LINKÖPINGS UNIVERSITET

TDP005 Projekt: Objektorienterat system

Introduktion till CLion

Författare

Filip Strömbäck

INSTITUTIONEN FÖR DATAVETENSKAP

Höstterminen 2016
Version 1.0

2016-10-04

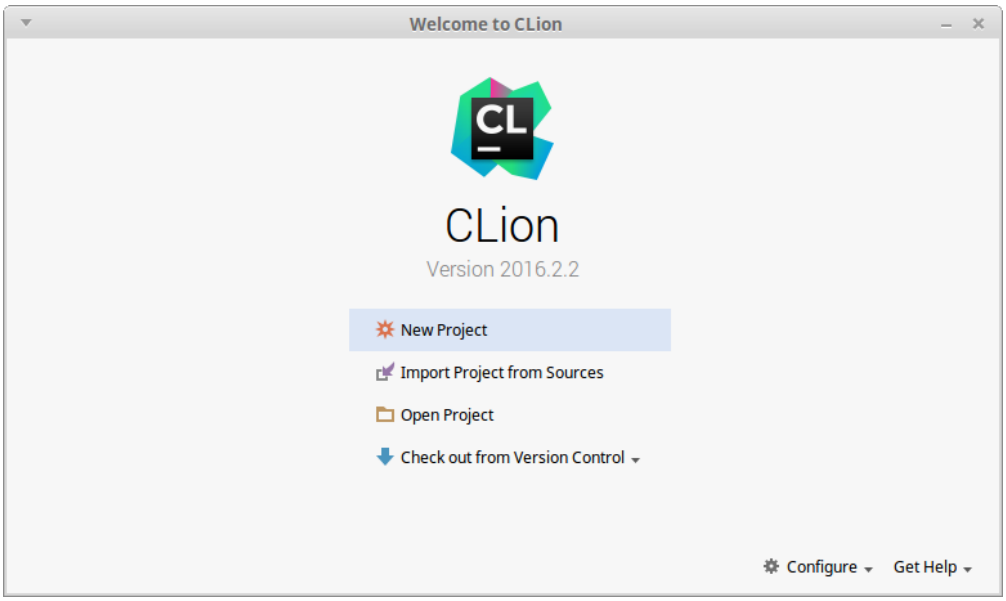
Introduktion

I många projekt är det fullt tillräckligt att använda en vanlig texteditor för att skriva kod. Det finns olika editorer ämnade för att skriva programkod och alla hjälper till olika mycket. Många enklare verktyg hjälper till med att färga olika nyckelord i texten så att det blir enklare att läsa koden. Många hjälper också till att formattera koden i viss mån. Allt detta gör det smidigare att läsa och redigera koden, men det kan fortfarande vara svårt att navigera i stora kodmängder med dessa enklare verktyg. De verktyg som är steget kraftfullare än dessa texteditors brukar kallas för *Integrated Development Environment* eller *IDE*. En IDE gör allt som en texteditor gör, men förstår den kod som är skriven och låter programmeraren utforska och förändra koden på en högre nivå än den text som är skriven. På grund av denna förståelse kan en IDE med stor säkerhet svara på frågor som *Vilka klasser ärver från denna klass?* och *Var i koden används denna variabel?* Det är ofta också möjligt att refaktorera sin kod enkelt med hjälp av en IDE. Exempelvis är det enkelt att byta namn på klasser eller variabler utan att själv behöva hitta alla ställen där klassen eller variabeln används.

I denna lab kommer vi titta närmare på CLion, som är en IDE för C++. Vi kommer se hur CLion kompilerar vår kod åt oss och hur den kan hjälpa oss att läsa och ändra vår kod.

1 Installation

Som student kan du få en gratis licens för CLion som är giltig i ett år. När den tiden gått ut kan du upprepa nedanstående process igen för att få en ny licens så länge du är student. För att få licensen, gå till `https://www.jetbrains.com/clion/buy/#edition=discounts`, välj *Special Offers*, sedan *For students and teachers*. På nästa sida, välj sedan *Apply Now* och ange din studentmailadress. Följ stegen i de mail som skickas till dig för att skapa ett konto och ladda ner CLion.



Fönstret *Welcome to Clion*

När du kommer till nedladdningssidan (länken gömmer sig i det övre högra hörnet), välj *Linux* och klicka på *Download* så får du hem en `tar.gz`-fil (**Notera:** Om ni använder Chrome eller Chromium kan den göra något konstigt med `tar`-filen så att kommande steg inte fungerar). Spara den någonstans i din hemkatalog, öppna en terminal och `cd`:a till mappen där din `tar`-fil finns (det kanske är en bra idé att flytta den bort från Downloads-mappen först). Extrahera filen med kommandot `tar -zxvf CLion-2016.?.?.tar.gz`. Detta skapar mappen `clion-2016.?.?`. Kör `cd clion-2016.?.?/bin` och kör `./clion.sh`. När ni startar

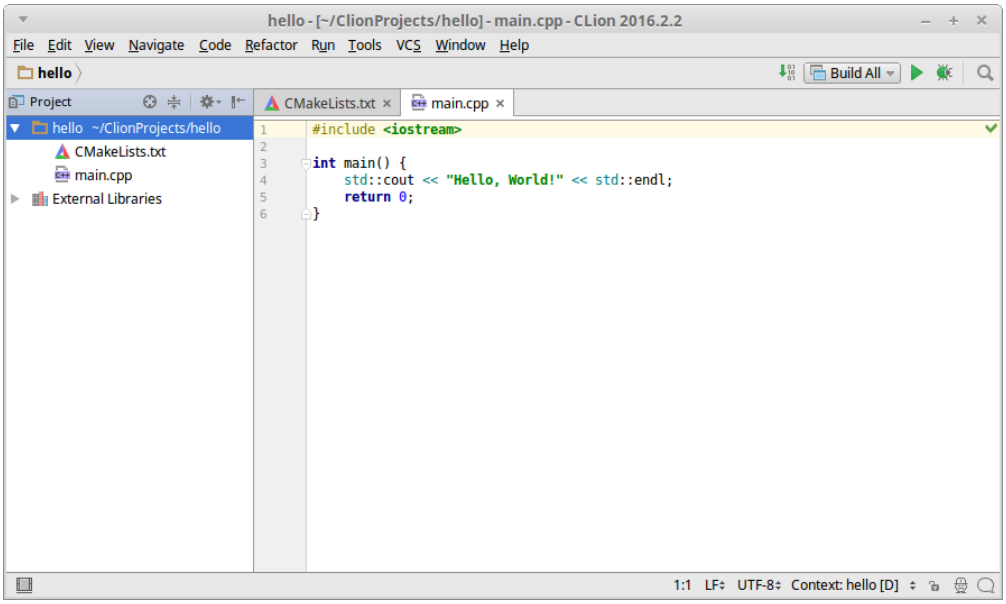
CLion första gången kommer den först fråga om ni vill importera inställningar från en tidigare CLion-installation. Välj *I do not...* och klicka på *OK*. Efter detta kommer CLion be er logga in med det konto ni skapade tidigare i registreringsprocessen för att kunna verifiera er licens. Sen kommer CLion fråga vilket tema den ska använda, vilka verktyg den ska använda, och om ni vill installera extra plugin. Standardalternativen fungerar utmärkt i alla stegen.

När ni ser fönstret *Welcome to CLion* är ni klara. Om ni gillar de tangentbordsgenvägar som Emacs använder, kan ni gå till *Customize* → *Settings* → *Keymap* och sedan välja *Emacs* i listan märkt *Keymaps*. Detta går självklart att göra senare också.

2 Projekt

För att kompilera och redigera kod i CLion krävs att ni skapar ett projekt. Ett projekt är en samling källkodsfiler som hör ihop tillsammans med instruktioner om hur de ska kompileras till en eller flera körbara filer. Allt detta kan CLion använda för att förstå vårt program och därmed hjälpa till när vi utvecklar.

Skapa ett projekt genom att välja *New Project* och ange ett namn i fönstret *New CMake Project* (**Notera:** Ni får problem om ni använder `test` som namn). När ni klickar *OK* öppnas projektet, och ni borde se något liknande detta:



Till vänster visas alla filer som projektet innehåller, just nu bara `main.cpp` och `CMakeLists.txt`, och i resterande delen av fönstret finns en editor där man kan se och redigera filer. Vi kan se att CLion genererade ett *Hello World*-program åt oss. Vi kan köra det genom att välja *Run* → *Run 'Build All'*, eller genom att klicka på den gröna pilen i övre högra hörnet (om den är grå så gör CLion någonting i bakgrunden. Vänta då ett litet tag så blir den grön).

Första gången vet inte CLion vad den körbara filen ska heta, så den visar fönstret *Edit Configuration* och vill att ni ska fylla i fältet *Executable*. Välj det alternativ som är markerat med en ikon och som har samma namn som ert projekt (dvs. det som inte är *Not selected* eller *Select other*). Klicka sedan på *Run*, så kompilerar och kör CLion ert program.

Filen `CMakeLists.txt` innehåller instruktioner om hur CLion ska bygga programmet. I de flesta fall behöver man inte redigera den manuellt, CLion löser det. Ska ni däremot använda externa bibliotek kan ni behöva

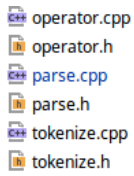
lägga till dem här. Titta på dokumentationen för de bibliotek ni vill använda för hur det görs. Vi kommer titta lite närmare på CMake och Make i en senare lab.

3 Versionshantering

CLion har inbyggt stöd för Git och diverse andra versionhanteringssystem. Det ska vi utnyttja för att hämta den kod vi ska arbeta med i den här labben. Vi vill hämta hem `https://gitlab.liu.se/filst04/tdp005-lab.git`. Gå tillbaka till fönstret *Welcome to CLion* genom att välja *File* → *Close Project*. Hämta sedan koden genom att välja *Check out from Version Control* → *Git*. Ange URL och klicka på *Clone*. Välj *Yes* när den är klar och frågar om ni vill öppna mappen ni hämtade.

Om ni versionshanterar ett projekt där CLion används är det en bra idé att se till att de projektfiler CLion sparar i mappen `.idea` inte kommer med i Git. CLion sparar användarspecifika saker där, så när flera personer samarbetar kommer det sannolikt bli konflikter i filerna där om de inte ignoreras. Detta görs lämpligtvis genom att lägga till raden `.idea/` i filen `.gitignore` om den inte redan finns där.

När vi arbetar med ett versionshanterat projekt kommer vi se att de filer som vi har ändrat sedan senaste commit blir blåa i listan med filer (se bilden till höger). I texteditorn markeras också ändringarna i vänstermarginalen. För att göra en commit, välj *VCS* → `Commit Changes...` Då visas ett fönster där man kan välja vilka filer som ska committas, och där man kan ange ett commitmeddelande. Härifrån kan man också välja att direkt ladda upp ändringarna till exempelvis GitLab genom att välja *Commit and Push*. Det kommer dock inte fungera med den delade koden eftersom ni inte har tillåtelse att skriva dit.



4 Refaktorering

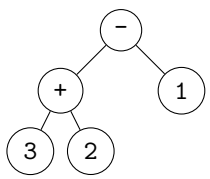
En av de stora fördelarna med att använda en IDE (exempelvis CLion) är den hjälp man kan få med att navigera och utforska kod. Dessutom kan många IDE:er också hjälpa till med att formattera och refaktorera kod mer eller mindre automatiskt.

Den kod vi nyss hämtade från Git är en miniräknare. Där kan man mata in uttryck som i C/C++ och så räknar programmet ut värdet av uttrycket. Man kan exempelvis mata in `2 * (1 + 3) / 3` så svarar den med hur den tolkar uttrycket och värdet, `((2 * (1 + 3)) / 3) = 2.66667` i detta fall. Kompilera och kör programmet som i steg 2 och testa det för att se hur det fungerar.

Notera: uppgifterna nedan är till för att utforska mer avancerade men smidiga funktioner i CLion, och är därför valfria. För att illustrera meningen med denna funktionalitet behöver man ha en lite större mängd kod att arbeta med. Det är därför miniräknaren används som exempel. Det är alltså inte något krav på att förstå eller att göra ändringar i miniräknaren.

4.1 Kort introduktion

I filen `main.cpp` finns programmets huvudloop. Den läser in en rad text och försöker att bygga ett syntaxträd av uttrycket (med hjälp av `parse`-funktionen). Om det går bra så returnerar `parse` en pekare till rotnoden i trädet som representerar uttrycket. Alla noder i trädet representeras av en subklass till `Expression`. Just nu finns klasserna `Constant` och `BinaryOp` som representerar tal (literaler) och binära operatorer. Binära operatorer innehåller i sin tur två noder för de uttryck som motsvarar operatorns vänsterled och högerled. Exempelvis motsvaras uttrycket `(3 + 2) - 1` av trädet till höger.



Träd för `(3 + 2) - 1`

När vi har skapat ett träd av uttrycket kan vi sedan enkelt räkna ut värdet av uttrycket genom att be rotnoden att beräkna sitt värde. I exemplet ovan kommer rotnoden vara en binär operator, så den beräknar sitt värde genom att be vänsterledet och högerledet att beräkna sina värden och sedan beräknar den skillnaden mellan dessa. Detta sköts av medlemsfunktionen `evaluate`.

Målet med dessa extrauppgifter är att lägga till stöd för konstanter till miniräknaren. Vi vill alltså kunna skriva exempelvis `2 * pi` och få ett vettigt resultat.

4.2 Navigation

Innan man kan göra några ändringar i ett projekt måste man förstå koden som redan finns. Här kan CLion hjälpa till genom att låta oss snabbt hoppa till definitioner, visa vilka subklasser som finns och liknande. Om vi utgår från filen `main.cpp` kan vi exempelvis snabbt hitta var klassen `Expression` är deklarerad genom att högerklicka på ordet `Expression` (på rad 18) och välja *Go To* → *Declaration* (Ctrl+B).

Vi kan också se efter vilka klasser som ärver från `Expression` genom att gå till `expression.h` och klicka på den lilla blå bollen i marginalen. Detta fungerar även för att se vilka funktioner som skriver över funktioner i en basklass.

Vi kan också hitta alla ställen i koden som använder en viss klass eller funktion att högerklicka på den och välja *Find Usages...*. Då öppnas ny panel öppnas i botten av CLion som bland annat innehåller *Found usages*. Där finns alla referenser till funktionen eller klassen listade baserat på hur de används.

4.3 Byta namn på saker

I miniräknaren så har klassen `Constant` (se `constant.h`) fått fel namn. Egentligen borde den heta `Literal`. För att byta namn på klassen måste vi gå igenom all vår kod för att hitta alla ställen där `Constant` används och ändra namnet där också. Det blir snabbt väldigt jobbigt i ett stort projekt. Till viss del går det att ta hjälp av kompilatorn för att göra detta, men det blir snabbt tidskrävande ändå. Vi kan dock låta CLion göra jobbet åt oss! Gå till klassen `Constant`, högerklicka på `Constant` och välj *Refactor* → *Rename...*. CLion markerar då texten `Constant`. Ändra det till `Literal` och tryck på *enter* för att genomföra namnbytet i hela projektet. I detta fall kommer CLion fråga om den ska ändra `Constant` till `Literal` i kommentarer också. Klicka på *Show Usages* för att se vart den vill ändra (bekräfta med *Do Refactor*), eller välj helt enkelt *Rename All Usages* för att lita blint på CLion.

Vi vill också byta namn på filerna `constant.h` och `constant.cpp` till `literal.h` och `literal.cpp`. Skulle vi göra detta manuellt skulle vi behöva hitta alla ställen där `constant.h` inkluderas och uppdatera dem. Högerklickar vi i stället på `constant.h` i fillistan och väljer *Refactor* → *Rename...* och anger `literal.h` som nytt namn kommer CLion lösa allt åt oss. Den döper till och med om `.cpp`-filen också.

Testa nu att kompilera och köra programmet för att se till så att allt fortfarande fungerar som det ska!

4.4 Lägga till konstanter

Det första vi vill göra för att lägga till stöd för konstanter i miniräknaren är att skapa en ny subklass till `Expression` för att representera konstanter i ett uttryck. Gör det genom att högerklicka på någon fil i fillistan och välja *New* → *C++ class*. Ange `Constant` som namn och välj sedan *OK*. CLion skapar då filerna `Constant.h` och `Constant.cpp` innehållande den nya klassen.

Se sedan till så att `Constant` ärver från `Expression` genom att ändra klassdeklarationen till följande (se också till att inkludera `expression.h`. CLion föreslår detta ibland):

```
class Constant : public Expression {
};
```

När vi gjort det så kan vi be CLion att lägga till de funktioner vi vill skriva över från basklassen. Gör detta genom att högerklicka på klassen och välja *Generate...* → *Override Functions...*. Markera konstruktor, `evaluate` och `output` i fönstret som visas och klicka sedan *OK*. Om allt gått bra så ska CLion ha genererat tre funktioner åt oss. CLion gör tyvärr fel när den genererar konstruktorn i detta fall. I headerfilen lägger den till `Constant() = default;` när det bara ska vara `Constant();`. Fixa detta.

Nästa steg är att lägga till en strängvariabel inuti `Constant`-klassen. I den ska vi lagra namnet på den konstant som noden representerar. Lägg också till en strängparameter till konstruktorn och initiera medlemsvariabeln i konstruktorn. Implementera sedan `output`-funktionen på följande sätt.

```
void Constant::output(std::ostream &to) const {
    to << name;
}
```

För att kunna implementera `evaluate`-funktionen måste vi se till att `evaluate` får tillgång till en lista över alla konstanter som ska finnas. Vi vill lägga den i `main.cpp` för att det ska vara enkelt att hitta den. I detta fall använder vi en `std::map<std::string, double>` för att lagra konstanterna. Lägg följande deklaration globalt i `main.cpp` innan funktionen `main` (glöm inte att inkludera `map`):

```
const std::map<std::string, double> constants = {
    { "pi", 3.14159 },
    { "e",  2.71828 },
};
```

Nu vill vi se till att vi skickar med `constants` till alla `evaluate`-funktioner. Vi kan börja med att låta CLion lägga till en parameter till alla överskrivna funktioner åt oss genom att gå till `expression.h`, högerklicka på `evaluate` och välj *Refactor* → *Change Signature...*. Klicka på det gröna plustecknet i högerkanten av fönstret som kommer upp så läggs det till en ny rad i listan i mitten av fönstret. Fyll i `const std::map<std::string, double> &` som typ och `constants` som namn och klicka på *Refactor*. Om allt har gått rätt så ska CLion nu ha lagt till en parameter till alla `evaluate`-funktionerna som finns. Dock kan den inte lista ut vad som ska skickas in som den nya parametern. För att hitta dessa ställen, kompilera programmet och titta på det första kompileringsfelet tills programmet kompilerar. I `main.cpp` vill vi skicka med variabeln `constants` som vi just skapade. I `operator.cpp` vill vi helt enkelt skicka vidare parametern `constants`.

Nu kan vi implementera `Constant::evaluate`. Den ska se ut så här:

```
double Constant::evaluate(const std::map<std::string, double> &constants) const {
    auto found = constants.find(name);
    if (found != constants.end())
        return found->second;
    else
        return 0;
}
```

Till slut ska vi bara se till att vår nya klass skapas någon gång. Detta ska ske på rad 56 i filen `parse.cpp`. Byt ut raden `return new Literal(0.0);` mot `return new Constant(src);`. Se till att inkludera `Constant.h`.

Nu borde kalkylatorn fungera och kunna hantera uttryck som `2*pi` och `4*e` korrekt.