra **Run CMake** eftersom vi gjort ändringar i **CMakeLists.txt**. När vi kör **CMake** kommer texten **Release mode** att skrivas ut och om vi skulle kompilera kommer inte *Memstat* att köras.

*Liknande **preprocessing directives** har du säkert använt för andra syften, som exempelvis **makrodefinitioner**, och kan placeras vart som helst i koden. Till exempel så kan en funktion innehålla följande kodsnitt som skriver ut det mode som används;*

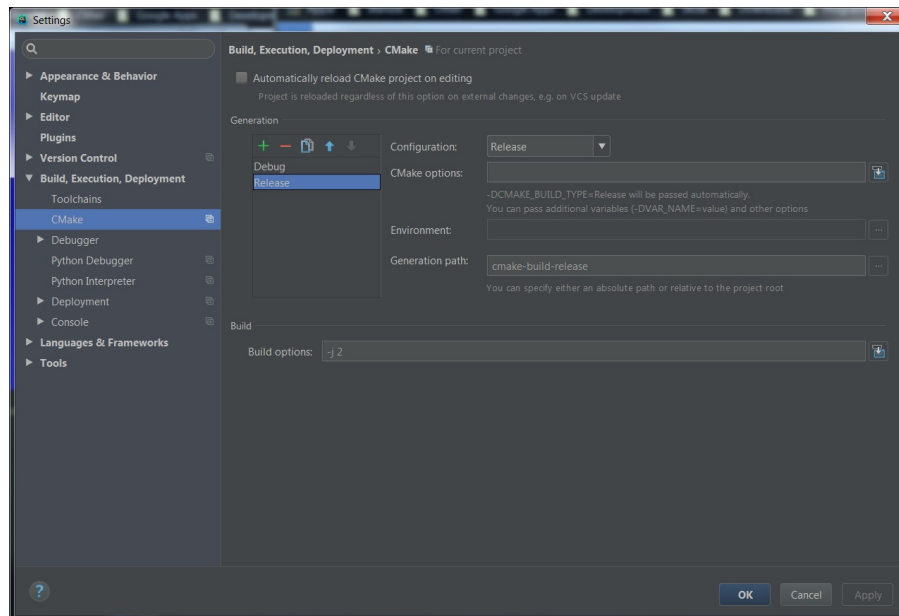
```
#if DEBUG  
    cout << "Debug mode" << endl;  
#else  
    cout << "Release mode" << endl;  
#endif
```

3.1.1 CLion

Du sätter upp konfigurationer för **Debug** / **Release** mode i **CLion** genom att gå till;

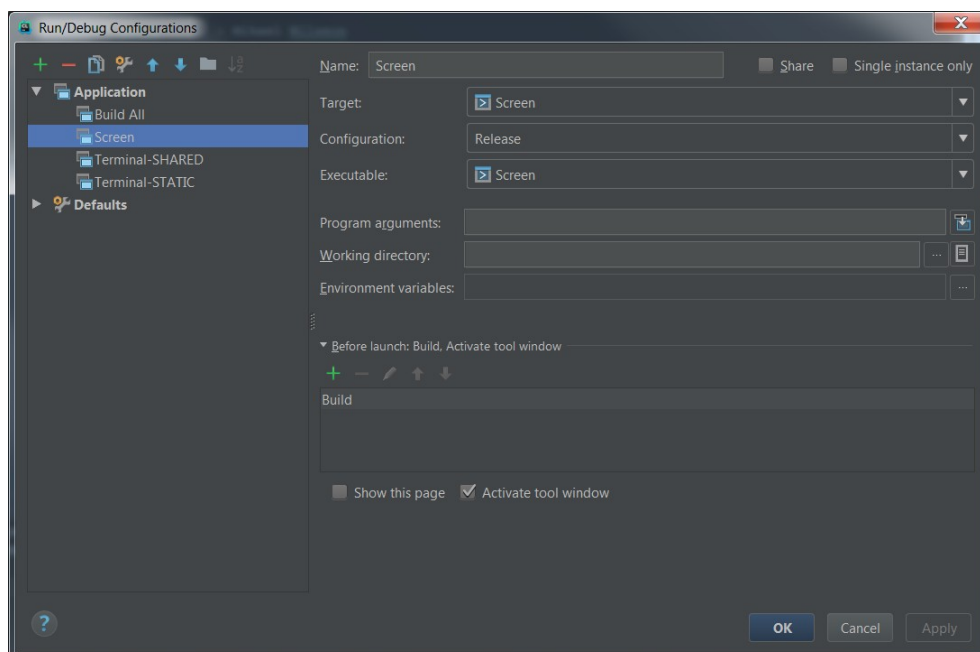
File → Settings → Build, Execution, Deployment → CMake

Här kommer en konfiguration för **Debug** redan finnas, men du kan enkelt lägga till **Release** via plustecknet!



Du kan sedan skifta mellan dessa konfigurationer genom;

Run → Edit Configurations...

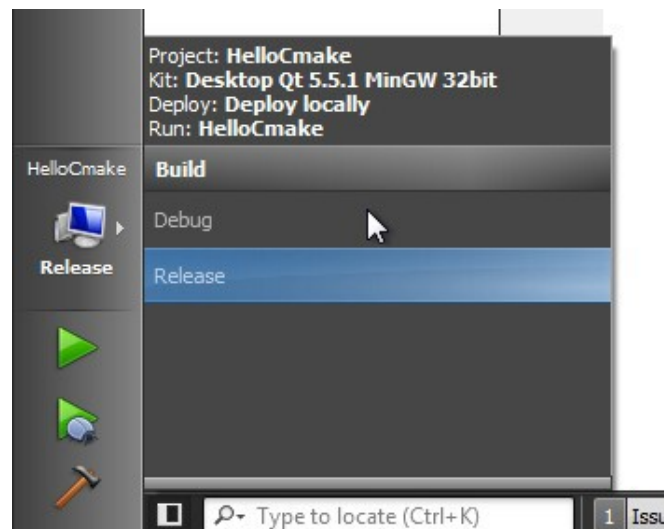


3.1.2 Qt Creator

På vissa operativsystem sätts *release* och *debug* automatiskt upp när man skapar ett *CMake*-projekt i *Qt Creator*, men om det inte gör det måste du göra följande;

1. **Projects** → **Build** och under **Build Settings** väljer du **Add** → **Build**, samt namnger bygget **Debug**.
2. Välj en katalog där projektet skall byggas, dock avskilt från projektkatalogen.
3. I fältet **Arguments** skriver du; **-DCMAKE_BUILD_TYPE=Debug**
4. Kör **Run CMake** (nu skall **Debug mode** skrivas ut) och tryck på **Finish**
5. Upprepa steg 1-4, men ersätt **Debug** med **Release**.

När detta är uppsatt kan ni byta mellan *debug* och *release* i byggmenyn på följande sätt;



Observera att om ni valt flera Kit så kommer man även se dem i denna meny. **Debug** och **Release** skall även byggas i olika kataloger, som båda är avskilda från såväl källkodskatalogen och varandra!

4 Git Submodule

Den slutliga källkoden för detta avsnitt finns i **branchen** som heter **finished** i repositoryet för [Timer](#) [6] på Bitbucket!

För att ytterligare minska på kodduplicering kan man använda [submodules](#) [4] i *Git* som i grund och botten är en pekare från ett *repository* till ett annat. Fördelen med att ha det så är att man endast behöver göra ändringar på ett ställe.

Istället för att skapa ett nytt projekt som vi gjorde tidigare, skall vi nu importera ett befintligt [CMake-projekt](#) från *Bitbucket*, och kan ofta utföras direkt i din *IDE*!

URL till repot är git@bitbucket.org:miun_dt042g/timer.git

För att importera detta i *CLion*;

VCS → Checkout from Version Control → Git

...och ange URL till repot samt en lokal destinationsmapp.

I *Qt Creator* skapar man först ett nytt projekt och väljer därefter;

Import Project → Git Clone

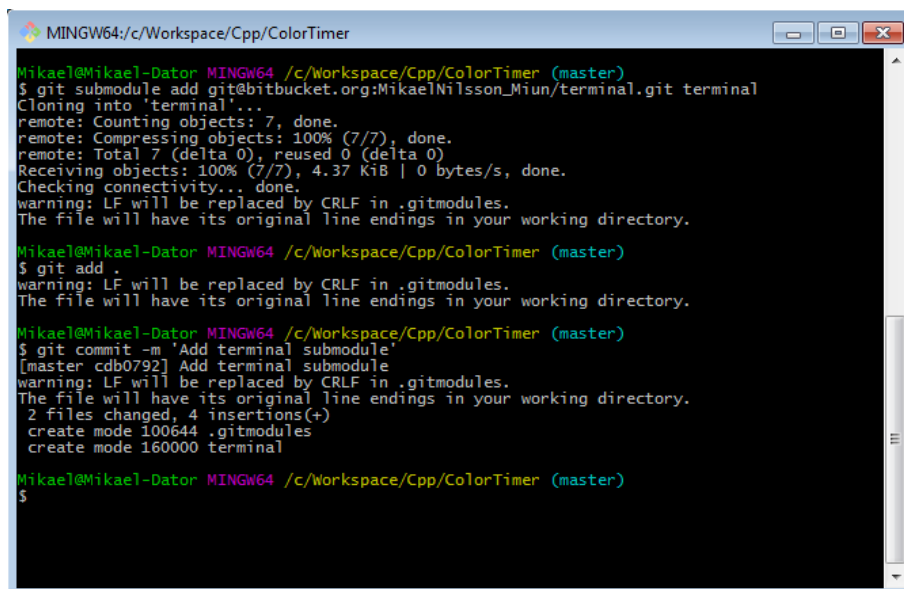
...och ange URL till repot, värdet **master** för **branch** samt en lokal destinationsmapp.

Om du använder *Qt Creator* måste du även skapa en byggkatalog för *CMake*, avskild från projektkatalogen!

Nu är det bara att bygga samt testa projektet (vad den gör är att skapa en slumpmässig lista och mäta tiden det tar att sortera den).

För att lägga till **Terminal** som en *submodule* går du in i projektkatalogen med en terminal / *Git Bash* och skriv följande kommandon;

```
$ git submodule add git@bitbucket.org:miun_dt042g/terminal.git terminal
$ git add .
$ git commit -m "Add terminal submodule"
```



```
Mikael@Mikael-Dator MINGW64 /c/Workspace/Cpp/ColorTimer (master)
$ git submodule add git@bitbucket.org:MikaelNilsson_Miun/terminal.git terminal
Cloning into 'terminal'...
remote: Counting objects: 7, done.
remote: Compressing objects: 100% (7/7), done.
remote: Total 7 (delta 0), reused 0 (delta 0)
Receiving objects: 100% (7/7), 4.37 KiB | 0 bytes/s, done.
Checking connectivity... done.
warning: LF will be replaced by CRLF in .gitmodules.
The file will have its original line endings in your working directory.

Mikael@Mikael-Dator MINGW64 /c/Workspace/Cpp/ColorTimer (master)
$ git add .
warning: LF will be replaced by CRLF in .gitmodules.
The file will have its original line endings in your working directory.

Mikael@Mikael-Dator MINGW64 /c/Workspace/Cpp/ColorTimer (master)
$ git commit -m 'Add terminal submodule'
[master cdb0792] Add terminal submodule
warning: LF will be replaced by CRLF in .gitmodules.
The file will have its original line endings in your working directory.
2 files changed, 4 insertions(+)
create mode 100644 .gitmodules
create mode 160000 terminal

Mikael@Mikael-Dator MINGW64 /c/Workspace/Cpp/ColorTimer (master)
$
```

För att använda *Terminal* måste vi lägga till det i **CMakeLists.txt** (i projektet). Lägg till katalogen som ett *subdirectory*;

```
# Add terminal subdirectory
add_subdirectory(terminal)
```

Terminal har sin egen **CMakeLists.txt** som bygger två versioner av biblioteket; en för **statisk länkning** (**.a**, **.lib**) och en för **dynamisk länkning** (**.so**, **.dylib**, **.dll**). Den statiska versionen heter **Terminal** och vi länkar den till projektet med kommandot **target_link_libraries**. Lägg till raden nedan efter anropet till **add_executable**;

```
# Static link Terminal with executable ${PROJECT_NAME}
target_link_libraries(${PROJECT_NAME} Terminal)
```

Vill man istället länka *dynamiskt* ändrar man bara namnet på biblioteket till **Terminal-SHARED**;

```
target_link_libraries(${PROJECT_NAME} Terminal-SHARED)
```

För att kunna köra programmet måste man kopiera (**Terminal.so**, **.dll**, **.dylib**) till samma katalog som den exekverbara filen **Timer** ligger. Filen **Terminal** byggs i en underkatalog i byggprojektkatalogen.

För information kring hur **Terminal** fungerar se **README.md** i dess [repository](#) [5] på Bitbucket!

För att testa och använda modulen i *timer*-projektet kan du ändra **main.cpp** så att det skriver ut sorteringstiden med grön text;

```
#ifdef DEBUG
#include <memstat.hpp>
#endif

#include <iostream>
#include <vector>
#include <algorithm>
#include <cstdlib>
#include "timer.hpp"
#include "terminal.h"
using namespace std;

int main() {
    // Create randomized vector with 1000000 integers
    srand(time(nullptr));
    vector<int> numbers;
    numbers.resize(1000000);
    generate(numbers.begin(), numbers.end(), rand);

    // Create timer and measure sorting time
    Timer timer;
    timer.start();
    sort(numbers.begin(), numbers.end());
    auto nanosec = timer.stop<chrono::nanoseconds>();
    Terminal term;
    auto green = term.strColor(TerminalColor(COLOR::GREEN));
    cout << "Sorting 1000000 random numbers with sort took "
         << green(to_string(nanosec)) << " nanoseconds!" << endl;
    return 0;
}
```

Sedan är det bara att kompilera och köra och då skall tiden skrivas ut med grön färg. Om koden körs i **Application Output** i *Qt Creator* kommer förmodligen ingen färg att skrivas ut. För att köra en applikation i en standard-terminal gå till **Projects** och på det aktuella **Kit** väljer du **Run**. Under **Run** kryssar du i **Run in terminal**. För att välja vilket terminalprogram som skall användas går du in i **Tools → Options → System** (i äldre versioner av *Qt Creator* heter det **Environment**).

I laboration 2 får du arbeta vidare med några av de teman som tagits upp i denna lektion!

Referenser

- [1] CMake, "CMake download", 2017. [Online]
Tillgänglig: <https://cmake.org/download/>
[Åtkomst: 20 Augusti 2017]
- [2] Qt, "Download Qt Open Source", 2017. [Online]
Tillgänglig: <https://www.qt.io/download-open-source/#section-2>
[Åtkomst: 20 Augusti 2017]
- [3] GitHub, "Catch", 2017. [Online]
Tillgänglig: <https://github.com/philsquared/Catch>
[Åtkomst: 20 Augusti 2017]
- [4] Atlassian Blog, "Git submodules", 2017. [Online]
Tillgänglig: <https://www.atlassian.com/blog/archives/git-submodules>
[Åtkomst: 20 Augusti 2017]
- [5] Bitbucket, Mikael Nilsson / Terminal, 2016. [Online]
Tillgänglig: https://bitbucket.org/miun_dt042g/terminal
[Åtkomst: 20 Augusti 2017]
- [6] Bitbucket, Mikael Nilsson / Timer, 2016. [Online]
Tillgänglig: https://bitbucket.org/miun_dt042g/timer
[Åtkomst: 20 Augusti 2017]
- [7] JetBrains, "Free for students", 2017. [Online]
Tillgänglig: <https://www.jetbrains.com/student/>
[Åtkomst: 03 September 2017]
- [8] JetBrains, "Toolbox App", 2017. [Online]
Tillgänglig: <https://www.jetbrains.com/toolbox/app/>
[Åtkomst: 03 September 2017]
- [9] mingw-w64, "GCC for Windows 32 & 64 bits", 2017. [Online]
Tillgänglig: <http://mingw-w64.org/doku.php/download#mingw-builds>
[Åtkomst: 03 September 2017]
- [10] The GNU Project Debugger, 2017. [Online]
Tillgänglig: <https://www.gnu.org/software/gdb/>
[Åtkomst: 03 September 2017]

Bilaga A: Cmake Referenser

Användbara CMake-kommandon för denna kurs

[project\(\)](#)
[include_directories\(\)](#)
[if\(\), else\(\), endif\(\)](#)

[set\(\)](#)
[add_subdirectory\(\)](#)
[add_executable\(\)](#)

[cmake_minimum_required\(\)](#)
[aux_source_directory\(\)](#)
[target_link_libraries\(\)](#)

Fler användbara kommandon och variabler i CMake finner du i denna källa!

För att generera projekt för andra IDE's än **Qt Creator** och **Clion** så skapar du en ny katalog och skriver kommandot;

```
cmake -G [generator] [path to project directory]
```

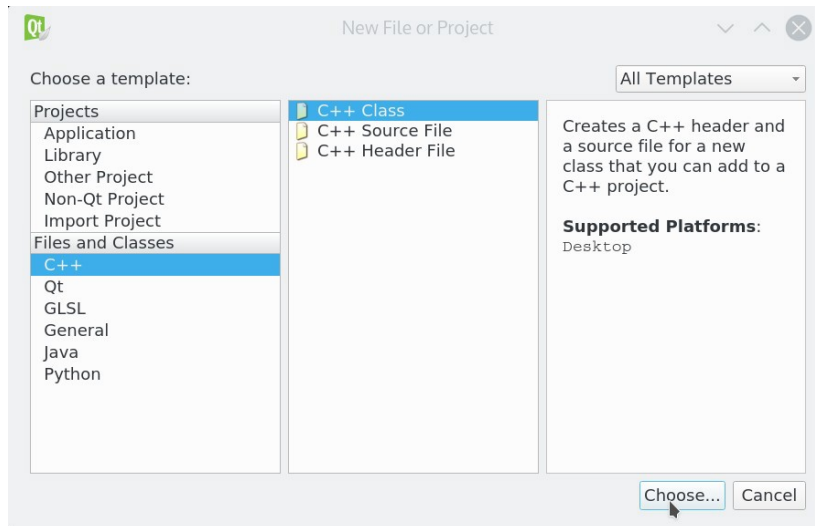
En lista över Generatorer som går att använda finner du [här](#)!

För att exempelvis generera ett **Visual Studio 2013** projekt, skriver du;

```
cmake -G "Visual Studio 12 2013" ../Timer
```

Bilaga B: Skapa klasser & filer med Qt Creator

Som du kanske har märkt går det inte att skapa och lägga till filer via menyn i *Qt Creator*. Detta beror på att *Qt Creator* inte redigerar *CMake*-skriptet utan det är upp till utvecklaren att själv göra. För att lägga till en klass / källkodsfil välj **New File or Project** under menyn för **File**. Välj sedan **C++** under **Files and Classes** och den typ av fil du vill lägga till.



Därefter startas en **Template** som hjälper till att skapa filen eller klassen. Viktigt är att den sparas i samma katalog som *CMake* letar efter filer (med kommandot `aux_source_directory`).

Observera att efter man skapat filerna så syns de inte i *Qt Creator*, utan man måste först köra **Run CMake**!