Handbok KDevelop

Det här dokumentet är konverterat från KDE:s användarbas sida KDevelop4/Manual.
Översättare: Stefan Asserhäll

Innehåll

Sammanfattning

KDevelop är en integrerad utvecklingsmiljö att använda för ett brett utbud av programmeringsuppgifter.

Kapitel 1

Vad är KDevelop?

KDevelop är en modern integrerad utvecklingsmiljö för C++ (och andra språk) som är ett av många KDE program. Som sådant kör det på Linux® (även om någon av de andra skrivborden, såsom GNOME, används) men det är också tillgängligt för de flesta andra varianter av UNIX® och dessutom för Windows.

KDevelop erbjuder alla bekvämligheter i en modern utvecklingsmiljö. För stora projekt och program är den viktigaste funktionen att KDevelop *förstår* C++: det tolkar hela källkodsbasen och kommer ihåg vilka klasser som har vilka medlemsfunktioner, var variabler definieras, vad deras typer är, och många andra saker om koden. Låt oss exempelvis anta att en av projektets deklarationsfiler deklarerar klassen

```
class Car {
  // ...
  public:
    std::string get_color () const;
};
```

och senare i programmet har du

```
Car my_ride;
// ... gör något med variabeln ...
std::string color = my_ride.ge
```

kommer det ihåg att `my_ride` på sista raden är en variabel av typen `Car` och erbjuder dig att komplettera `ge` som `get_color()` eftersom det är enda medlemsfunktionen i klassen `Car` som börjar så. Istället för att fortsätta skriva, tryck bara på returtangenten för att få hela ordet. Det sparar tid, undviker stavfel, och kräver inte att man kommer ihåg de exakta namnen på de hundratals eller tusentals funktioner och klasser som ingår i stora projekt.

Som ett andra exempel, anta att du har kod som det här:

```
double foo ()
{
  double var = my_func();
  return var * var;
}
double bar ()
{
  double var = my_func();
  return var * var * var;
}
```

Om du håller musen över symbolen `var` i funktionen `bar` får du möjlighet att se alla användningar av symbolen. Att klicka på den visar bara användningen av variabeln i funktionen `bar` eftersom KDevelop förstår att variabeln `var` i funktionen `foo` inte har någonting att göra med den. På liknande sätt, kan du byta namn på variabeln med ett högerklick på variabelnamnet. Att göra det påverkar bara variabeln i `bar`, men inte den andra med samma namn i `foo`.

Men KDevelop är inte bara en intelligent kodeditor. Det finns andra saker som KDevelop är bra på. Naturligtvis färglägger det källkoden med olika färger, har anpassningsbar indentering, ett integrerat gränssnitt för GNU avlusaren gdb, kan visa dokumentationen för en funktion om musen hålls över där funktionen används, kan hantera olika sorters byggmiljöer och kompilatorer (t.ex. för projekt baserade på **make** och **cmake**), och har många andra praktiska funktioner som beskrivs i den här handboken.

Kapitel 2

Sessioner och projekt: KDevelops grunder

I det här avsnittet går vi igenom en del av terminologin rörande hur KDevelop ser på världen och hur arbete struktureras. Närmare bestämt introducerar vi koncepten *session* och *projekt* och förklarar hur projekten man vill arbeta på ställs in i KDevelop.

2.1 Terminologi

KDevelop har koncepten *session* och *projekt*. En session innehåller alla projekt som har något att göra med varandra. I exemplen som följer, anta att du är utvecklare av både ett bibliotek och ett program som använder det. Du kan se KDE:s kärnbibliotek som det förra och KDevelop som det senare. Ett annat exempel: Låt oss säga att du är en programmerare av Linux® kärna, men också arbetar på en drivrutin för Linux® som inte ännu har infogats i kärnans träd.

Så med det senare som exempel, skulle du ha en session i KDevelop som har två projekt: Linux®-kärnan och drivrutinen. Du bör vilja gruppera dem i en enda session (istället för att ha två sessioner med ett enda projekt vardera) eftersom det är användbart att kunna se kärnans funktioner och datastrukturer i KDevelop när du skriver källkod för drivrutinen, så att du exempelvis kan få kärnans funktions- och variabelnamn automatiskt expanderade, eller så att du kan se kärnans funktionsdokumentation medan du arbetar på drivrutinen.

Föreställ dig nu att du också råkar vara en utvecklare av KDE. Då skulle du ha en andra session som innehåller KDE som ett projekt. Du skulle i princip kunna ha en session för allt detta, men det finns inte någon riktig anledning att göra det. För arbete med KDE, behöver du inte kunna komma åt kärnans eller drivrutinens funktioner, och du vill inte att KDE:s klassnamn ska expanderas automatiskt medan du arbetar med Linux®-kärnan. Till sist, att bygga några KDE-bibliotek är oberoende av omkompilering av Linux®-kärnan (medan när du än kompilerar drivrutinen vore det också bra att kompilera om Linux®-kärnan om några av kärnans deklarationsfiler har ändrats).

Slutligen, en annan användning av sessioner är om du både arbetar på den aktuella utvecklingsversionen av ett projekt, samt på en gren. I detta fall vill du inte att KDevelop blandar ihop klasser som hör till huvudversionen och grenen, så du har två sessioner med samma uppsättning projekt men från olika kataloger (som motsvarar olika utvecklingsgrenar).

2.2 Ställa in en session och importera ett befintligt projekt

Låt oss hålla oss till exemplet med Linux®-kärnan och drivrutinen. Du kanske vill ersätta din egen uppsättning bibliotek eller projekt istället för de här två exemplen. För att skapa en ny session som innehåller de två projekten gå till menyn **Session** → **Starta ny session** längst upp till vänster (eller om det är första gången du använder KDevelop, använd helt enkelt sessionen du får första gången, default, som är tom).

Därefter vill vi befolka sessionen med projekt som vi för tillfället antar redan finns någonstans (fallet med att starta projekt från början beskrivs på ett annat ställe i handboken). För att göra det finns det i huvudsak två metoder, beroende på om projektet redan finns någonstans på hårddisken eller om det måste laddas ner från en server.

2.2.1 Alternativ 1: Importera ett projekt från en server för ett versionskontrollsystem

Låt oss först anta att projektet vi vill skapa, Linux®-kärnan, finns i något versionskontrollsystem på en server, men att du inte ännu har checkat ut det till din lokala hårddisk. I detta fall, gå till menyn **Projekt** för att skapa Linux®-kärnan som ett projekt inne i den aktuella sessionen och utför följande steg:

- Gå till **Projekt** → **Hämta projekt** för att importera ett projekt
- Därefter har du flera alternativ för att starta ett nytt projekt i den aktuella sessionen, beroende på var källkodsfilerna ska komma från. Du kan helt enkelt peka KDevelop på en befintlig katalog (se alternativ 2 nedan), eller be KDevelop att hämta källkoden från ett arkiv.
- Med antagandet att du inte redan har en version utcheckad:
  - Under **Välj källa** i dialogrutan, välj att använda **Från filsystem**, **Subversion**, **Git**, **GitHub** eller **KDE**
  - Välj en arbetskatalog som mål dit källkoden ska checkas ut
  - Välj en webbadress för platsen för arkivet där källkodsfilerna kan hämtas
  - Klicka på **Hämta**. Det kan ta ganska lång tid, beroende på anslutningens hastighet och projektets storlek. Tyvärr visar inte förloppsraden någonting i KDevelop 4.2.x, men du kan följa förloppet genom att periodiskt titta på utmatningen på kommandoraden från kommandot

    ```
    du -sk /sökväg/till/KDevelop/projekt
    ```

    för att se hur mycket data som redan har laddats ner.

> NOT
> Problemet med förloppsraden har rapporterats som KDevelop fel 256832.

> NOT
> Under processen får jag också felmeddelandet *Du måste ange en giltig plats för projektet*, vilket kan ignoreras utan problem.

- Du blir tillfrågad om att välja en KDevelop projektfil i katalogen. Eftersom du troligen inte har någon ännu, klicka helt enkelt på **Nästa**.

- Tryck på **Nästa** igen
- KDevelop ber dig sedan välja en projekthanterare. Om projektet använder vanliga UNIX® make-filer, välj projekthanteraren för eget byggsystem.
- KDevelop börjar sedan tolka hela projektet. Återigen tar det en hel del tid att gå igenom alla filer och indexera klasser, etc. Längst ner till höger i huvudfönstret finns en förloppsrad som visar hur långt processen har kommit (om du har flera processorkärnor kan du accelerera processen genom att gå till menyalternativet **Inställningar** → **Anpassa KDevelop**, sedan välja **Bakgrundstolkning** till vänster och öka antal trådar för bakgrundstolkning till höger).

2.2.2 Alternativ 2: Importera ett projekt som redan finns på hårddisken

Som ett alternativ, om projektet du vill arbeta med redan finns på hårddisken (exempelvis för att du har laddat ner det som en tar-fil från en FTP-server, eftersom du redan har checkat ut en version av projektet som ett versionskontrollsystem, eller för att det är ditt eget projekt som *bara* finns på din egen hårddisk), använd då **Projekt** → **Öppna/Importera projekt** och välj katalogen där projektet finns i dialogrutan.

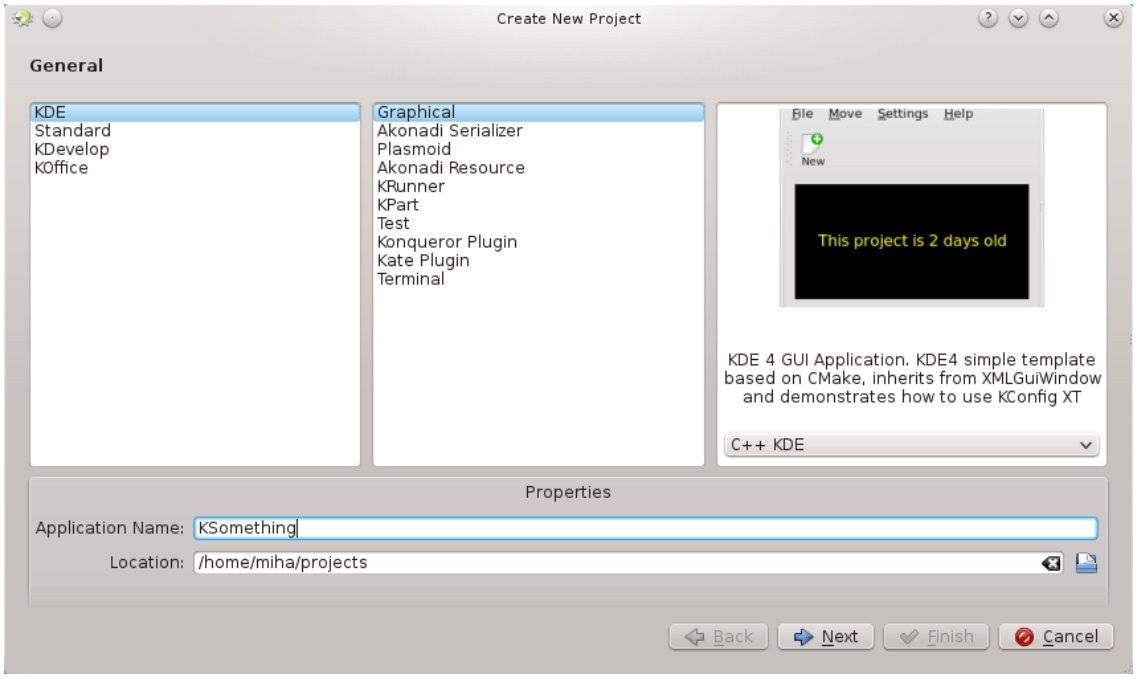
2.3 Ställa in ett program som andra projekt

Nästa sak som du ska göra är att lägga till andra projekt i samma session. I exemplet ovan, skulle du lägga till drivrutinen som andra projekt, vilket kan göras med exakt samma steg.

Om du har flera program eller bibliotek, upprepa helt enkelt stegen för att lägga till fler och fler projekt i sessionen.

2.4 Skapa projekt från början

Det finns förstås också möjligheten att du vill påbörja ett nytt projekt från början. Det kan göras genom att använda menyalternativet **Projekt** → **Nytt från mall...**, som ger dig en dialogruta för att välja mall. Vissa projektmallar tillhandahålls med KDevelop men ännu fler är tillgängliga genom att installera programmet KAppTemplate. Välj projekttyp och programspråk i dialogrutan, ange ett namn och plats för projektet, och klicka på **Nästa**.



Den andra sidan i dialogrutan låter dig ställa in ett versionskontrollsystem. Välj system som du vill använda och fyll i den systemspecifika inställningen om det behövs. Om du inte vill använda ett versionskontrollsystem, eller vill ställa in det manuellt senare, välj **Inget**. När du är nöjd med ditt val, klicka på **Slutför**.

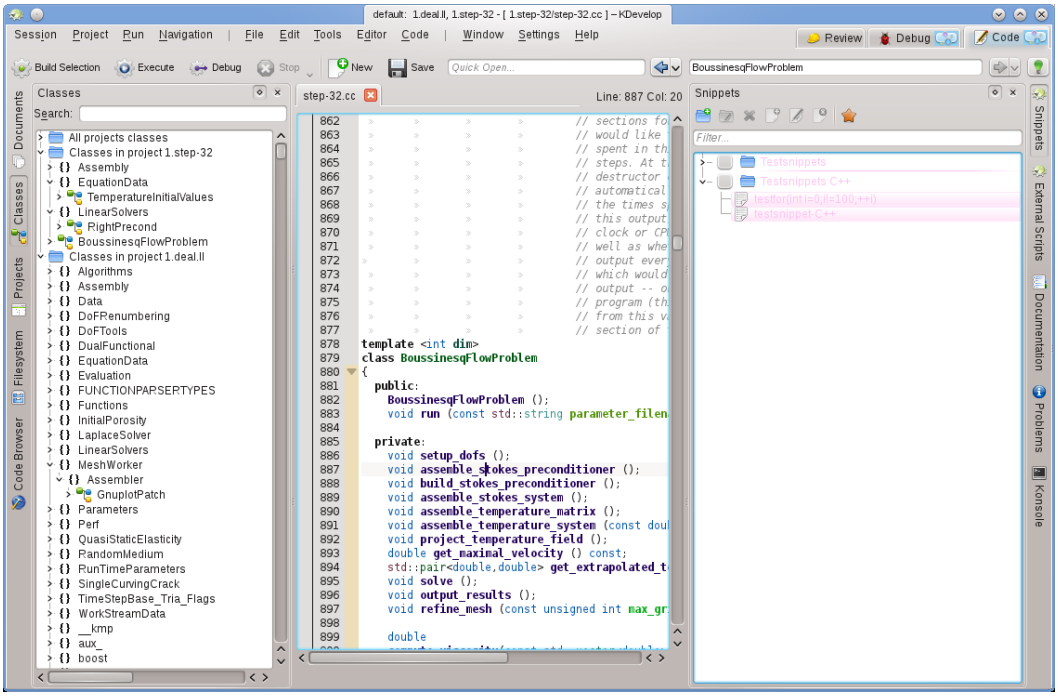
Projektet har nu skapats, så du kan försöka bygga eller installera det. Vissa mallar inkluderar kommentarer i koden, eller till och med en separat README-fil, och det rekommenderas att du läser dem först. Därefter kan du börja arbeta på projektet genom att lägga till vilka funktioner du än vill ha.

Kapitel 3

Arbeta med källkod

Förutom avlusning, är att läsa igenom och skriva källkod det du spenderar längst tid med vid programvaruutveckling. I detta syfte erbjuder KDevelop dig många olika sätt att utforska källkod och skriva den på ett produktivare sätt. Som beskrivs mer detaljerat i följande avsnitt, är inte KDevelop bara en källkodseditor. Istället är det ett källkodshanteringssystem som ger dig olika vyer av extraherad information om filerna som kollektivt utgör sessionens källkod.

3.1 Verktyg och vyer



För att fungera med projekt har KDevelop konceptet *verktyg*. Ett verktyg tillhandahåller en särskild vy av källkoden, eller en åtgärd som kan göras med den. Verktyg representeras av knappar omkring fönstrets omkrets (i vertikal textredigering längs vänster och höger marginal, eller horisontellt längs nedre marginalen). Om du klickar på dem, expanderas de till ett delfönster, en *vy*, i huvudfönstret. Om du klickar på verktygsknappen igen, försvinner delfönstret.

För att få ett delfönster att försvinna, kan du också klicka på **x** längst upp till höger i delfönstret.

Bilden ovan visar ett visst urval av verktyg, placerade längs vänster och höger marginaler. På bilden är verktyget **Klasser** öppet till vänster och verktyget **Testsnuttar** till höger, tillsammans med en editor av en källkodsfil i mitten. I praktiken har man troligen bara editorn och kanske verktyget **Klasser** eller **Kodbläddrare** öppet till vänster. Andra verktygsvyer är troligen bara tillfälligt öppna medan man använder verktyget, vilket oftast lämnar mer utrymme för editorn.

När KDevelop startas för första gången ska redan verktygsknappen **Projekt** finnas. Klicka på den: det öppnar ett delfönster som visar projekten som du har lagt till i sessionen längst ner, och en filsystemvy med katalogerna i projekten längst upp.

Det finns många andra verktyg som kan användas med KDevelop, där inte alla initialt presenteras som knappar längs omkretsen. För att lägga till några, gå till menyalternativet **Fönster** → **Lägg till verktygsvy**. Här är några som troligen är användbara:

- **Klasser**: En fullständig lista över alla klasser som är definierade i ett av projekten eller sessionen med alla deras medlemsfunktioner och variabler. Att klicka på någon av medlemmarna öppnar ett fönster för källkodsredigering på platsen för objektet som du klickade på.
- **Dokument**: Listar några av de senast besökta filerna, enligt sort (t.ex. källkodsfiler, programfixar, vanliga textdokument).
- **Kodbläddrare**: Beroende på markörposition i en fil, visar verktyget saker som är relaterade. Om du exempelvis är på en rad med `#include` visar den information om filen som du inkluderar såsom vilka klasser som är deklarerade i den filen. Om du är på en tom rad på filnivå visar den klasserna och funktionerna deklarerade och definierade i den aktuella filen (alla som länkar, att klicka på dem går till den positionen i filen där deklarationen eller definitionen faktiskt finns). Om du är i en funktionsdefinition, visar den var deklarationen är och erbjuder en lista över platser där funktionen används.
- **Filsystem**: Visar dig en trädvy av filsystemet.
- **Dokumentation**: Låter dig söka efter manualsidor och andra hjälpdokument.
- **Textsnuttar**: Det tillhandahåller textföljder som man använder gång på gång och inte vill skriva in varje gång. Exempelvis, i projektet som bilden ovan skapades från, är det ofta behov av att skriva kod som

```
for (typename Triangulation< dim>::active_cell_iterator cell
      = triangulation.begin_active();
   cell != triangulation.end();
   ++cell)
```

Det är ett klumpigt uttryck men det ser nästan exakt likadant ut varje gång du behöver en sådan snurra, vilket gör det till en bra kandidat för en textsnutt.

- **Terminal**: Visar ett kommandoradsfönster inne i KDevelops huvudfönster, för enstaka kommandon som du kan vilja använda (t.ex. för att köra `./configure`).

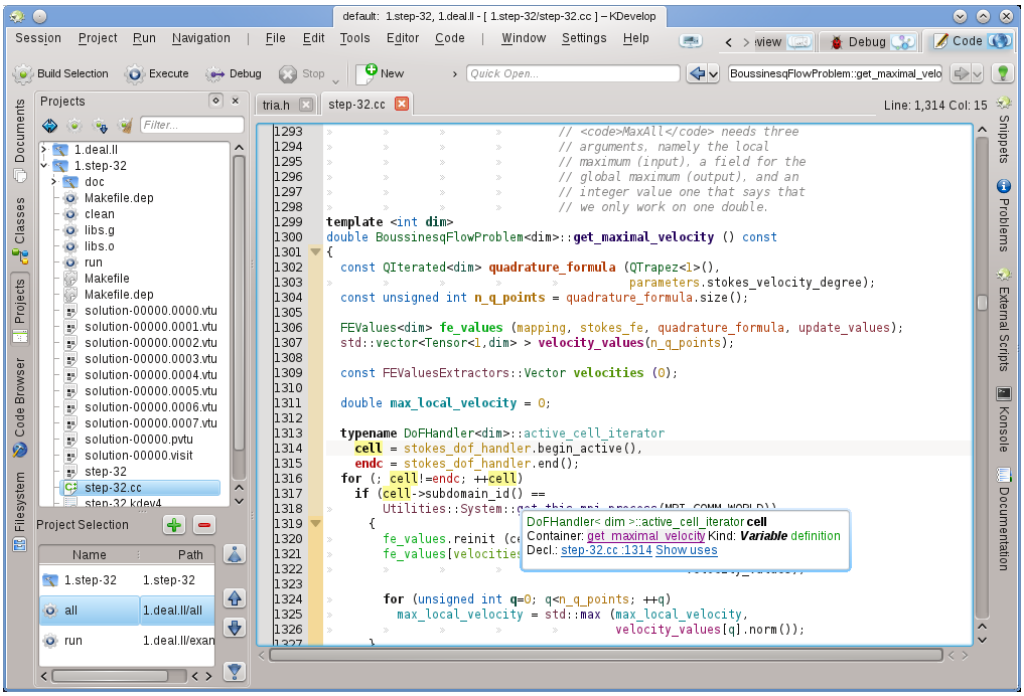
En fullständig lista över verktyg och vyer ges här.

För många programmerare är vertikalt skärmutrymme det viktigaste. För att uppnå det, kan verktygsvyerna arrangeras på vänster och höger marginal i fönstret. För att flytta ett verktyg, klicka på dess symbol med höger musknapp och välj en ny position för det.

3.2 Utforska källkod

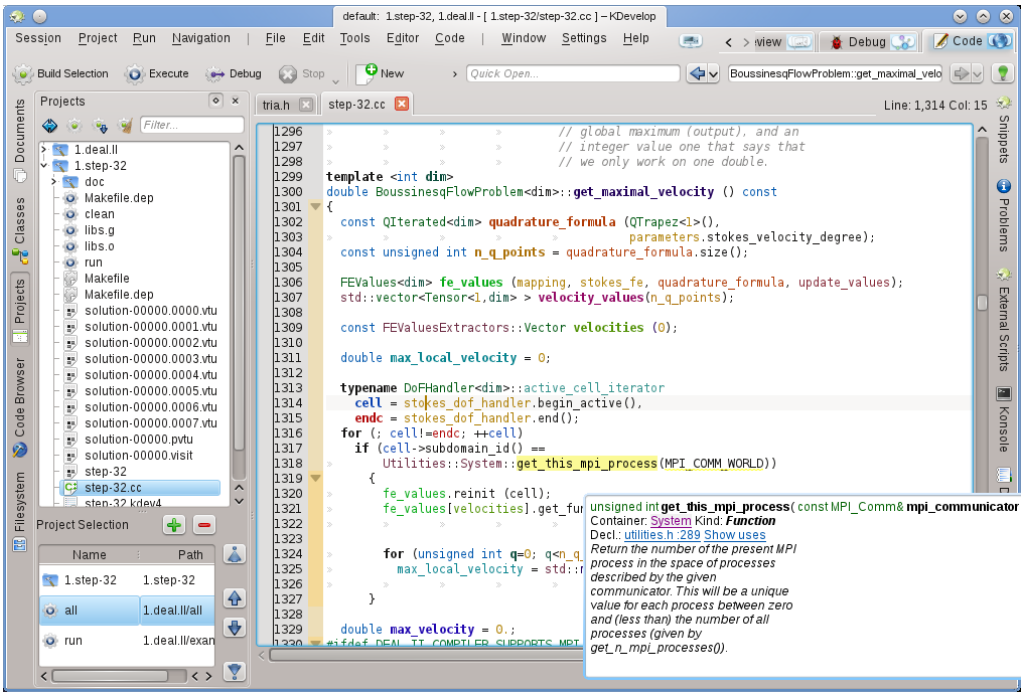
3.2.1 Lokal information

KDevelop *förstår* källkod, och är som en följd av det mycket bra på att tillhandahålla information om variabler eller funktioner som kan dyka upp i programmet. Här är exempelvis en ögonblicksbild av att arbeta med ett stycke kod och hålla musen över symbolen `cell` på rad 1316 (om du arbetar tangentbordsorienterat, kan du åstadkomma samma effekt genom att hålla ner tangenten **Alt** en stund):



KDevelop visar ett verktygstips som inkluderar variabelns typ (här: `DoFHandler<dim>active_cell_iterator`), var variabeln är deklarerad (*behållaren*, vilket här är den omgivande funktionen `get_maximal_velocity` eftersom det är en lokal variabel), vad den är (en variabel, inte en funktion, klass eller namnrymd) och var den är deklarerad (på rad 1314, bara några få rader upp i koden).

I det aktuella sammanhanget, har inte symbolen som musen hölls över någon tillhörande dokumentation. Hade musen hållits över symbolen `get_this_mpi_process` på rad 1318, hade resultatet blivit följande:

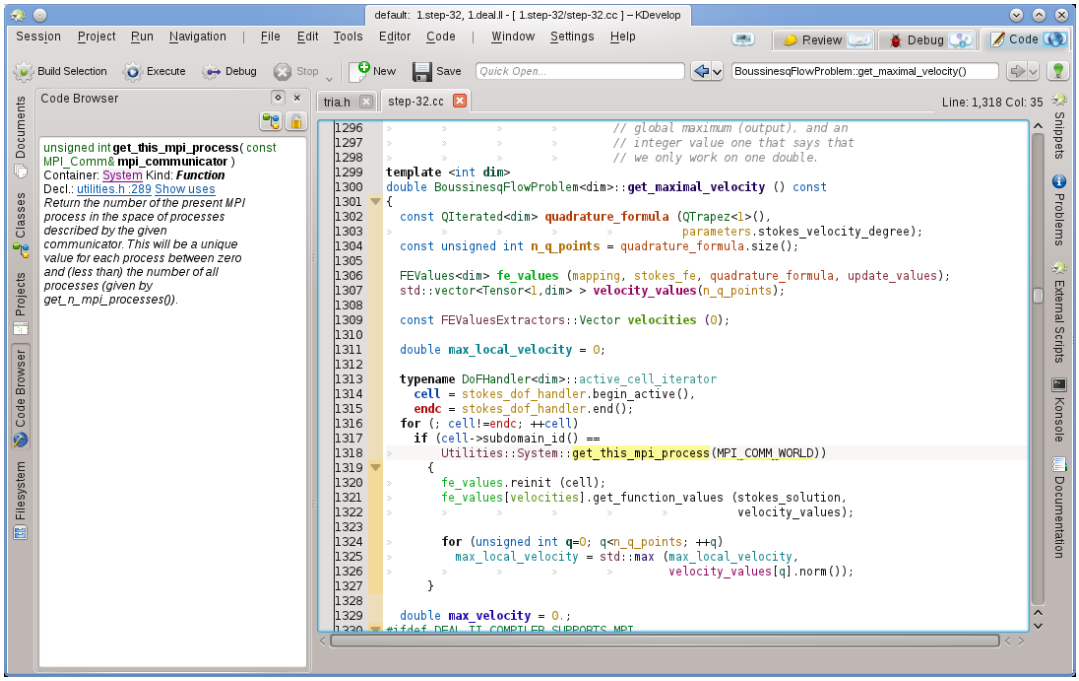


Här har KDevelop korsrefererat deklarationen från en helt annan fil (`utilities.h`, som i själva verket ingår i ett annat projekt i samma session) tillsammans med den doxygen-liknande kommentaren som hör till deklarationen där.

Vad som gör verktygstipsen ännu användbarare är att de är dynamiska: Man kan klicka på behållaren för att få information om sammanhanget där variabeln är deklarerad (dvs. om namnrymden `System`, såsom var den är deklarerad, definierad, använd, eller vad dess dokumentation är) och man kan klicka på de blåa länkarna som flyttar markörposition till platsen där Symbol är deklarerad (t.ex. i `utilities.h`, på rad 289) eller ger en lista över platser där symbolen används i den aktuella filen eller i den aktuella sessionens samtliga projekt. Det senare är ofta användbart om du exempelvis vill utforska hur en viss funktion används i en större kodbas.

NOT

Informationen i ett verktygstips är flyktig: den beror på att du håller nere tangenten **Alt** eller håller musen över det. Om du vill ha en mer permanent plats för det, öppna verktygsvyn **Kodbläddrare** i ett av delfönstren. Här är exempelvis markören på samma funktion som i exemplet ovan, och verktygsvyn till vänster visar samma sorts information som i verktygstipset tidigare:



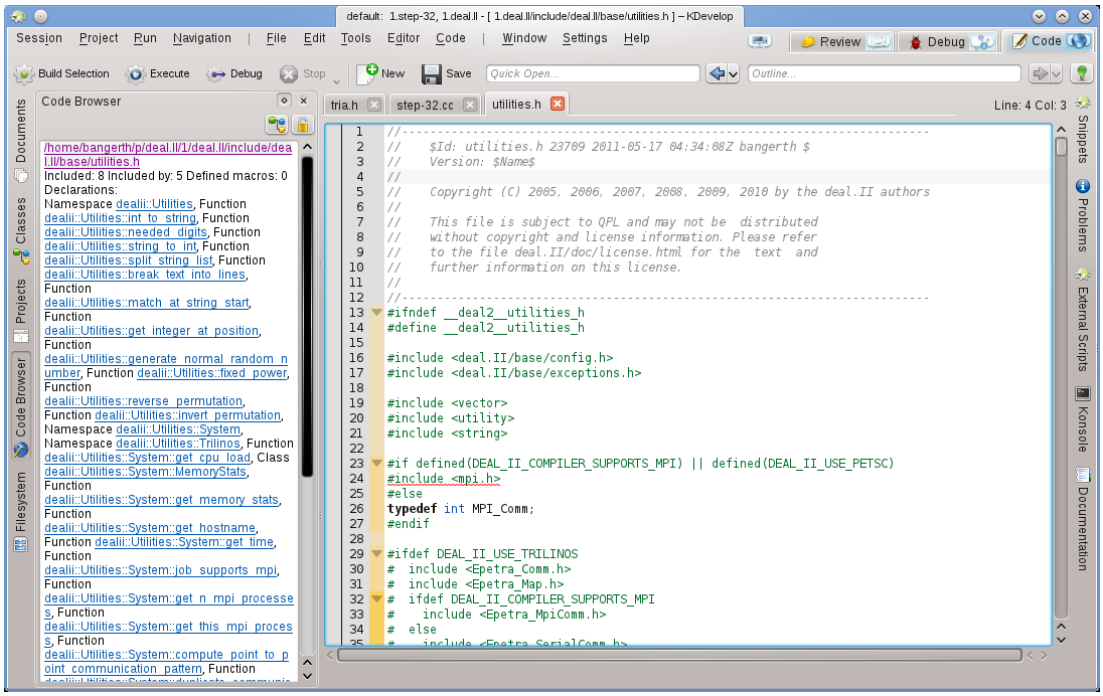
Att flytta omkring markören till höger ändrar informationen som visas till vänster. Dessutom, att klicka på knappen **Lås nuvarande vy** låter dig låsa informationen, vilket gör den oberoende av markörförflyttningar medan du utforskar informationen som visas där.

NOT

Den här sortens sammanhangsberoende information är tillgänglig på många andra ställen i KDevelop, inte bara i källkodseditorn. Genom att exempelvis hålla ner tangenten **Alt** i en kompletteringslista (t.ex. vid snabböppna) ger också sammanhangsinformation om den aktuella symbolen.

3.2.2 Information på filnivå

Nästa nivå uppåt är att erhålla information om hela källkodsfilen som du för närvarande arbetar på. För att åstadkomma det, placera markören på filnivå i aktuell fil och titta på vad verktygsvyn **Kodbläddrare** visar:

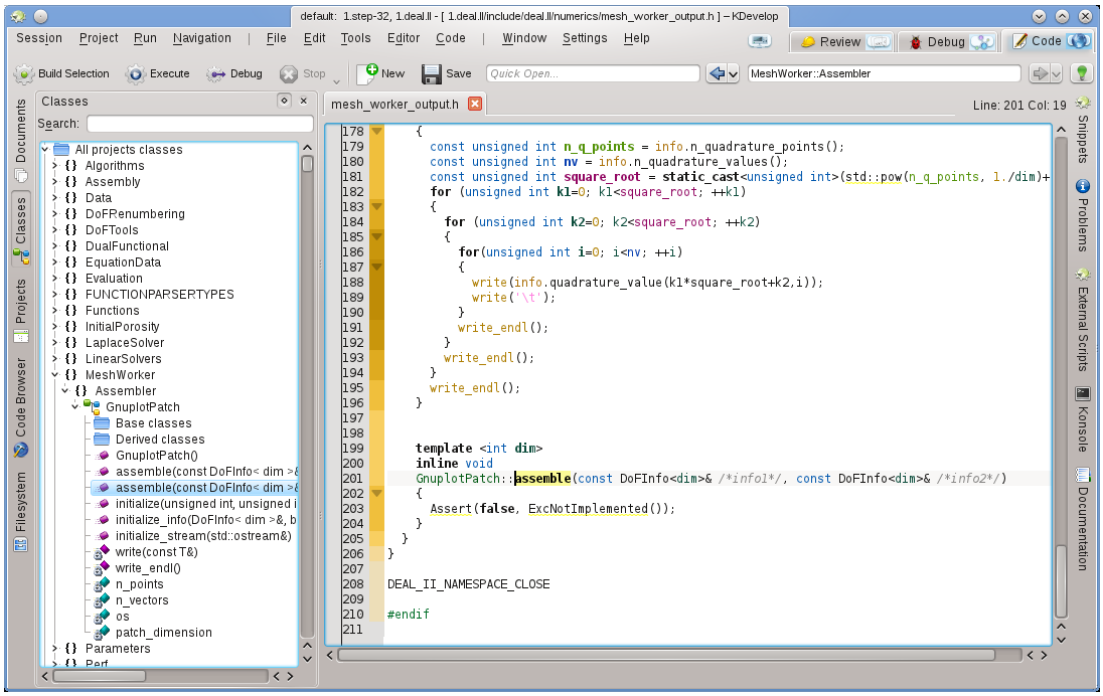


Här visar den en lista över namnrymder, klasser och funktioner deklarerade eller definierade i den aktuella filen, ger dig en översikt av vad som händer i filen och ett sätt att gå direkt till vilken som helst av deklarationer eller definitioner utan att rulla uppåt eller neråt i filen eller söka efter en specifik symbol.

> NOT
> Informationen som visas för filnivå är samma som presenteras med ‘Översikt’ som beskrivs nedan för att navigera i källkod. Skillnaden är att översikt bara ger ett tillfälligt verktygstips.

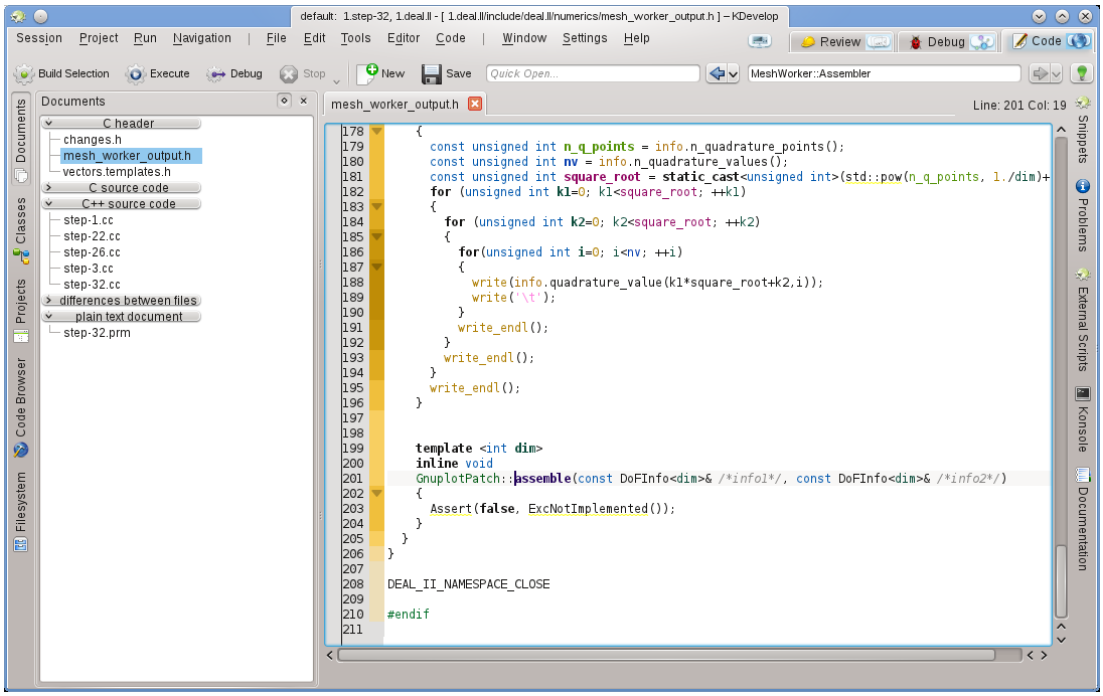
3.2.3 Information på projekt- och sessionsnivå

Det finns många sätt att erhålla information om ett helt projekt (eller, i själva verket, om alla projekt i en session). Den här sortens information tillhandahålls typiskt via diverse verktygsvyer. Exempelvis tillhandahåller verktygsvyn **Klasser** en trädstruktur över alla klasser och omgivande namnrymder för alla projekt i en session, tillsammans med medlemsfunktionerna och variablerna för var och en av klasserna:



Att hålla musen över en post tillhandahåller återigen information om symbolen. Platserna för dess deklaration och definition, och dess användningar. Att dubbelklicka på en post i trädvyn öppnar ett editorfönster på platsen där symbolen deklareras eller definieras.

Men det finns andra sätt att titta på global information. Exempelvis ger verktyget **Dokument** en vy av projektet med avseende på vilka sorters filer eller andra dokument som projektet består av:



3.2.4 Förklaring av färgläggning med regnbågsfärger

KDevelop använder diverse färger för att markera olika objekt i källkoden. Om du vet vad de olika färgerna betyder, kan du mycket snabbt extrahera en mängd information från källkoden bara genom att titta på färgerna, utan att läsa ett enda tecken. Färgläggningsreglerna är följande:

- Objekt av typerna Klass, Struct, Enum (värdena och typen), (globala) funktionerna, och klassmedlemmarna har var och en sina egna färgtilldelningar (klasser är gröna, enum är mörkröda, och medlemmar är mörkgula eller violetta, (globala) funktioner är alltid violetta).
- Alla globala variabler färgläggs med mörkgrönt
- Identifierare som är en typedef av en annan typ färgläggs med blågrönt.
- Alla deklarationer och definitioner av objekt använder fetstil.
- Om en medlem används inne i sammanhanget den är definierad (basklass eller härledd klass) visas den med gult, annars visas den med violett.
- Om en medlem är privat eller skyddad, färgläggs den med en något mörkare färg när den används.
- För variabler lokala på funktionsnivå, väljs regnbågsfärger baserad på ett kondensat av identifieraren. Det inkluderar funktionens parametrar. En identifierare har alltid samma färg på dess nivå (men samma identifierare får en annan färg om den representerar ett annat objekt, dvs. om den definieras om på en mer nästlad nivå), och man får oftast samma färg på samma identifierarnamn på olika nivåer. Om du sålunda har olika funktioner som har parametrar med samma namn, ser argumenten alla likadana ut kodmässigt. Regnbågsfärgerna kan stängas av separat från den globala färgläggningen i inställningsdialogrutan:
- Identifierare där KDevelop inte kan bestämma motsvarande deklaration är färglagda med vitt. Det kan ibland orsakas av saknade `#include`-direktiv.
- Förutom den färgläggningen, används den vanliga syntaxfärgläggningen i editorn, som är känd från Kate. KDevelops semantiska färgläggning överskrider alltid editorns färgläggning om det finns en konflikt.

3.3 Navigera i källkod

I det föregående avsnittet, har vi beskrivit hur källkod utforskas, dvs. få information om symboler, filer och projekt. Nästa steg är sedan att flytta sig i kodbasen, dvs. navigera i den. Det finns återigen olika nivåer där det är möjligt: lokalt, inom en fil, och inom ett projekt.

> NOT
> Många av sätten att navigera genom koden kan kommas åt via menyn **Navigera** i KDevelops huvudfönster.

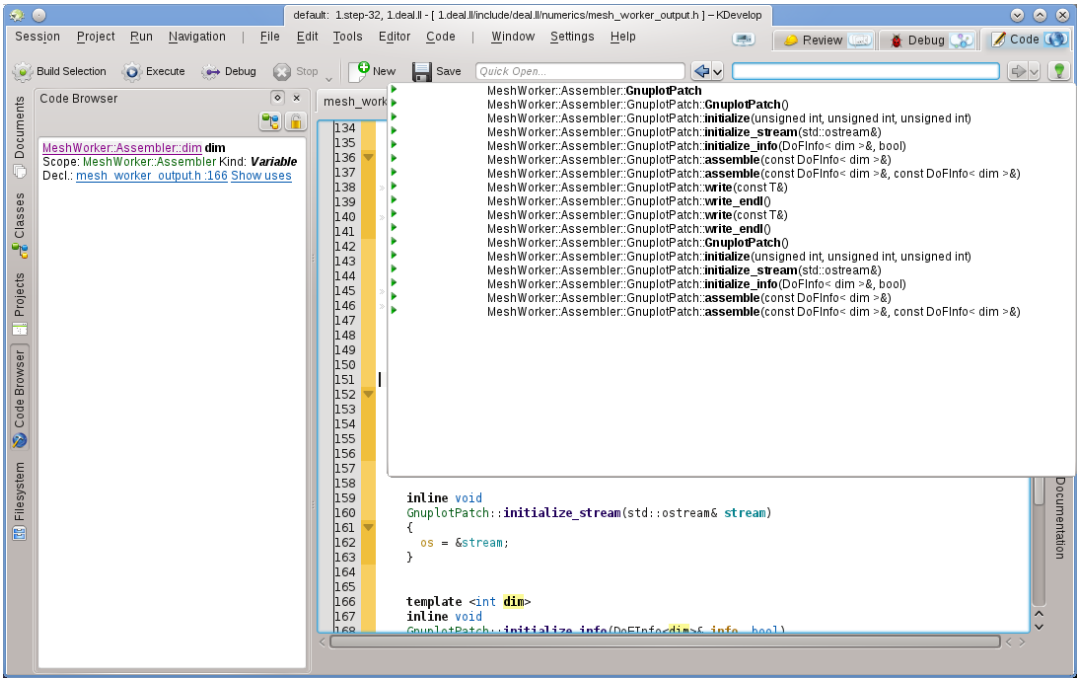
3.3.1 Lokal navigering

KDevelop är mycket mer än en editor, men den är *också* en källkodseditor. Som sådan, kan du förstås flytta markören uppåt, neråt, åt vänster och åt höger i en källkodsfil. Du kan också använda tangenterna **PageUp** och **PageDown**, och alla andra kommandon som du är van vid från vilken användbar editor som helst.

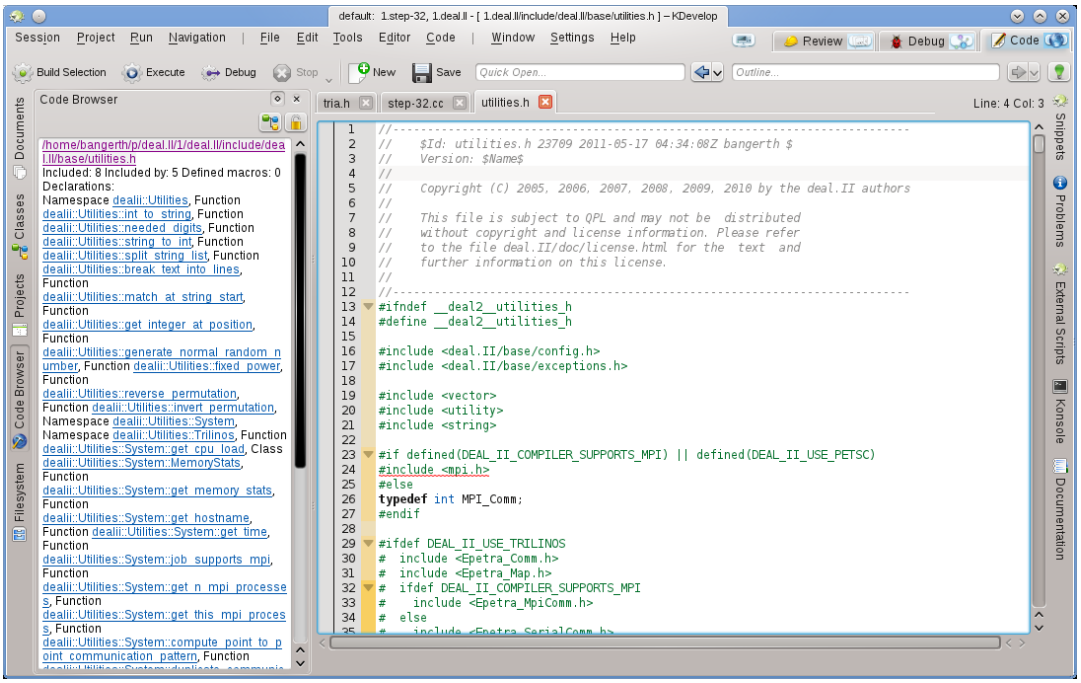
3.3.2 Navigering på filnivå och översiktsläge

På filnivå erbjuder KDevelop många möjliga sätt att navigera igenom källkoden. Exempelvis:

- **Översikt**: Du kan få en översikt av vad som finns i den aktuella filen på åtminstone tre olika sätt:
  - Att klicka på textrutan **Översikt** längst upp till höger i huvudfönstret, eller trycka på **Alt-Ctrl-N**, visar en kombinationsmeny som listar alla funktioner och klassdeklarationer:



    Därefter kan du helt enkelt välja vilken du vill gå till, eller (om det finns många) börja skriva vilken text som helst som ingår i namnen som visas. I detta fall, medan du fortsätter skriva, blir listan mindre och mindre medan namn som inte motsvarar texten som redan har skrivits tills du är klar att välja ett av de som visas.
  - Placera markören på filnivå (dvs. utanför någon funktion eller klassdeklarationer eller definitioner) och ha verktyget **Kodbläddrare** öppet:



Det tillhandahåller en översikt av vad som händer i den aktuella filen, och låter dig välja vart du vill gå.

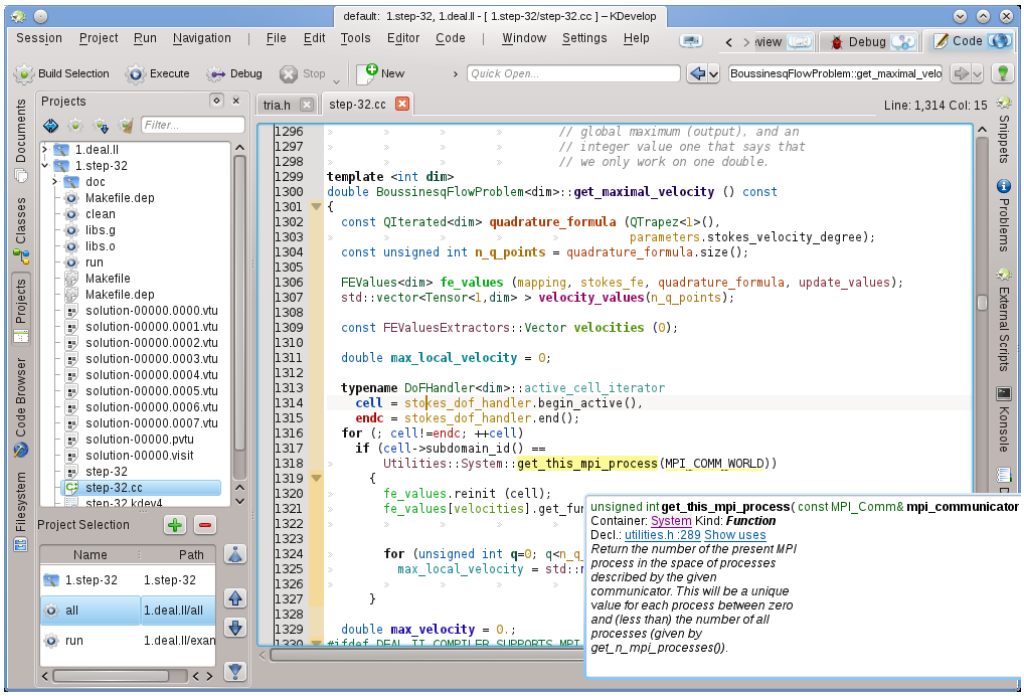
– Att hålla musen över fliken för en av de öppna filerna ger också en översikt av filen under den fliken.

- Källkodsfilerna är organiserade som en lista över funktionsdeklarationer eller definitioner. **Alt-Ctrl-Page Up** eller **Alt-Ctrl-Page Down** går till föregående eller nästa funktionsdefinition i filen.

3.3.3 Navigering på projekt- och sessionsnivå: semantisk navigering

Som nämnts på andra ställen, betraktar KDevelop inte oftast individuella källkodsfiler utan tittar istället på projektet som en helhet (eller snarare på alla projekt som ingår i den aktuella sessionen). Som en konsekvens, erbjuder det många möjligheter att navigera igenom hela projekt. Vissa av dem härleds från vad vi redan beskrivit i avsnittet Utforska källkod medan andra är genuint annorlunda. Det gemensamma temat är att navigeringsfunktionerna är baserade på en *semantisk förståelse* av koden, dvs. de erbjuder någonting som kräver tolkning av hela projekt och anslutande data. Följande lista visar några sätt att navigera igenom källkod som är utspridda över ett potentiellt mycket stor antal filer:

- Som syns i avsnittet Utforska källkod, kan du få ett verktygstips som förklarar individuella namnrymder, funktions- eller variabelnamn genom att hålla musen över dem eller hålla tangenten **Alt** nedtryckt en stund. Här är ett exempel:



Att klicka på länkarna för deklarationen av en symbol eller expandera listan av användningar låter dig gå till positionerna, och öppnar respektive fil om nödvändigt och placerar markören på motsvarande position. En liknande effekt kan åstadkommas genom att använda verktygsvyn **Kodbläddrare** som också beskrivits tidigare.

- Ett snabbare sätt att gå till deklarationen av en symbol utan att behöva klicka på länkarna i verktygstipset är att tillfälligt aktivera **Bläddringsläge för källkod** genom att hålla nere tangenten **Alt** eller **Ctrl**. I det läget är det möjligt att direkt klicka på vilken symbol som helst i editorn för att gå till dess deklaration.

- **Snabböppna**: Ett mycket kraftfullt sätt att gå till andra filer eller platser är att använda de olika metoderna för att *snabböppna* i KDevelop. Det finns fyra versioner av dem:

  – **Snabböppna klass** (**Navigera** → **Snabböppna plats** eller **Alt-Ctrl-C**): Du får en lista över alla klasser i sessionen. Börja skriva (en del av) namnet på en klass så fortsätter listan att skäras ner till bara de som verkligen motsvarar vad du hittills har skrivit in. Om listan är kort nog, välj ett element genom att använda tangenterna uppåt- eller neråtpil så tar KDevelop dig till positionen där klassen är deklarerad.

  – **Snabböppna funktion** (**Navigera** → **Snabböppna funktion** eller **Alt-Ctrl-M**): Du får en lista över alla (medlems) funktioner som ingår i projekten i den aktuella sessionen, och du kan välja dem i den på samma sätt som ovan. Observera att listan kan innehålla både funktionsdeklarationer och definitioner.

  – **Snabböppna fil** (**Navigera** → **Snabböppna fil** eller **Alt-Ctrl-O**): Du får en lista över alla filer som ingår i projekten i den aktuella sessionen, och du kan välja från den på samma sätt som ovan.

  – **Generell snabböppna** (**Navigera** → **Snabböppna** eller **Alt-Ctrl-Q**): Om du glömmer bort vilken tangentkombination hör ihop med vilket av de ovanstående kommandona, är det den generella lösningen. Det visar helt enkelt en kombinerad lista över alla filer, funktioner, klasser, och andra saker som du kan välja bland.

- **Gå till deklaration/definition**: När en (medlems) funktion implementeras, behöver man ofta gå tillbaka till stället där funktionen är deklarerad, för att exempelvis hålla listan med funktionsargument synkroniserad mellan deklarationen och definitionen, eller för att uppdatera dokumentationen. För att göra det, placera markören på funktionsnamnet och välj **Navigera** → **Gå till deklaration** (eller tryck på **Ctrl-.**) för att gå till positionen där funktionen deklareras. Det finns många olika sätt att komma tillbaka till den ursprungliga positionen:

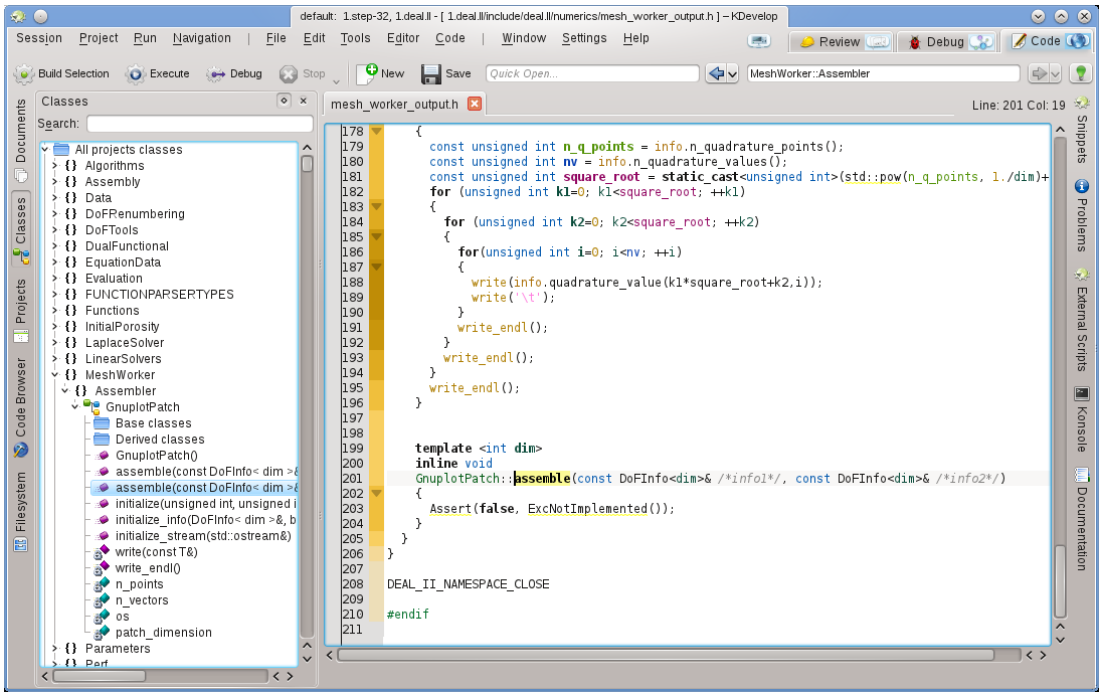
  - Välja **Navigering** → **Gå till definition** (eller tryck på **Ctrl-,**).
  - Välja **Navigering** → **Föregående besökta sammanhang** (eller tryck på **Meta-Vänster**), som beskrivs nedan.

NOT
Att gå till deklarationen av en symbol är något som inte bara fungerar när markören placeras på funktionsnamnet som du för närvarande håller på att implementera. Det fungerar dessutom på andra symboler: Att placera markören på en (lokal, global, eller medlems-) variabel och gå till dess deklaration tar dig också till deklarationens position. På liknande sätt kan du placera markören på ett klassnamn, exempelvis i en variabel- eller funktionsdeklaration, och gå till positionen för dess deklaration.

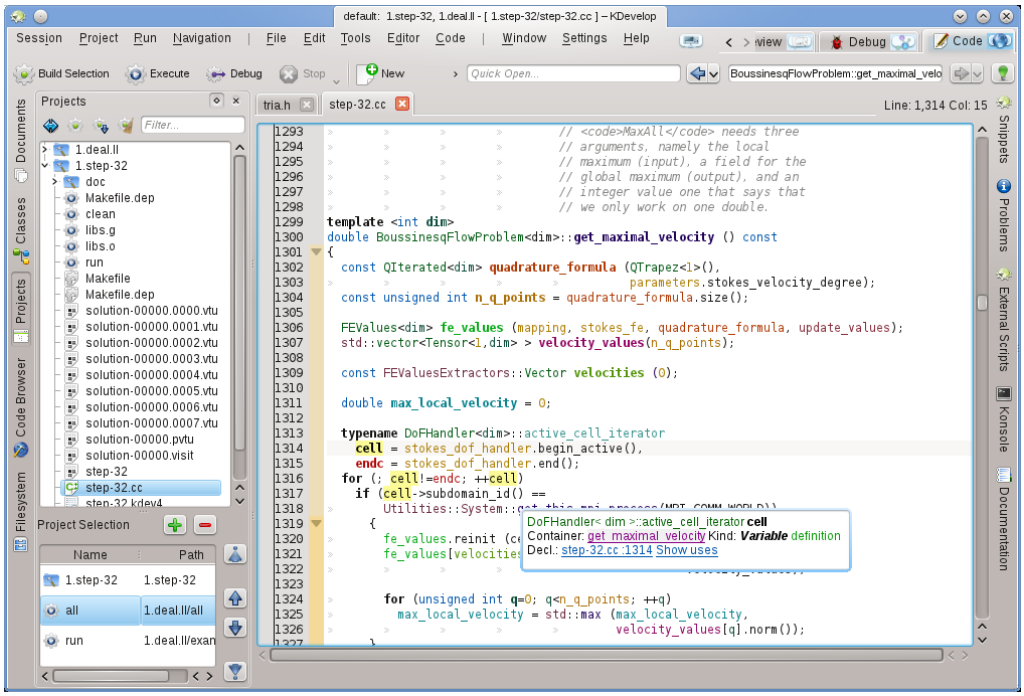
- **Byt mellan deklaration och definition**: I exemplet ovan måste markören först placeras på funktionsnamnet för att gå till positionen för deklarationen av den aktuella funktionen. För att undvika steget, kan du välja **Navigera** → **Byt definition med deklaration** (eller tryck på **Skift-Ctrl-C**) för att gå till funktionens deklaration som markören för närvarande befinner sig i. Att välja samma menyalternativ en gång till leder dig tillbaka till positionen där funktionen är definierad.
- **Föregående/Nästa användning**: Att placera markören på en lokal variabels namn och välja **Navigering** → **Nästa användning** (eller trycka på **Meta-Skift-Högerpil**) tar dig till nästa användning av variabeln i koden. (Observera att inte bara söker efter nästa förekomst av variabelnamnet utan tar i själva verket hänsyn till att variabler med samma namn men på olika nivåer är olika.) Samma sak fungerar för användning av funktionsnamn. Att välja **Navigering** → **Föregående användning** (eller trycka på **Meta-Skift-Vänsterpil**) tar dig till föregående användning av symbolen.

NOT
För att se listan över alla användningar av ett namn som kommandona går igenom, placera markören på det och öppna verktygsvyn **Kodbläddrare** eller tryck och håll nere tangenten **Alt**. Det förklaras mer detaljerat i avsnittet Utforska kod.

- **Sammanhangslistan**: Webbläsare har funktionen så att man kan gå bakåt och framåt i listan med senast besökta webbsidor. KDevelop har samma sorts funktioner, utom att istället för webbsidor du besökt är det *sammanhang*. Ett sammanhang är markörens nuvarande plats, och du kan ändra den genom att navigera från den genom att använda vad som helst utom markörkommandon, till exempel genom att klicka på en plats som tillhandahålls av ett verktygstips, i verktygsvyn **Kodbläddrare**, ett av alternativen som ges av menyn **Navigera**, eller något annat navigeringskommando. Genom att använda **Navigera** → **Föregående besöka sammanhang** (**Meta-Vänsterpil**) och **Navigera** → **Nästa besökta sammanhang** (**Meta-Högerpil**) tar dig till sammanhang precis som knapparna **bakåt** och **framåt** i en bläddrare tar dig till föregående eller nästa webbsida i listan med besökta sidor.
- Slutligen finns det verktygsvyer som låter dig navigera till olika platser i din kodbas. Verktyget **Klasser** tillhandahåller exempelvis en lista med namnrymd och klasser för alla projekt i den aktuella sessionen, och låter dig expandera den för att se medlemsfunktioner och variabler i var och en av klasserna:



Dubbelklicka på objektet (eller gå via den sammanhangsberoende menyn genom att använda höger musknapp), så tillhandahåller verktygsvyn **Projekt** en lista över filer som ingår i en session:



Återigen, dubbelklicka på en fil öppnar den.

3.4 Skriva källkod

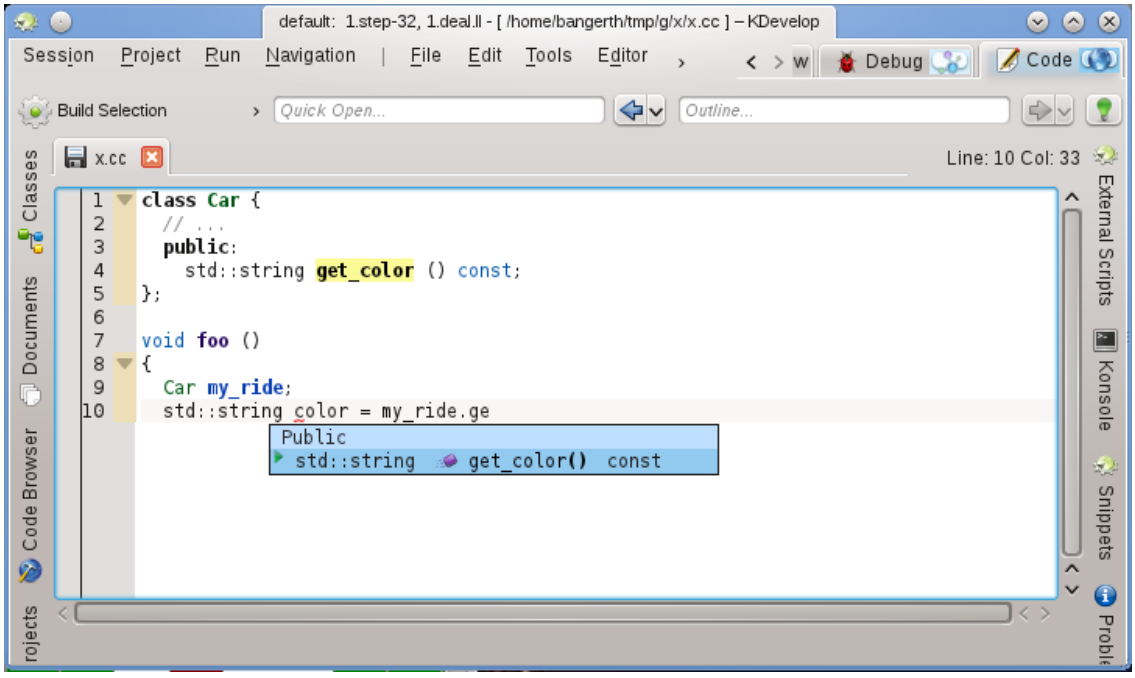
Eftersom KDevelop förstår projektets källkod, kan det hjälpa till med att skriva mer kod. Det följande ger en översikt av några av de olika sätt som det görs.

3.4.1 Automatisk komplettering

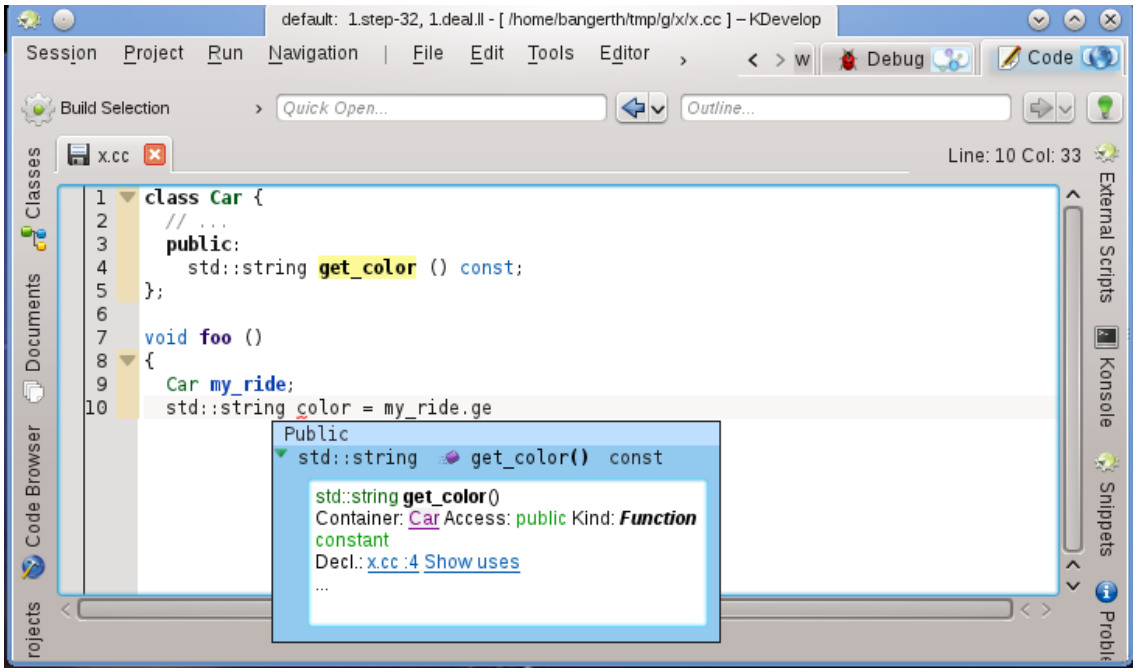
Den troligtvis mest användbara av alla funktioner i att skriva ny kod är automatisk komplettering. Betrakta exempelvis följande stycke kod:

```
class Car {
  // ...
  public:
    std::string get_color () const;
};
void foo()
{
  Car my_ride;
  // ...gör något med variabeln...
  std::string color = my_ride.ge
```

På den sista raden kommer KDevelop ihåg att variabeln `my_ride` har typen `Car`, och erbjuder sig att automatiskt komplettera namnet på medlemsfunktionen `ge` som `get_color`. I själva verket är allt du behöver göra att fortsätta skriva tills funktionen för automatisk komplettering har reducerat antal träffar till en, och sedan trycka på returtangenten:



Observera att du kan klicka på verktygstipset för att få mer information om funktionen förutom dess returtyp och om den är öppen:



Automatisk komplettering kan spara mycket skrivande om projektet använder långa variabel- och funktionsnamn. Dessutom undviker det felstavning av namn (och resulterande kompileringsfel) och gör det mycket enklare att komma ihåg de exakta funktionsnamnen. Om exempelvis alla hämtningsfunktioner börjar med `get_`, kan funktionen för automatisk komplettering bara visa dig en lista med möjliga hämtningsfunktioner när du har skrivit in de fyra första bokstäverna, vilket troligtvis påminner dig under processen vilken av funktionerna som är den riktiga. Observera att för att automatisk komplettering ska fungera, behöver varken deklarationerna av klassen `Car` eller variabeln `my_ride` vara i samma fil som du för närvarande skriver kod i. KDevelop behöver enbart veta att klasserna och variablerna hör ihop, dvs. filerna där kopplingen görs måste ingå i projektet som du för närvarande arbetar på.

NOT

KDevelop känner inte alltid till när det ska hjälpa dig med att komplettera kod. Om verktygstipset för automatisk komplettering inte visas automatiskt, tryck på **Ctrl**-**Mellanslag** för att visa en lista över kompletteringar manuellt. I allmänhet, för att automatisk komplettering ska fungera, måste KDevelop tolka dina källkodsfiler. Det sker i bakgrunden för alla filer som ingår i projekten i den aktuella sessionen efter du startar KDevelop, samt en stund efter du slutar skriva en bråkdels sekund (fördröjningen kan ställas in).

NOT

KDevelop tolkar bara filer som det anser vara källkod, som bestäms av filens Mime-typ. Typen är inte inställd innan första gången en fil sparas. Som konsekvens, utlöses inte tolkning för automatisk komplettering när en ny fil skapas och man börjar skriva kod i den förrän efter den har sparats för första gången.

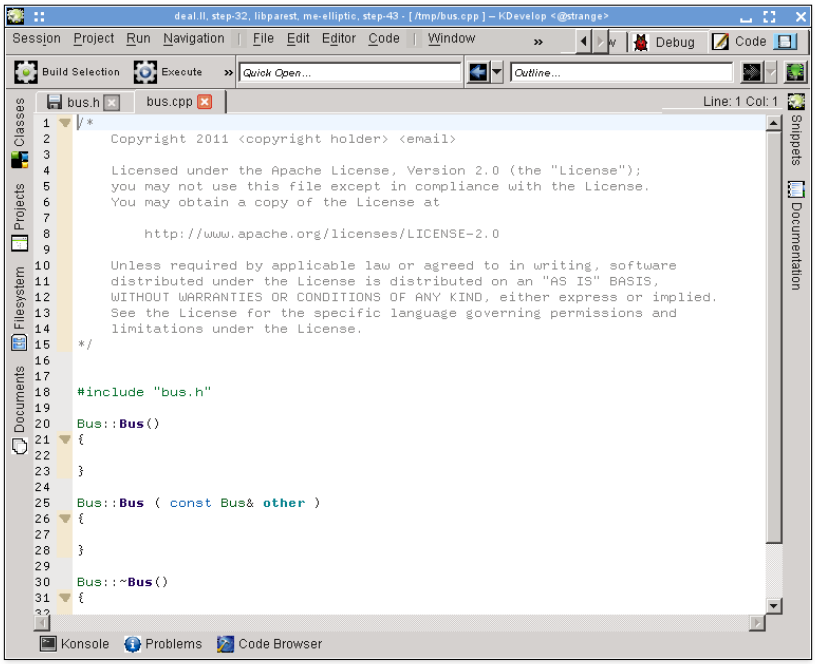
NOT
Som i föregående anmärkning, för att automatisk komplettering ska fungera måste KDevelop kunna hitta deklarationer i deklarationsfiler. För att göra det, söks ett antal standardsökvägar igenom. Om en deklarationsfil inte hittas automatiskt, stryks namnet på deklarationsfilen under med rött. I detta fall, högerklicka på den för att explicit tala om för KDevelop var filerna och informationen de tillhandahåller kan hittas.

NOT
Anpassning av automatisk komplettering beskriv i det här avsnittet i handboken.

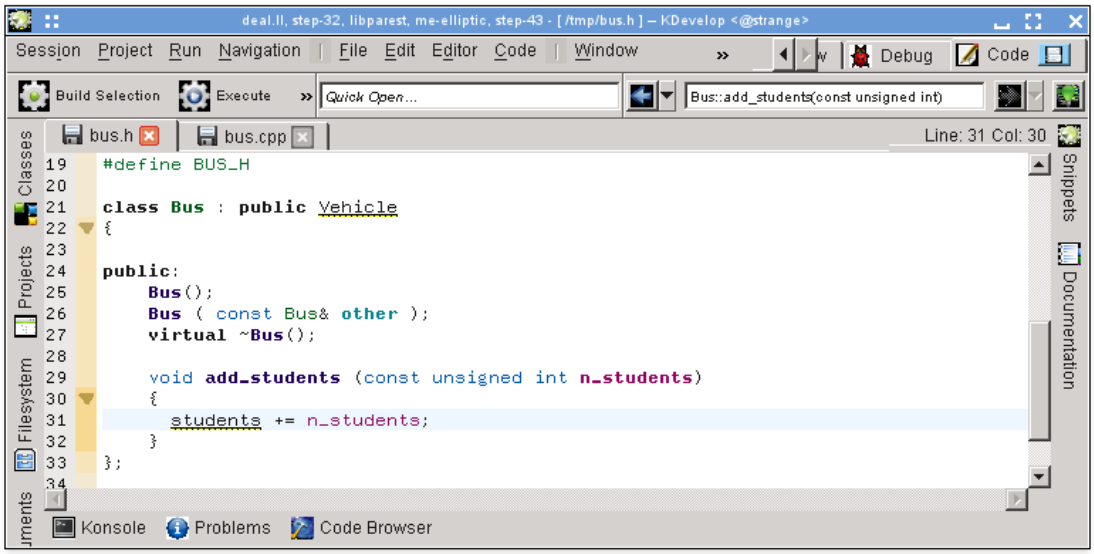
3.4.2 Lägga till nya klasser och implementera medlemsfunktioner

KDevelop har en guide för att lägga till ny klasser. Proceduren beskrivs i Skapa en ny klass. En enkel C++ klass kan skapas genom att välja den grundläggande C++ mallen från kategorin `Klass`. I guiden kan vi välja några fördefinierade medlemsfunktioner, exempelvis en tom konstruktor, en kopieringskonstruktor och en destruktor.

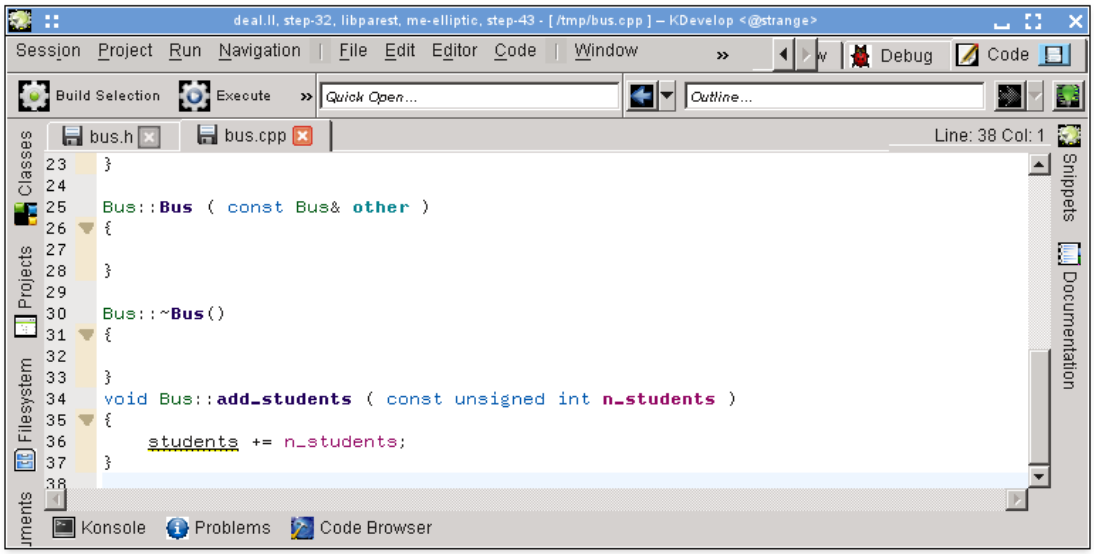
Efter att ha gjort färdigt guiden, skapas de nya filerna och öppnas i editorn. Deklarationsfilen har redan inkluderingsskydd och den nya klassen har alla medlemsfunktioner som vi valde. De följande två stegen är att dokumentera klassen och dess medlemsfunktioner och implementera dem. Vi beskriver hjälp att dokumentera klasser och funktioner nedan. För att implementera specialfunktionerna som redan lagts till, gå helt enkelt till fliken **bus.cpp** där skelett för funktionerna redan tillhandahålls:



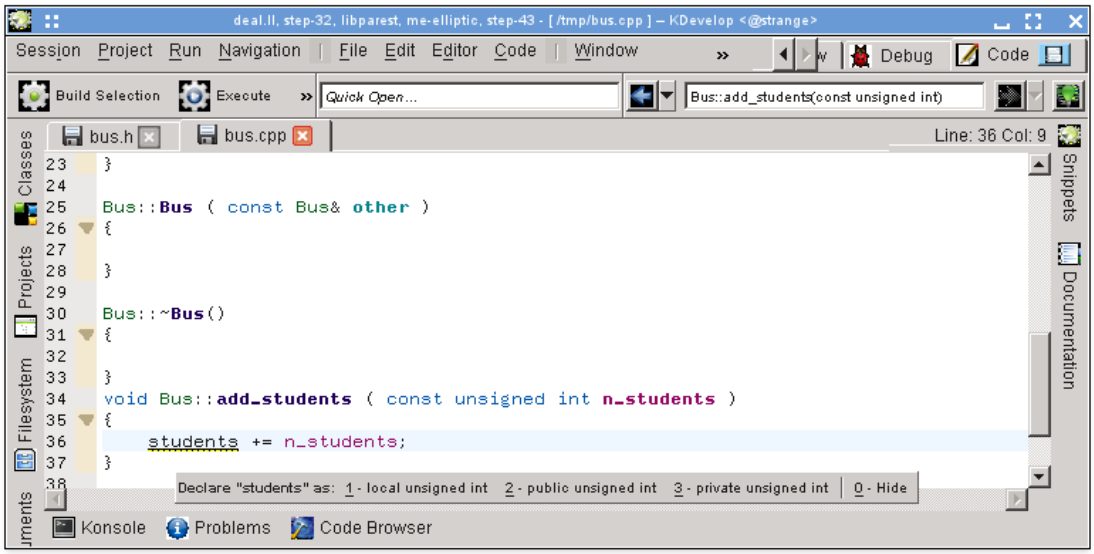
För att lägga till nya medlemsfunktioner, gå tillbaka till fliken **bus.h** och lägg till namnet på en funktion. Låt oss exempelvis lägga till det här:



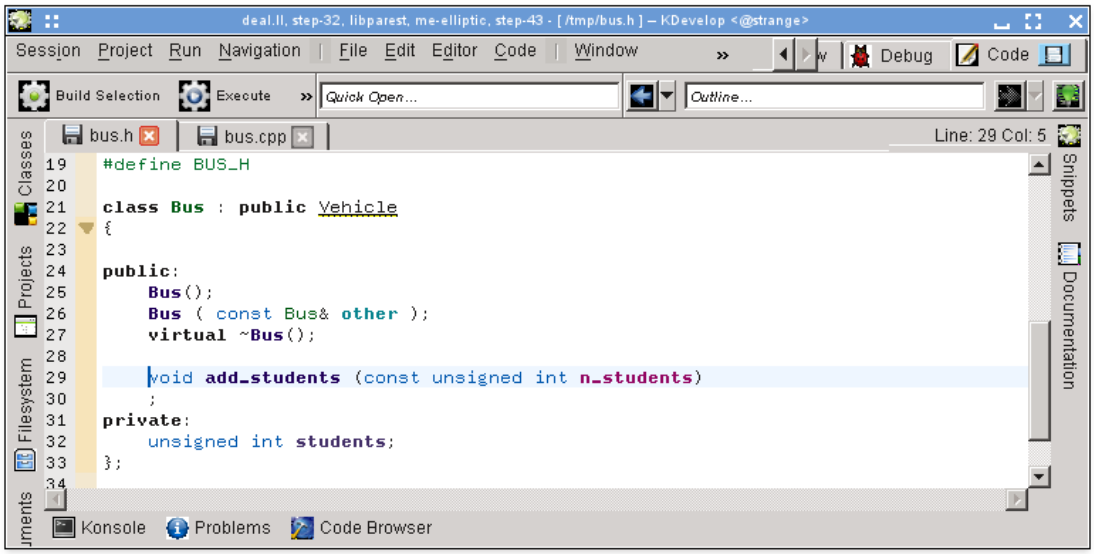
Observera hur jag redan har börjat med implementeringen. Dock ska inte funktionen implementeras i deklarationsfilen med många kodningsstilar, utan istället i den motsvarande .cpp-filen. För att åstadkomma det, placera markören på funktionsnamnet och välj **Kod** → **Flytta till källkod** eller tryck på **Ctrl-Alt-S**. Det tar bort koden mellan klammerparenteser.



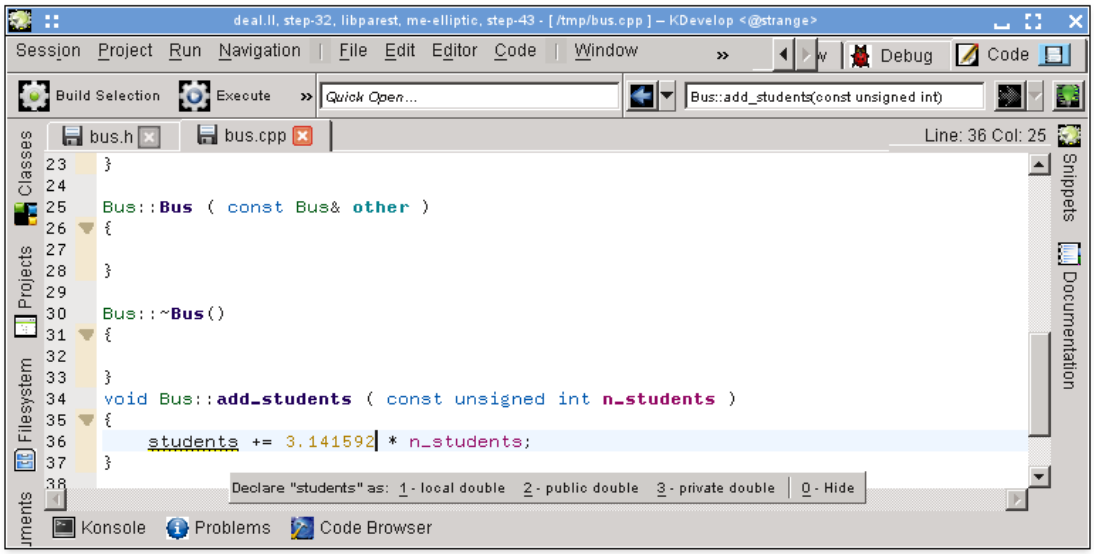
Observera hur jag precis har börjat skriva och att jag har som avsikt att variabeln `student` troligen ska vara en medlemsvariabel i klassen `Bus` men att jag inte ännu har lagt till den. Observera också hur KDevelop stryker under den för att klargöra att ingenting är känt om variabeln. Men problemet kan lösas: genom att klicka på variabelnamnet ger följande verktygstips:



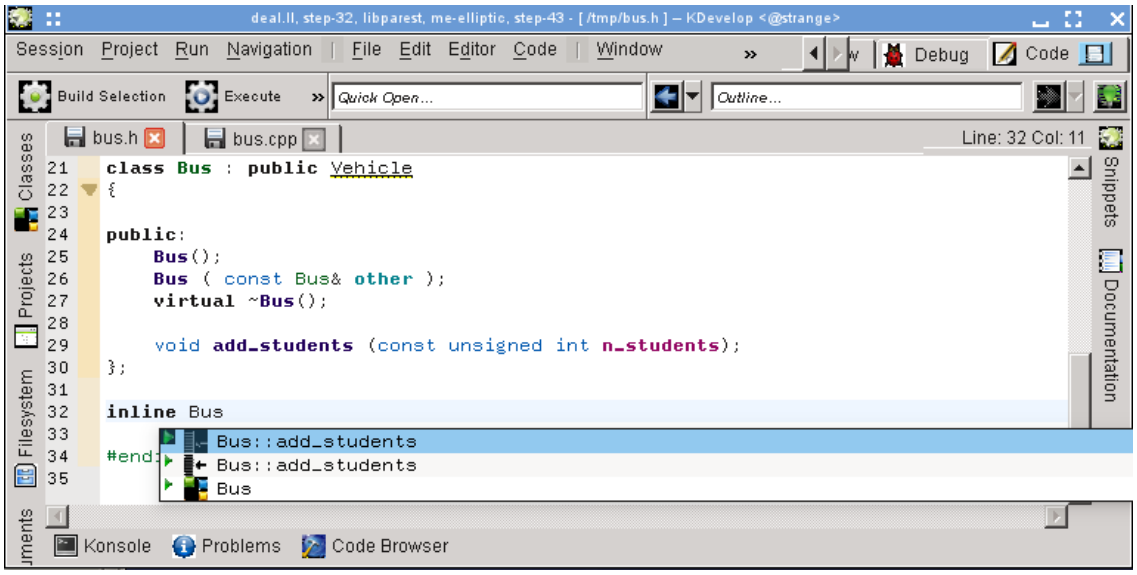
(Samma sak kan åstadkommas genom att högerklicka på det och välja **Lös: Deklarera som**.) Låt mig välja '3 - privat unsigned int' (antingen med musen eller genom att klicka på **Alt-3**) och sedan se hur det blir i deklarationsfilen:



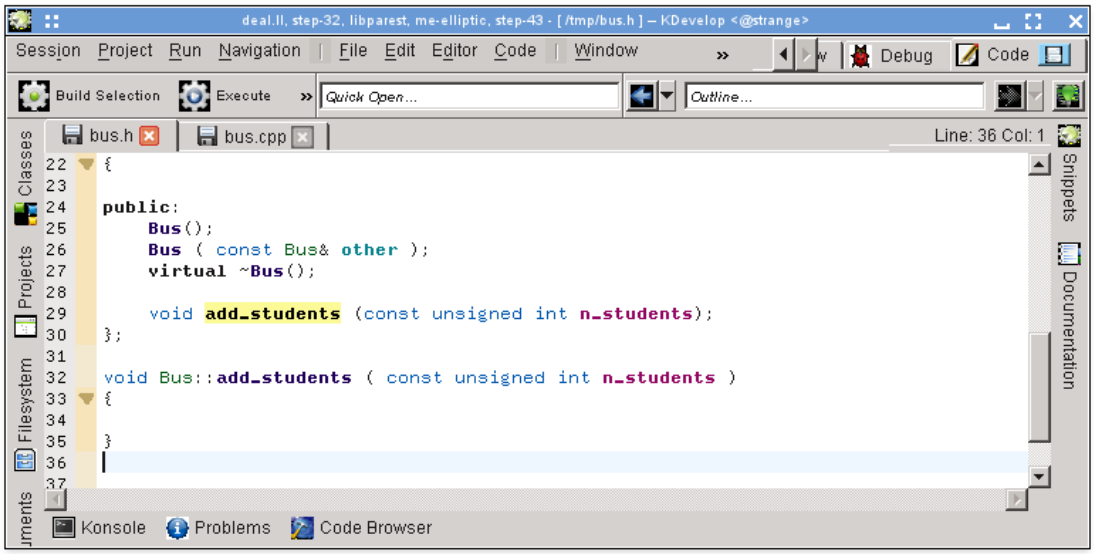
Det är värt att nämna att KDevelop extraherar typen på variabeln som ska deklareras från uttrycket som används för att initiera den. Om vi exempelvis hade skrivit additionen på följande något tveksamma sätt, skulle det ha föreslagit att deklarera variabeln som typ `double`:



Som en sista punkt: Metoden som används för **Kod** → **Flytta till källa** infogar inte alltid den nya medlemsfunktionen där du vill ha den. Du kanske vill markera den som `inline` och placera den längst ner i deklarationsfilen. I ett sådant fall, skriv deklarationen och börja sedan skriva funktionsdefinitionen på följande sätt:



KDevelop erbjuder automatiskt alla möjliga kompletteringar som kan finnas här. Att välja en av de två `add_students` posterna resulterar i följande kod som redan har fyllt i den fullständiga argumentlistan:



NOT
Att acceptera ett av valen som verktyget för automatisk komplettering erbjuder i exemplet ger korrekt signatur, men tar tyvärr bort markören `inline` som redan skrivits in. Det har rapporterats som KDevelop fel 274245.

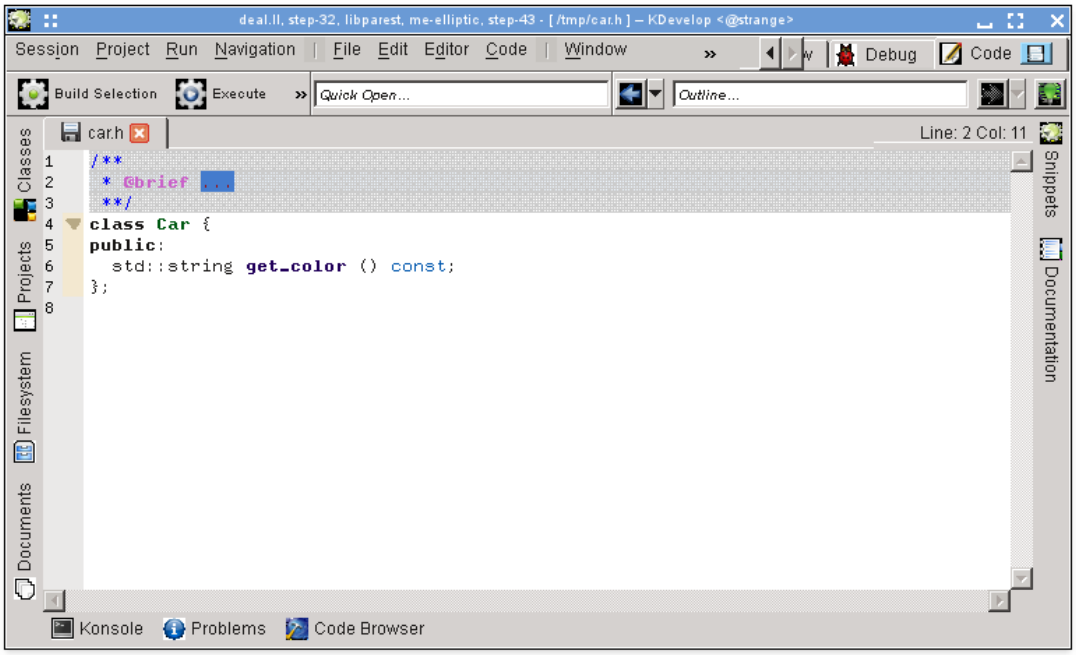
3.4.3 Dokumentera deklarationer

Bra kod är väldokumenterad, både på nivån av algoritmimplementation inom funktioner samt gränssnittsnivå, dvs. klasser, (globala och medlems-) funktioner, och (globala eller medlems-) variabler måste dokumenteras för att förklara deras avsikt, möjliga argumentvärden, för- och eftervillkor, etc. När det gäller att dokumentera gränssnittet, har doxygen blivit de facto standard för att formatera kommentarer som sedan kan extraheras och visas på sökbara webbsidor.

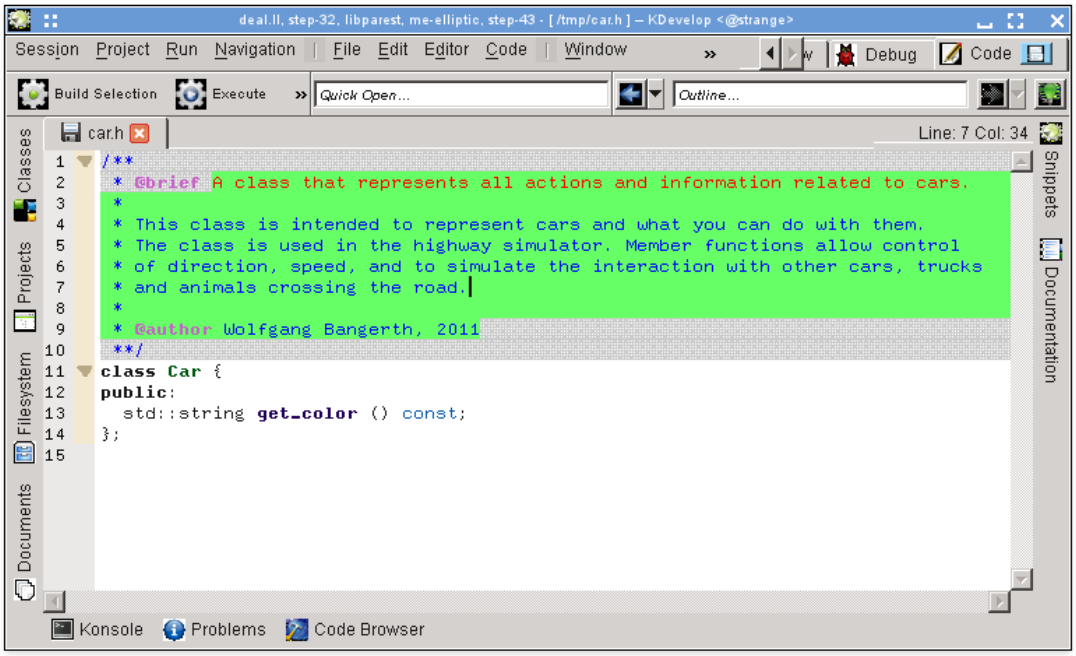
KDevelop stöder den här kommentarstilen genom att tillhandahålla en genväg för att generera kommentarramverket som dokumenterar en klass eller medlemsfunktion. Antag exempelvis att du redan har skrivit följande kod:

```
class Car {
  public:
    std::string get_color () const;
};
```

Nu vill du lägga till dokumentation för både klassen och medlemsfunktionen. För att åstadkomma det, flytta markören till första raden och välj **Kod** → **Dokumentdeklaration** eller tryck på **Alt-Skift-D**. KDevelop svarar med följande:

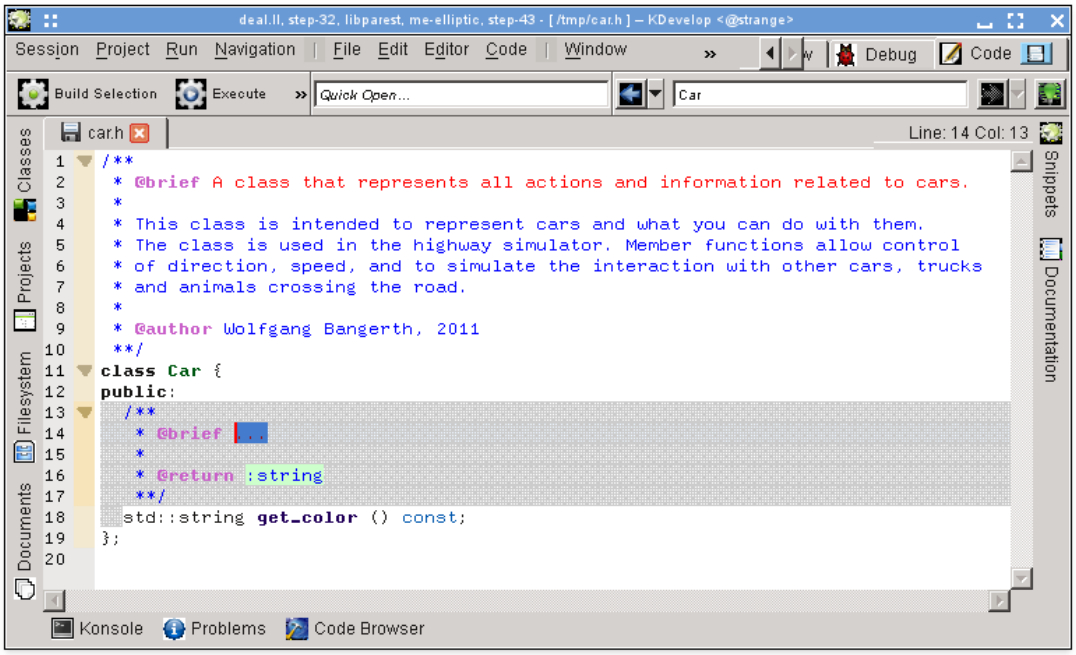


Markören är redan i det gråa området så att du kan fylla i den korta beskrivningen av klassen (efter doxygen-nyckelordet `@breif`). Därefter kan du fortsätta att lägga till dokumentation i kommentaren som ger en mer detaljerad översikt av vad klassen gör:

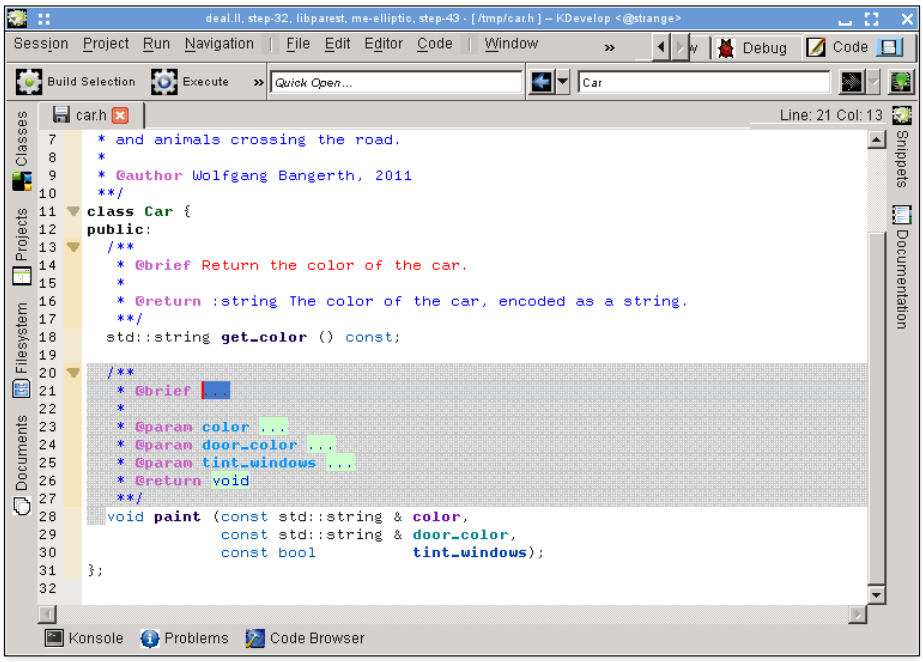


När editorn är inne i kommentaren, markeras kommentartexten med grönt (markeringen försvinner när markören flyttas från kommentaren). När du kommer till radens slut, tryck på returtangenten så påbörjar KDevelop automatiskt en ny rad som börjar med en asterisk och placerar markören indenterad ett tecken.

Låt oss nu dokumentera medlemsfunktionen, återigen genom att placera markören på deklarationsraden och välja **Kod** → **Dokumentdeklaration** eller trycka på **Alt-Skift-D**:



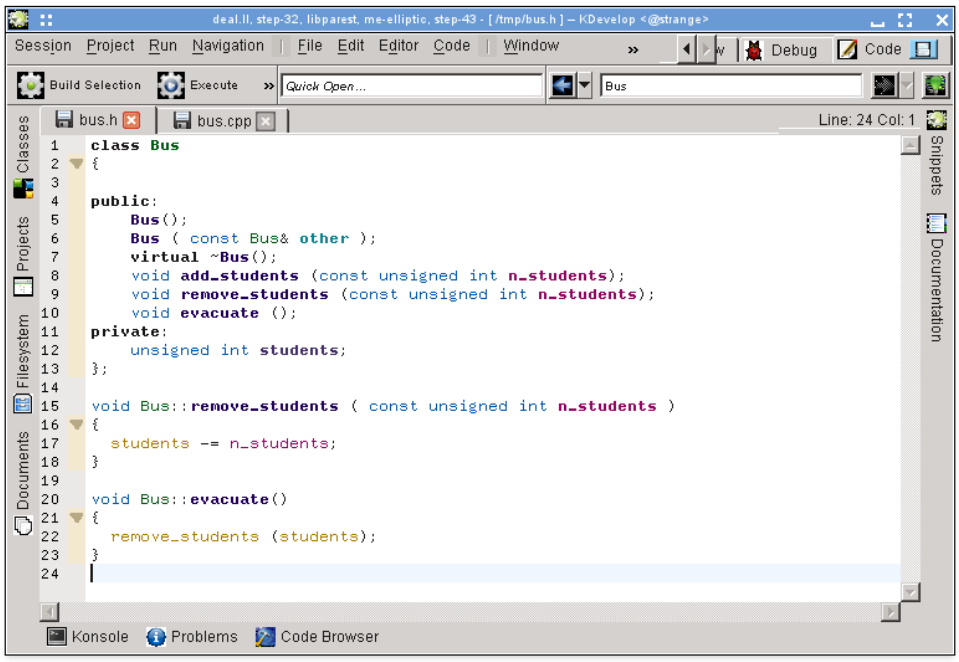
Återigen skapar KDevelop automatiskt ett kommentarmall, inklusive dokumentation av själva funktionen, samt dess returtyp. I det aktuella fallet, är själva funktionens namn rätt självförklarligt, men ofta är funktionsargumenten inte det och bör dokumenteras individuellt. För att åskådliggöra det, låt oss betrakta en något mer intressant funktion och kommentarerna som KDevelop automatiskt genererar:



Här innehåller den föreslagna kommentaren redan alla Doxygen-fält för de individuella parametrarna, exempelvis.

3.4.4 Byta namn på variabler, funktioner och klasser

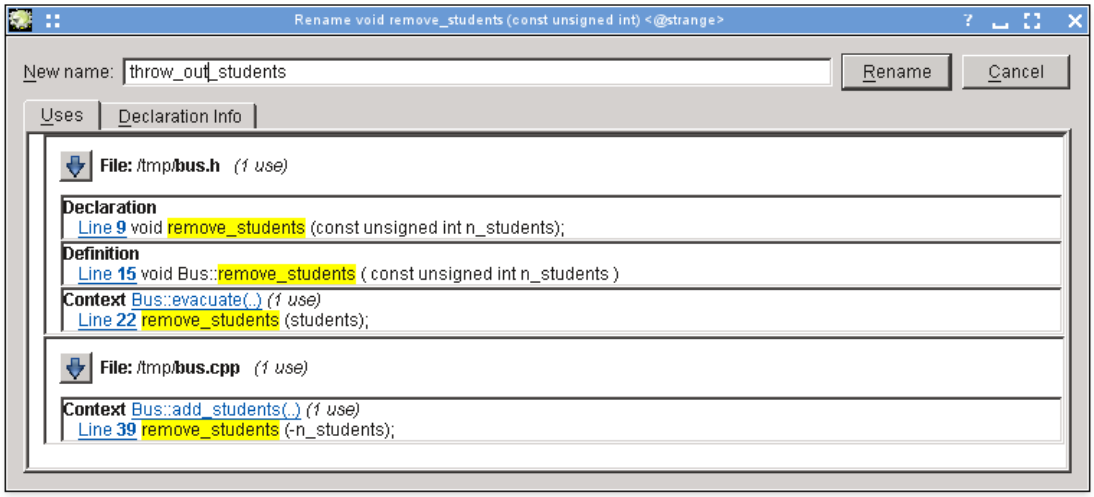
Ibland vill man byta namn på en funktion, klass eller variabel. Låt oss exempelvis anta att vi redan har det här:



Därefter inser vi att vi inte är nöjda med namnet `remove_students` och skulle hellre ha kallat det, låt oss säga, `throw_out_students`. Vi skulle kunna göra sök-och-ersätt av namnet, men det har två nackdelar:

- Funktionen kan användas i mer än en fil.
- Vi vill egentligen bara byta namn på funktionen och inte röra funktioner som kan ha samma namn men är deklarerade i andra klasser eller namnrymder.

Båda problemen kan lösas genom att flytta markören till någon av funktionsnamnets förekomster och välja **Kod** → **Byt namn på deklaration** (eller högerklicka på namnet och välja **Byt namn Bus::remove_students**). Det visar en dialogruta där du kan skriva in funktionens nya namn, och där du också kan se alla ställen där funktionen faktiskt används:

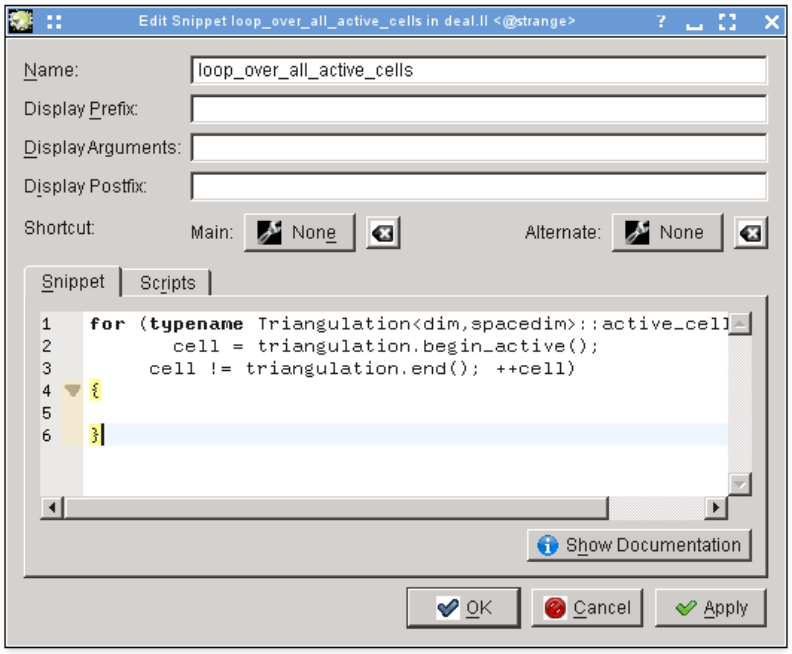


3.4.5 Kodsnuttar

De flesta projekt har kodavsnitt som man ofta måste skriva i källkoden. Exempel är: en snurra över alla instruktioner för kompilatorer, kontroll att användarinmatning är giltig och om inte visa ett felmeddelande för användargränssnitt. I upphovsmannens projekt för dessa rader, skulle det vara kod som liknar:

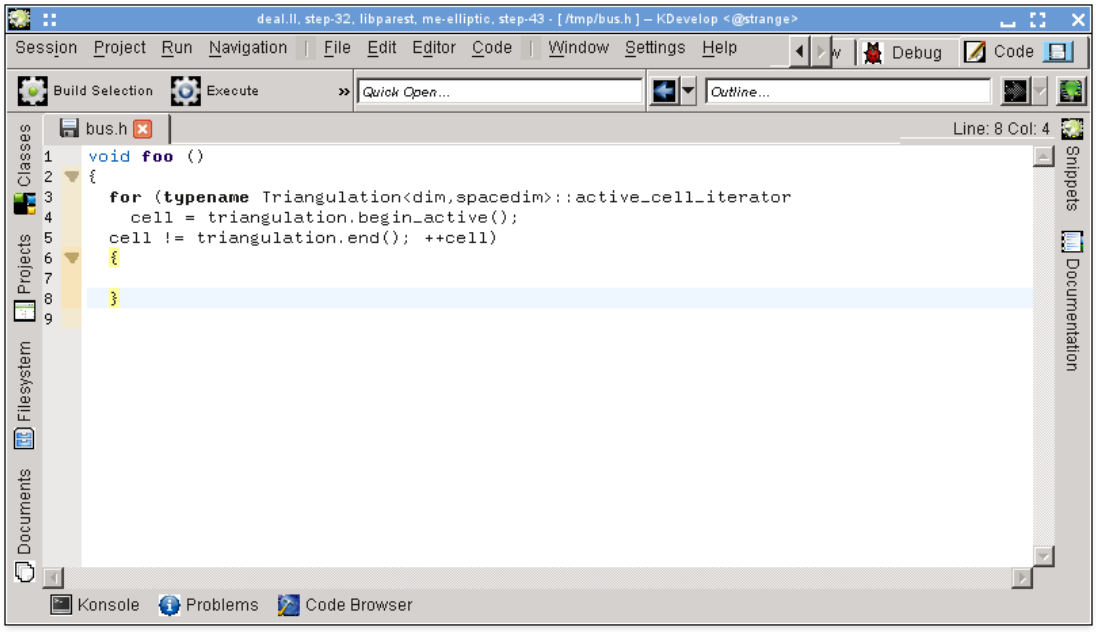
```
for (typename Triangulation::active_cell_iterator
       cell = triangulation.begin_active();
     cell != triangulation.end(); ++cell)
  ... gör något med cellen ...
```

Istället för att skriva in den här sortens text om och om igen (med alla tillhörande skrivfel man åstadkommer) kan verktyget **Textsnuttar** i KDevelop vara till hjälp. För att göra det, öppna verktygsvyn (se Verktyg och vyer om motsvarande knapp inte redan finns vid fönstrets omkrets). Klicka sedan på knappen 'Lägg till arkiv' (en något felaktig beteckning som låter dig skapa en namngiven samling textsnuttar för källkod av en viss sort, t.ex. C++ källkod) och skapa ett tomt arkiv. Klicka sedan på  för att lägga till en textsnutt, för att få en dialogruta som den följande:



> **NOT**
> Namnet på en textsnutt kan inte ha mellanslag eller andra specialtecken eftersom det måste se ut som ett normalt funktions- eller variabelnamn (av orsaker som klargörs i nästa stycke).

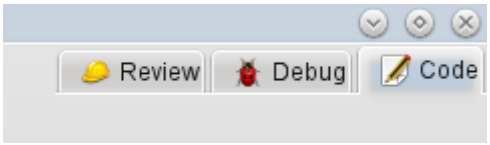
För att använda en textsnutt som är definierad på så sätt när du redigerar kod, kan du bara skriva in namnet på textsnutten som du skulle göra med vilket annat funktions- eller variabelnamn. Namnet blir tillgängligt för automatisk komplettering, vilket betyder att det inte skadar att använda ett långt och beskrivande namn på en textsnutt såsom den ovan, och när du accepterar förslaget från verktygstipset för automatisk komplettering (exempelvis genom att bara trycka på returtangenten), ersätts den redan inmatade delen av textsnuttens namn med hela den expanderade textsnutten, och indenteras riktigt:



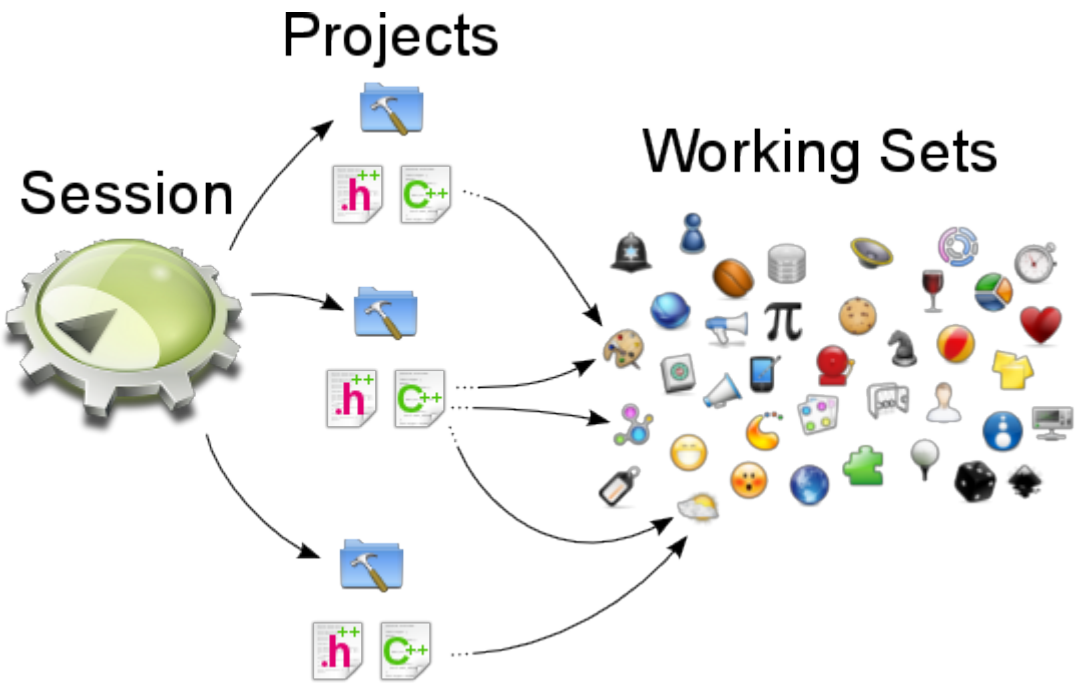
Observera att för att det ska fungera behöver inte verktygsvyn **Kodsnuttar** vara öppen eller synlig: du behöver bara verktygsvyn för att definiera nya kodsnuttar. Som ett alternativt, mindre bekvämt, sätt att expandera en kodsnutt är att helt enkelt klicka på den i respektive verktygsvy.

> NOT
> Textsnuttar är mycket kraftfullare än som vad just förklarats. För en fullständig beskrivning av vad du kan göra med dem, se den detaljerad dokumentationen av verktyget Textsnuttar.

3.5 Lägen och arbetsuppsättningar

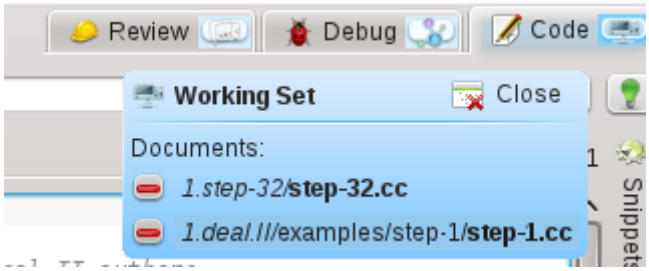


Om du har kommit så här långt, ta en titt längst upp till höger i KDevelops huvudfönster: Som visas i bilden, ser du att det finns tre **lägen** som KDevelop kan vara i: **Kod** (läget som vi beskrivit i det nuvarande kapitlet om att arbeta med källkod), **Avlusa** (se Avlusa program) och **Granska** (se Arbeta med versionskontrollsystem).



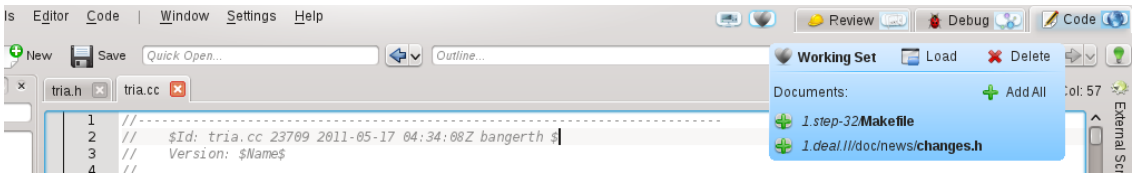
Varje läge har sin egen uppsättning verktyg som radas upp längs omkretsen, och varje läge har också en *arbetsuppsättning* av filer och dokument som för närvarande är öppna. Dessutom är varje sådan arbetsuppsättning kopplad till en aktuell session, dvs. vi har förhållandet som visas ovan. Observera att filerna i arbetsuppsättningen kommer från samma session, men de kan komma från olika projekt som ingår i samma session.

Om du öppnar KDevelop för första gången är arbetsuppsättningen tom: Det finns inte några öppna filer. Men medan du öppnar filer för redigering (eller avlusning, eller granskning i de andra lägena) växer arbetsuppsättningen. Det faktum att arbetsuppsättningen inte är tom anges av en symbol i fliken, som visas nedan. Du märker att så snart du stänger KDevelop och senare startar det igen, sparas och återställs arbetsuppsättningen, dvs. du får samma uppsättning öppna filer.



Om du håller musen över arbetsuppsättningens symbol, får du ett verktygstips som visar vilka filer som för närvarande är öppna i arbetsuppsättningen (här filerna `step-32.cc` and `step-1.cc`). Att klicka på det röda minustecknet stänger fliken för motsvarande fil. Kanske viktigare, att klicka på motsvarande namngivna knapp låter dig **stänga** hela arbetsuppsättningen på en gång (dvs, stänga alla filer som för närvarande är öppna). Vitsen med att stänga en arbetsuppsättning är dock att det inte bara stänger alla filer, utan faktiskt sparar arbetsuppsättningen och öppnar en ny, fortfarande tom. Du kan se det här:

Observera att de två symbolerna till vänster om de tre flikarna (hjärtat och den oidentifierbara symbolen till vänster om det). Var och en av de två symbolerna representerar en sparad arbetsuppsättning, förutom den nuvarande öppnade arbetsuppsättningen. Om du håller musen över hjärtsymbolen, får du något som liknar detta:



Det visar dig att motsvarande arbetsuppsättning innehåller två filer och deras motsvarande projektnamn: `Makefile` och `changes.h`. Att klicka på **Läs in** stänger och sparar den aktuella arbetsuppsättningen (som den visas här är filerna `tria.h` and `tria.cc` öppnade) och istället öppna den valda arbetsuppsättningen. Du kan också ta bort en arbetsuppsättning permanent, vilket tar bort den från de sparade arbetsuppsättningarna.

3.6 Några användbara snabbtangenter

KDevelops editor följer de vanliga snabbtangenterna för alla vanliga redigeringsåtgärderna. Dock stöder det också ett antal mer avancerade åtgärderna när källkod redigeras, där vissa är kopplade till en viss tangentkombination. De följande är ofta särskilt användbara:

Gå runt i koden	
Ctrl-Alt-O	Snabböppna fil: Ange en del av ett filnamn och välj bland alla filer i den aktuella sessionens projektkatalogträd som matchar strängen. Filen öppnas då.
Ctrl-Alt-C	Snabböppna klass: Ange en del av ett klassnamn och välj bland alla klassnamn som matchar. Markören går då till klassdeklarationen.
Ctrl-Alt-M	Snabböppna funktion: Ange en del av ett (medlems)funktionsnamn och välj bland alla namn som matchar. Observera att listan visar både deklarationer och definitioner och att markören går till det valda objektet
Ctrl-Alt-Q	Generell snabböppning: Skriv in vad som helst (filnamn, klassnamn, funktionsnamn) för att få en lista av allting som matchar att välja bland
Ctrl-Alt-N	Översikt: Tillhandahåll en lista över allt som finns i filen, t.ex. klassdeklarationer och funktionsdefinitioner
Ctrl-,	Gå till definitionen av en funktion om markören för närvarande befinner sig på en funktionsdeklaration
Ctrl-.	Gå till deklarationen av en funktion eller variabel om markören för närvarande befinner sig på en funktionsdefinition
Ctrl-Alt-Page Down	Gå till nästa funktion

Ctrl-Alt-Page Up	Gå till föregående funktion
Ctrl-G	Gå till rad

Sök och ersätt	
Ctrl-F	Sök
F3	Sök igen
Ctrl-R	Ersätt
Ctrl-Alt-F	Sök och ersätt i flera filer

Andra saker	
Ctrl-_	Dra ihop en nivå: Tar bort blocket från visning, om du till exempel vill fokusera på helheten i en funktion
Ctrl-+	Expandera en nivå: ångra sammandragning
Ctrl-D	Kommentera bort markerad text eller aktuell rad
Ctrl-Skift-D	Kommentera i markerad text eller på aktuell rad
Alt-Skift-D	Dokumentera aktuell funktion. Om markören är på en funktions- eller klassdeklaration skapas en kommentar med doxygen-stil förberedd med en lista över alla parametrar, returvärden, etc. när tangentkombinationen används.
Ctrl-T	Byt aktuellt och föregående tecken
Ctrl-K	Ta bort aktuell rad (observera: det är inte bara ‘ta bort härifrån till radens slut’ som i emacs)

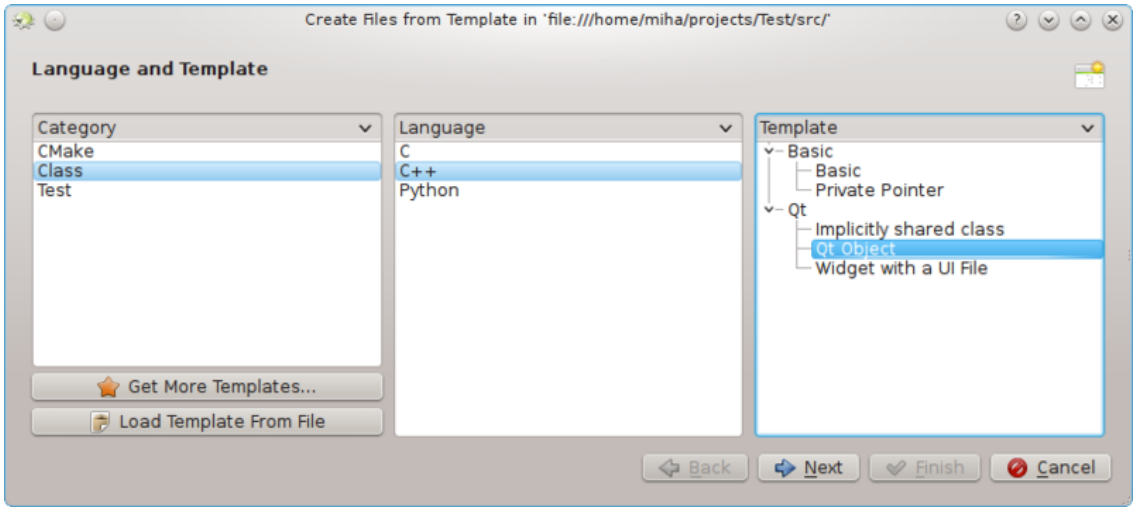
Kapitel 4

Kodgenerering med mallar

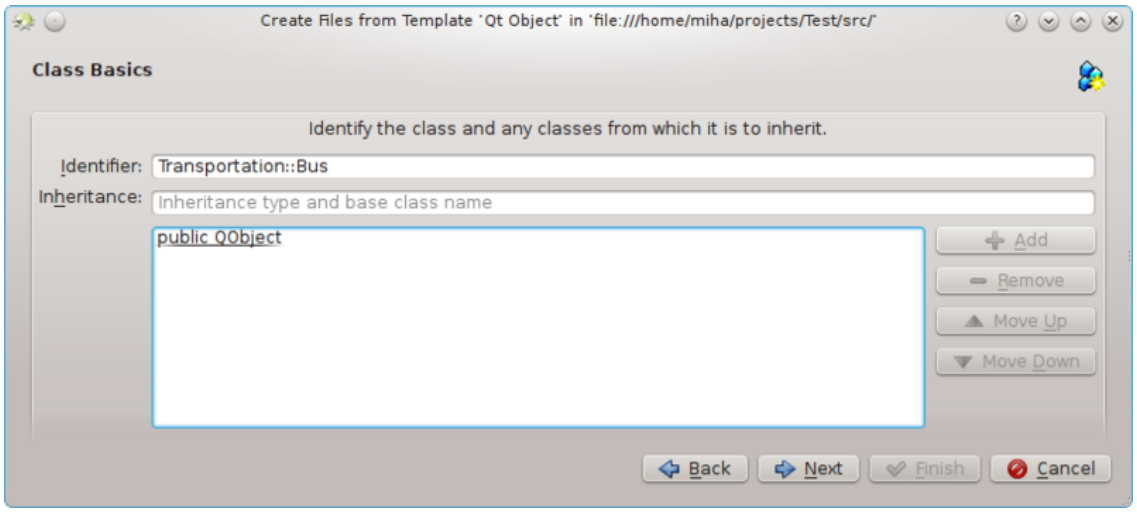
KDevelop använder mallar för att generera källkodsfiler och undvika att skriva upprepad kod.

4.1 Skapa en ny klass

Den vanligaste användningen av kodgenerering är troligen att skriva nya klasser. För att skapa en ny klass i ett befintligt projekt, högerklicka på ett projektkatalog och välj **Skapa från mall...**. Samma dialogruta kan visas från menyn genom att klicka på **Arkiv** → **Ny från mall...**, men använda en projektkatalog har fördelen att ställa in en baswebbadress för utdatafiler. Välj `Klass` i vyn för kategorival, och önskat språk och mall i de andra två vyerna. Efter du har valt en klassmall, måste du ange detaljinformation om den nya klassen.

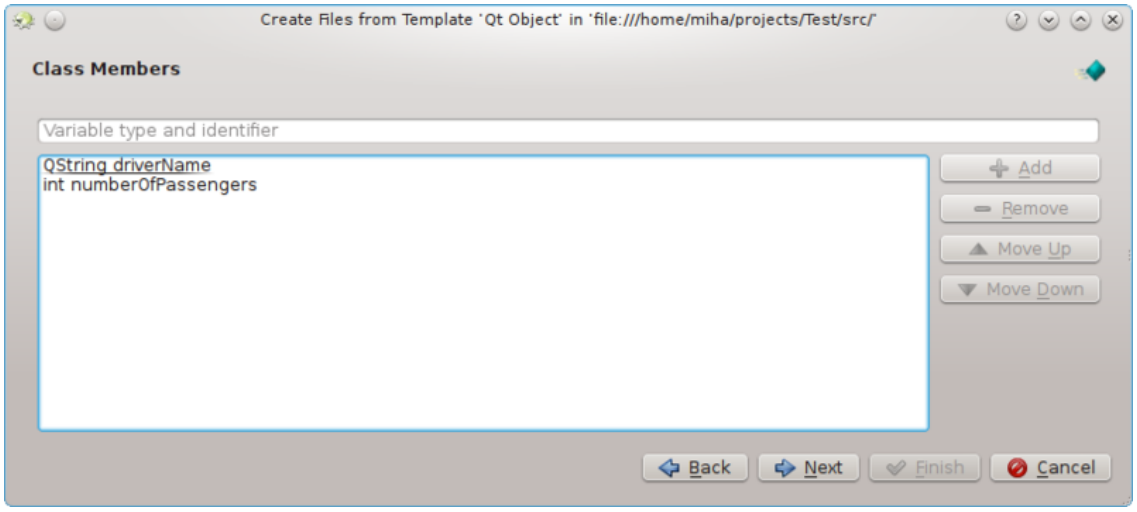


Först måste du ange en identifierare för den nya klassen. Det kan vara ett enkelt namn (som `Bus`) eller en fullständig identifierare med namnrymder (som `Transportation::Bus`). I det senare fallet tolkar KDevelop identifieraren och separerar namnrymden från själva namnet. På samma sida kan du lägga till en basklass för den nya klassen. Du kan märka att några mallar väljer en basklass själva, du har full frihet att ta bort den och/eller lägga till andra basklasser. Du bör skriva in det fullständiga arvsatsen här, vilket är språkberoende, såsom `public QObject` för C++, `extends SomeClass` för PHP eller helt enkelt klassens namn för Python.



På nästa sida erbjuds du ett urval av virtuella metoder från alla ärvda klasser, samt några standardkonstruktorer, destruktorer och operatorer. Att markera kryssrutan intill signaturen för en metod implementerar den metoden i den nya klassen.

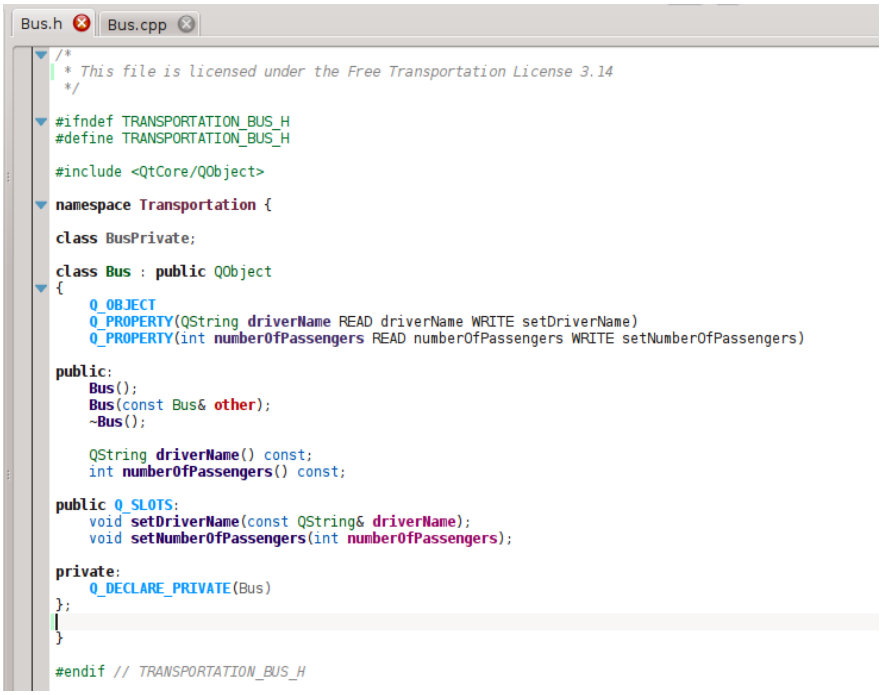
Att klicka på **Nästa** visar en sida där du kan lägga till medlemmar i en klass. Beroende på den valda mallen, kan de visas i den nya klassen som medlemsvariabler, eller så kan mallen skapa egenskaper med tilldelnings- och hämtningsfunktioner för dem. I ett språk där variabeltyper måste deklareras, såsom C++, måste du ange både typen och namnet på medlemmen, såsom `int number` eller `QString name`. För andra språk kan du utelämna typen, men det är bra att ange den ändå, eftersom den valda mallen kan dra nytta av det.



På följande sidor kan du välja en licens för den nya klassen, ställa in eventuella egna alternativ som krävs av den valda mallen, och anpassa utdataplatser för alla genererade filer. Genom att klicka på **Slutför**, avslutas guiden och den nya klassen skapas. De genererade filerna öppnas i editorn, så att du direkt kan börja lägga till kod.

Efter att ha skapat en ny C++ klass, får du alternativet att lägga till klassen i ett projektmål. Välj ett mål från dialogsidan, eller stäng sidan och lägg till filerna manuellt i ett mål.

Om du valde mallen `Qt-objekt`, markerade några av standardmetoderna, och lade till två medlemsvariabler, ska utmatningen se ut som på följande bild.



Du kan se att datamedlemmar konverteras till Qt-egenskaper, med åtkomstfunktioner och Q_PROPERTY makron. Argument för tilldelningsfunktioner skickas till och med som konstantreferenser där det är lämpligt. Dessutom deklareras en privat klass, och en privat pekare skapas med Q_DECLARE_PRIVATE. Allt det görs av mallen, att välja en annan mall i det första steget kan ändra utdata fullständigt.

4.2 Skapa en ny enhetstest

Även om de flesta testramverk kräver att varje test också är en klass, inkluderar KDevelop en metod för att förenkla att skapa enhetstester. För att skapa en ny test, högerklicka på en projektkatalog och välj **Skapa från mall...**. På sida för mallval, välj `Test` som kategorin, välj sedan programspråket och mallen och klicka på **Nästa**.

Du blir tillfrågad om testnamnet och en lista med testfall. För testfallen behöver du bara specificera en lista med namn. Vissa ramverk för enhetstest, såsom PyUnit och PHPUnit, kräver att testfallen börjar med ett särskilt prefix. I KDevelop är mallen ansvarig för att lägga till prefixet, så du behöver inte använda något för testfallen här. Efter att ha klickat på **Nästa**, ange licens och utdataplatser för de genererade filerna, så skapas testen.

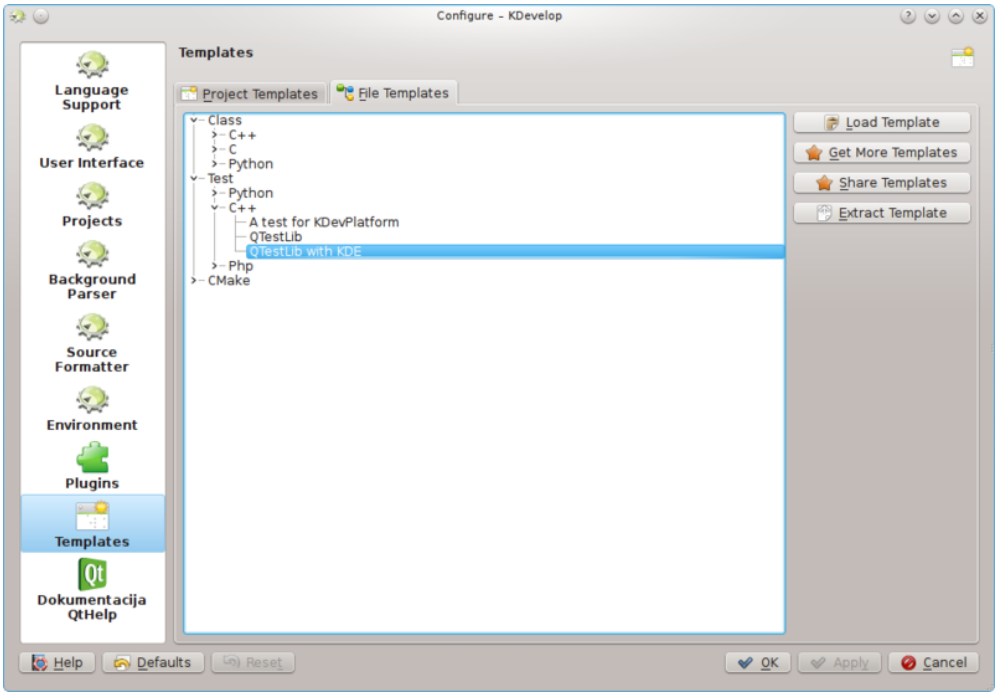
Enhetstester skapade på det här sättet läggs inte automatiskt till i något mål. Om du använder CTest eller något annat testramverk, säkerställ att nya filer läggs till i ett mål.

4.3 Andra filer

Medan klasser och enhetstester får särskild uppmärksamhet när kod genereras från mallar, kan samma metod användas för vilken sorts källkodsfiler som helst. Exempelvis kan man använda en mall för en CMake sökmodul eller en .desktop-fil. Det kan göras genom att välja **Skapa från mall...** och välja önskad kategori och mall. Om den valda kategorin varken är `Class` eller `Test`, får du bara alternativet att välja licens, eventuella egna alternativ angivna av mallen, och utdatafilens plats. Liksom med klasser och tester, att avsluta guiden genererar filerna och öppnar dem i editorn.

4.4 Hantera mallar

Från guiden **Arkiv** → **Ny från mall...**, kan du också ladda ner ytterligare filmallar genom att klicka på knappen **Hämta fler mallar...**. Det visar dialogrutan Hämta heta nyheter, där du kan installera ytterligare mallar, samt uppdatera eller ta bort dem. Det finns också en inställningsmodul för mallar, som kan nås genom att klicka på **Inställningar** → **Anpassa KDevelop** → **Mallar**. Härifrån kan du både hantera filmallar (förklarade ovan) och projektmallar (använda för att skapa nya projekt).



Om ingen av de tillgängliga mallarna passar ditt projekt, kan du naturligtvis skapa nya. Det enklaste sättet är troligen att kopiera och ändra en befintlig mall, där en kort handledning och ett längre specifikationsdokument är tillgängliga som hjälp. För att kopiera en installerad mall, öppna mallhanteraren genom att klicka på **Inställningar** → **Anpassa KDevelop...** → **Mallar**, välj mallen du vill kopiera, klicka sedan på knappen **Extrahera mall**. Välj en målkatalog, klicka sedan på **Ok**, så extraheras mallens innehåll till den valda katalogen. Nu kan du redigera mallen genom att öppna de extraherade filerna och ändra dem. När du är klar, kan du importera den nya mallen till KDevelop genom att öppna mallhanterarna, aktivera lämplig flik (antingen **Projektmallar** eller **Filmallar**) och klicka på **Läs in mall**. Öppna mallbeskrivningfilen, som är den med filändelsen `.kdevtemplate` eller `.desktop`. KDevelop komprimerar filerna till ett mallarkiv och importerar mallen.

> NOT
> När en befintlig mall kopieras, säkerställ att du byter namn på den innan den importeras igen. Annars skriver du antingen över den gamla mallen, eller slutar med två mallar med identiska namn. För att byta namn på en mall, ändra beskrivningsfilen till någonting unikt (men behåll filändelsen), och ändra posten `Nam` i beskrivningsfilen.

Om du vill skriva en mall från början, kan du starta med ett exempel på en C++ klassmall genom att skapa ett nytt projekt och välja projektet `C++ klassmall` under kategorin `KDevelop`.

Kapitel 5

Bygga (kompilera) projekt med egen Makefile

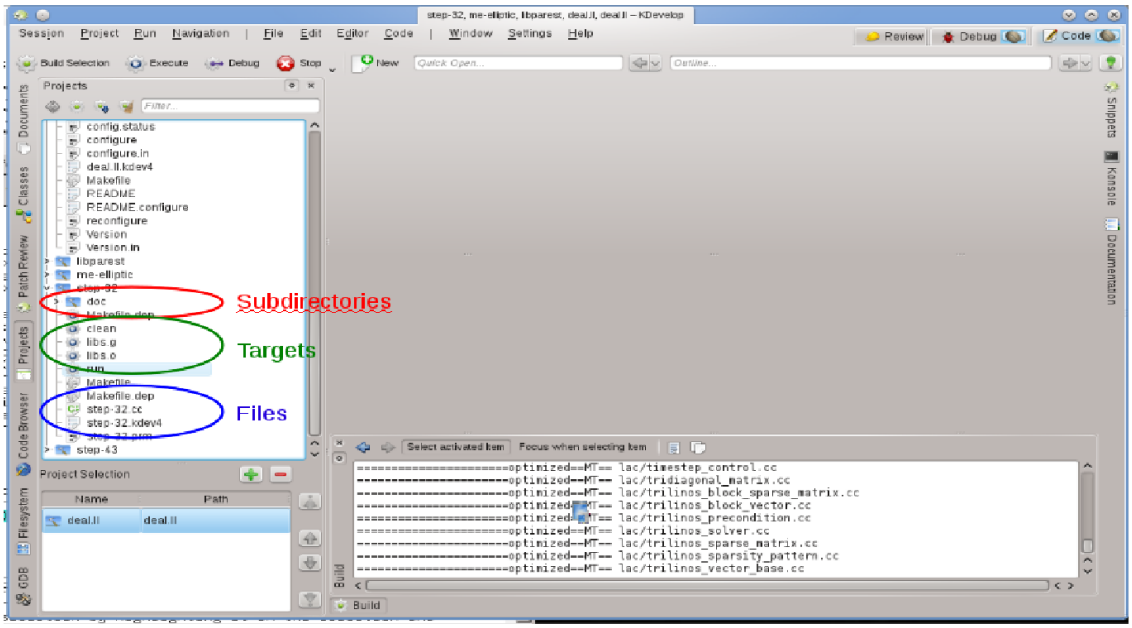
Många projekt beskriver hur källkodsfiler ska kompileras (och vilka filer som måste kompileras om när en källkodsfil eller deklarationsfil ändras) genom att använda en Makefile som tolkas av programmet **make** (se exempelvis GNU make). För enkla projekt, är det ofta mycket enkelt att skapa en sådan fil för hand. Större projekt integrerar sin Makefile med **GNU autotools** (autoconf, autoheader, automake). Låt oss helt enkelt anta i det här avsnittet att du har en Makefile för projektet och vill lära KDevelop hur det ska använda sig av den.

> NOT
> KDevelop 4.x känner inte till **GNU autotools** när det här avsnittet skrevs. Om projektet använder dem, måste du köra `./configure` eller något av de andra relaterade kommandona för hand på en kommandorad. Om du vill göra det inne i KDevelop, öppna verktyget **Terminal** (om det behövs, lägg till det i huvudfönstrets omkrets genom att använda menyn **Fönster** → **Lägg till verktygsvy**), som ger dig ett skalfönster, och kör `./configure` från kommandoraden i den vyn.

Det första steget är att lära KDevelop om målen i din Makefile. Det finns två sätt att göra det: välja individuella Makefile-mål, och välja en uppsättning mål som du ofta vill bygga. För båda sätten, öppna verktyget **Projekt** genom att klicka på knappen **Projekt** i omkretsen av KDevelops huvudfönster (om du inte har knappen, se ovan hur man lägger till knappen för ett verktyg där). Verktygsvyn **Projekt** har två delar: den övre halvan, benämnd **Projekt**, listar alla projekt och låter dig expandera de underliggande katalogträden, den undre halvan, benämnd **Byggföljd**, listar en delmängd av projekten som byggs om du använder menyalternativet **Projekt** → **Bygg valda** eller tryck på **F8**. Vi återkommer till den delen nedan.

5.1 Bygga enskilda mål i en Makefile

I den övre delen av projektvyn, expandera delträdet för ett projekt, låt oss säga det där du vill köra ett specifikt Makefile-mål. Det ger dig ikoner för (i) kataloger i projektet, (ii) filer i projektets toppnivåkatalog, (iii) Makefile-mål som KDevelop kan identifiera. Kategorierna visas på bilden till höger. Observera att KDevelop *förstår* syntaxen i en Makefile (även om förståelsen har sina begränsningar om målen är sammansatta eller implicita).



För att bygga något av målen som listas här, klicka på det med höger musknapp och välj **Bygg**. Genom att exempelvis göra det med målet ‘clean’ kör helt enkelt ‘make clean’. Du kan se det utföras i delfönstret vid namn **Bygg** som öppnas, där kommandot och utmatningen visas. (Fönstret motsvarar verktyget **Bygg**, så du kan stänga och senare öppna fönstret igen genom att använda verktygsknappen **Bygg** i huvudfönstrets omkrets. Den visas längst ner till höger på bilden.)

5.2 Välja en samling mål för en Makefile att bygga upprepade gånger

Att högerklicka på individuella Makefile-mål varje gång du vill bygga någonting blir snabbt jobbigt. Vi skulle istället vilja ha individuella mål för ett eller flera av projekten i sessionen som vi kan bygga upprepade gånger utan mycket arbete med musen. Det är här konceptet med ‘valda byggmål’ kommer in: det är en samling av Makefile-mål som byggs en i taget så fort du trycker på knappen **Bygg valda** i knapplistan längst upp, väljer menyalternativet **Projekt** → **Bygg valda**, eller trycker på funktionsknappen **F8**.

Listan med valda Makefile-mål visas i nedre halvan av verktygsvyn **Projekt**.

Normalt innehåller valet alla projekt, men du kan ändra det. Om listan över projekt exempelvis innehåller tre projekt (ett basbibliotek L och två program A och B), men du för närvarande bara arbetar på projekt A, kan du vilja ta bort projekt B från urvalet genom att markera det och trycka på knappen . Dessutom vill du troligen säkerställa att biblioteket L byggs före projekt A genom att flytta posterna i urvalet uppåt eller neråt genom att använda knapparna till höger om listan. Du kan också få ett visst Makefile-mål i urvalet genom att högerklicka på det och välja **Lägg till i bygguppsättning**, eller bara markera det och trycka på knappen  ovanför listan med valda mål.

KDevelop låter dig ställa in vad som ska göras så fort du bygger markeringen. För att göra det, använd menyalternativet **Projekt** → **Öppna konfiguration**. Där kan du exempelvis välja antal samtidiga jobb som ‘make’ ska köra. Låt oss säga att datorn har åtta processorkärnor: i så fall är det ett lämpligt val att mata in 8 i fältet. I den här dialogrutan är **Förvalt byggmål** ett Makefile-mål använt för *alla* mål i markeringen.

5.3 Vad man ska göra med felmeddelanden

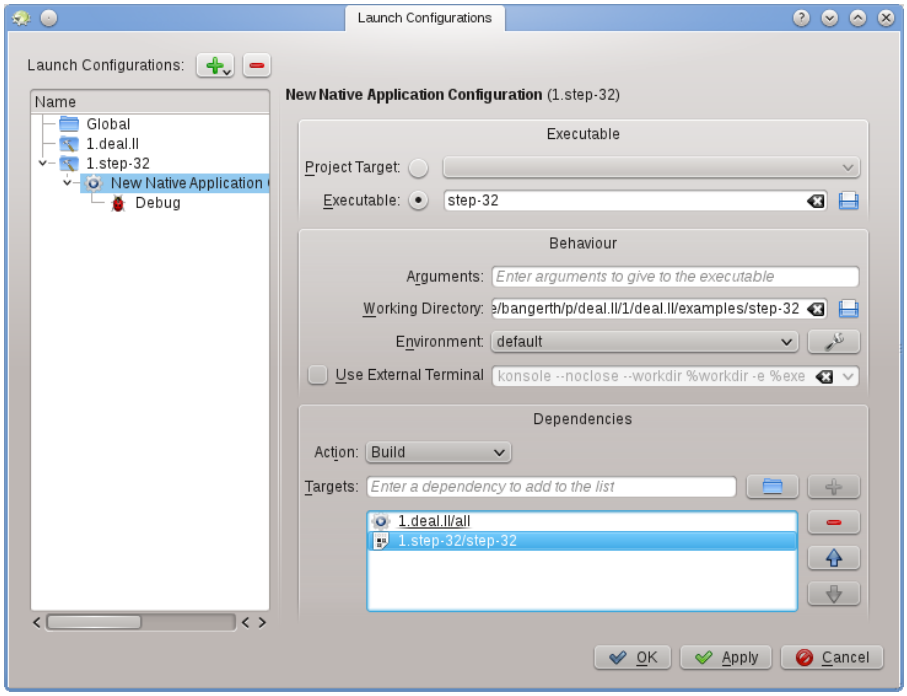
Om kompilatorn stöter på ett felmeddelande, klicka helt enkelt på raden med felmeddelandet så går editorn till raden (och om tillgängligt, kolumnen) där felet rapporterades. Beroende på felmeddelandet, kan KDevelop också erbjuda flera möjliga åtgärder för att rätta felet, exempelvis genom att deklarera en tidigare odeklarerad variabel, om en okänd symbol hittades.

Kapitel 6

Köra program i KDevelop

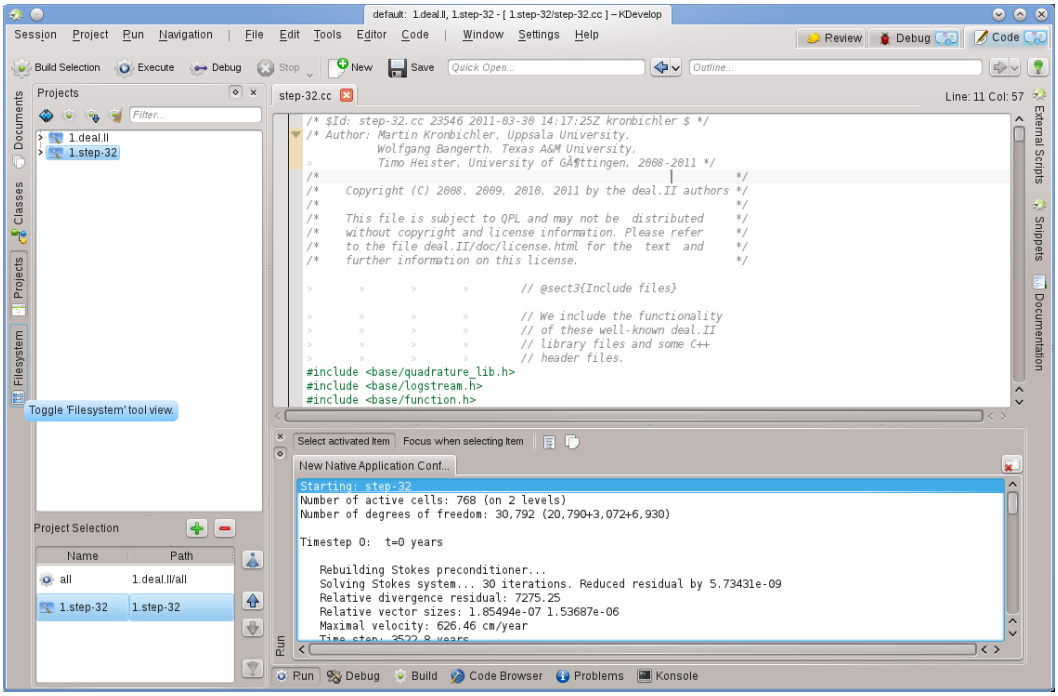
När du väl har byggt ett program vill du köra det. För att göra det, måste du ställa in *Starter* för dina projekt. En *Start* består av namnet på en körbar fil, en uppsättning kommandoradsparametrar, och en körningsmiljö (såsom 'kör programmet i ett skal', eller 'kör programmet i avlusaren').

6.1 Ställa in start i KDevelop



För att ställa in det gå till menyalternativet **Kör** → **Anpassa starter**, markera projektet som du vill lägga till en start i, och klicka på knappen  . Ange sedan namnet på den körbara filen, och sökvägen där programmet ska köras. Om att köra programmet kräver att det och/eller andra bibliotek först byggs, kan de behöva läggas till i listan längst ner: välj **Bygg** i kombinationsmenyn, klicka sedan på symbolen  till höger om textrutan och välj vilket mål som du vill ska byggas. I exemplet ovan har jag valt målet **all** i projektet `1.deal.II` och `step-32` i

projektet `1.step-32` för att säkerställa att både basbiblioteket och applikationsprogrammet har kompilerats och är uppdaterade innan programmet faktiskt körs. Medan du är där, kan du lika väl också ställa in en avlusningsstart genom att klicka på symbolen **Avlusa** och lägga till namnet på avlusningsprogrammet. Om det är systemets förvalda avlusare (t.ex. gdb på Linux®) behöver inte steget utföras.



Nu kan du försöka köra programmet: Välj **Kör** → **Kör start** i KDevelops meny i huvudfönstret (eller tryck på **Skift-F9**) så ska programmet köras i ett separat delfönster i KDevelop. Bilden ovan visar resultatet: Det nya delfönstret för verktyget **Kör** längst ner visar utmatningen från programmet som körs, i det här fallet programmet `step-32`.

NOT
Om du har anpassat fler starter, kan du välja vilken som ska köras när du trycker på **Skift-F9** genom att gå till **Kör** → **Nuvarande startinställning**. Det finns dock ett mindre uppenbart sätt att redigera namnet på inställningen: dubbelklicka på inställningens namn i trädvyn till vänster i dialogrutan som du får när du väljer **Kör** → **Nuvarande startinställning**, vilket låter dig redigera inställningens namn.

6.2 Några användbara snabbtangenter

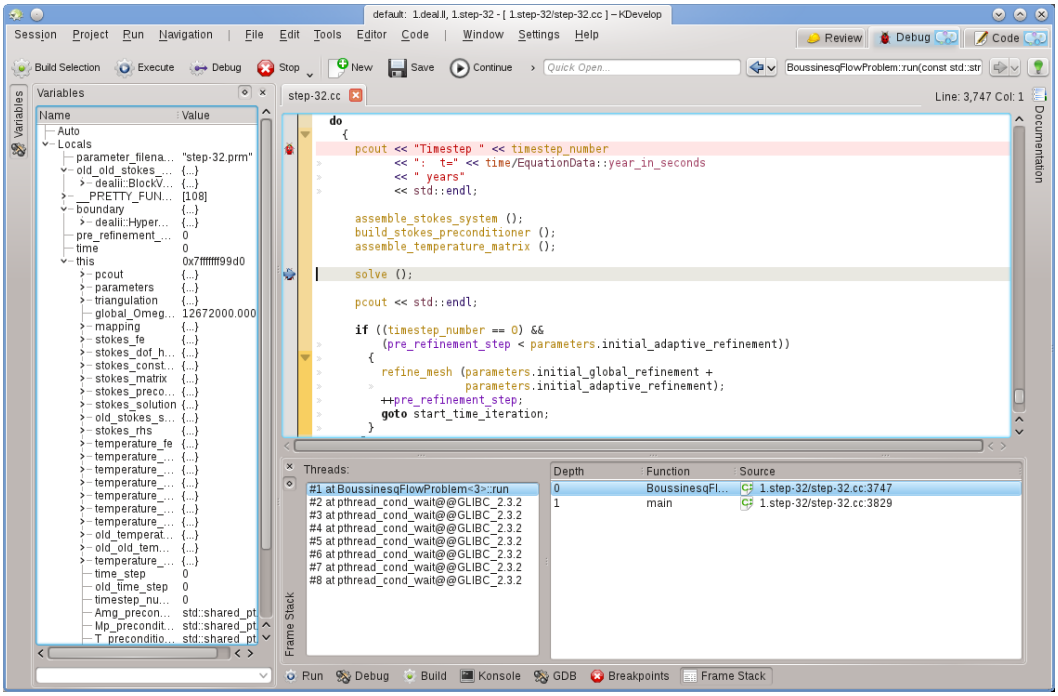
Köra ett program	
F8	Bygg (anropa make)
Skift-F9	Kör
Alt-F9	Kör program i avlusaren. Du kan vilja ställa in brytpunkter innan, exempelvis genom att högerklicka med musen på en viss rad i källkoden.

Kapitel 7

Avlusa ett program i KDevelop

7.1 Köra ett program i avlusaren

När du väl har ställt in en start (se Köra program), kan du också köra det i en avlusare: Välj menyalternativet **Kör** → **Avlusa start**, eller tryck på **Alt-F9**. Om du är bekant med gdb är effekten samma som att starta gdb med det körbara programmet angivet i startinställningen och sedan använda `run`. Det betyder att om programmet anropar `abort()` någonstans (t.ex. när du stöter på en kontroll som misslyckas) eller om ett segmenteringsfel uppstår, stoppar avlusaren. Å andra sidan, om programmet kör klart (vare sig det gör vad det ska eller inte) så stoppar inte avlusaren av sig själv innan programmet avslutas. I det senare fallet, måste du ange en brytpunkt på alla kodrader i din kodbas där du vill att avlusaren ska stoppa innan avlusaren startas. Du kan göra det genom att flytta markören till en sådan rad och välja menyalternativet **Kör** → **Växla brytpunkt**, eller högerklicka på en rad och välja **Växla brytpunkt** i den sammanhangsberoende menyn.



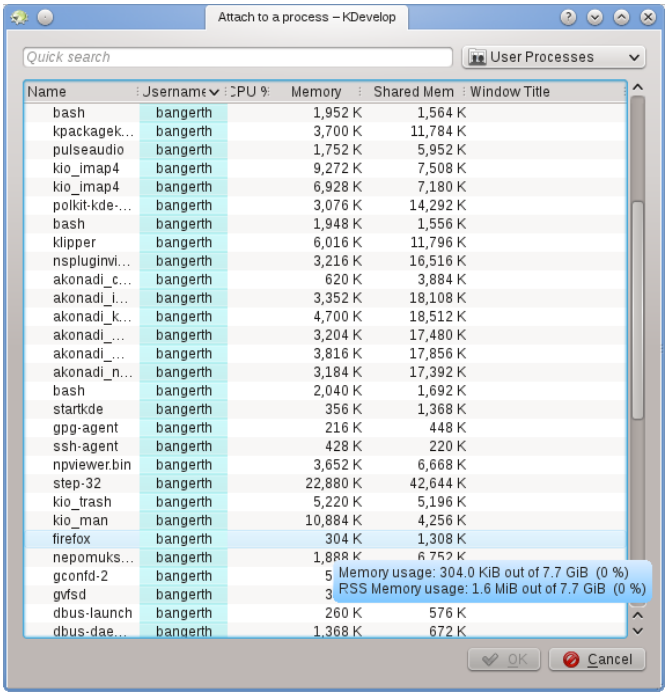
Att köra ett program i avlusaren ändrar läge i KDevelop: det ersätter alla ‘verktygsknappar’

längs huvudfönstrets omkrets med sådana som är lämpliga för avlusning, istället för redigering. Du kan se vilket läge som används genom att titta längst upp till höger i fönstret: det finns flikar som kallas **Granskning**. **Avlusning** och **Kod**. Genom att klicka på dem är det möjligt att byta fram och tillbaka mellan de tre lägena. Varje läge har en uppsättning egna verktygsvyer, som du kan ställa in på samma sätt som vi har ställt in verktygen för **Kod** i avsnittet Verktyg och vyer.

När avlusaren väl stoppar (vid en brytpunkt eller en punkt där `abort()` anropas) kan du inspektera en mängd olika information om programmet. På bilden ovan har vi exempelvis valt verktyget **Aktiveringspoststack** längst ner (som ungefär motsvarar kommandona 'backtrace' och 'info threads' i gdb) som visar de olika trådarna som för närvarande kör i programmet till vänster (här är det totalt åtta) och hur körningen har kommit till den aktuella stoppunkten till höger (här: `main()` anropade `run()`, listan skulle vara längre om vi hade stoppat i en funktion som anropas av `run()` själv). Till vänster kan vi inspektera lokala variabler inklusive det aktuella objektet (objektet som pekas ut av variabeln `this`).

Härifrån finns det olika möjligheter som du kan välja: Du kan köra den aktuella raden (**F10**, kommandot 'next' i gdb), stega in i funktionen (**F11**, kommandot 'step' i gdb), eller köra till funktionens slut (**F12**, kommandot 'finish' i gdb). Efter varje steg uppdaterar KDevelop variablerna som visas till vänster till deras aktuella värden. Du kan också hålla musen över en symbol i koden, t.ex. en variabel. Då visar KDevelop symbolens aktuella värde och erbjuder att stoppa programmets körning nästa gång variabelns värde ändras. Om du känner till gdb, kan du också klicka på verktygsknappen **GDB** längst ner, och få möjlighet att skriva in kommandon i gdb, exempelvis för att ändra värdet på en variabel (som inte för närvarande verkar gå att göra på något annat sätt).

7.2 Ansluter avlusaren till en process som kör



Ibland vill man avlusa ett program som redan kör. Ett scenario för det är avlusning av parallella program genom att använda MPI, eller för avlusning av en bakgrundsprocess som kört länge. För att göra det, gå till menyalternativet **Kör** → **Anslut till process**, vilket öppnar ett fönster som det ovan. Du måste välja programmet som motsvarar projektet som för närvarande är öppet i KDevelop: i mitt fall är det programmet step-32.

Listan över program kan vara förvirrande eftersom den ofta är lång som visas här. Du kan göra livet enklare genom att gå till kombinationsrutan längst upp till höger i fönstret. Förvalt värde är **Användarprocesser**, dvs. alla program som körs av någon av användarna för närvarande inloggade på datorn (om det är din skrivbordsdator eller bärbara dator, är du troligen den enda sådana användare, förutom root och diverse tjänstkonton). Listan omfattar dock inte processer som körs av användaren root. Du kan begränsa listan genom att antingen välja **Egna processer**, vilket tar bort alla program som körs av andra användare. Eller ännu bättre, välj **Bara program**, som tar bort många processorer som formellt kör under ditt namn men som du oftast inte interagerar med, såsom fönsterhanteraren, bakgrundsaktiviteter och så vidare, som är mindre troliga som avlusningskandidater.

När du väl har valt en process, kommer du till KDevelops avlusningsläge när du ansluter till den, och alla vanliga verktygsvyer för avlusning öppnas, och programmet stoppas där det råkar vara när du anslöt till det. Därefter kan du ställa in brytpunkter, visningspunkter, eller vad som helst annat som är nödvändigt, och fortsätta köra programmet genom att gå till menyalternativet **Kör** → **Fortsätt**.

7.3 Några användbara snabbtangenter

Avlusning	
F10	Stega över (‘next’ i gdb)
F11	Stega in i (‘step’ i gdb)
F12	Stega ut ur (‘finish’ i gdb)

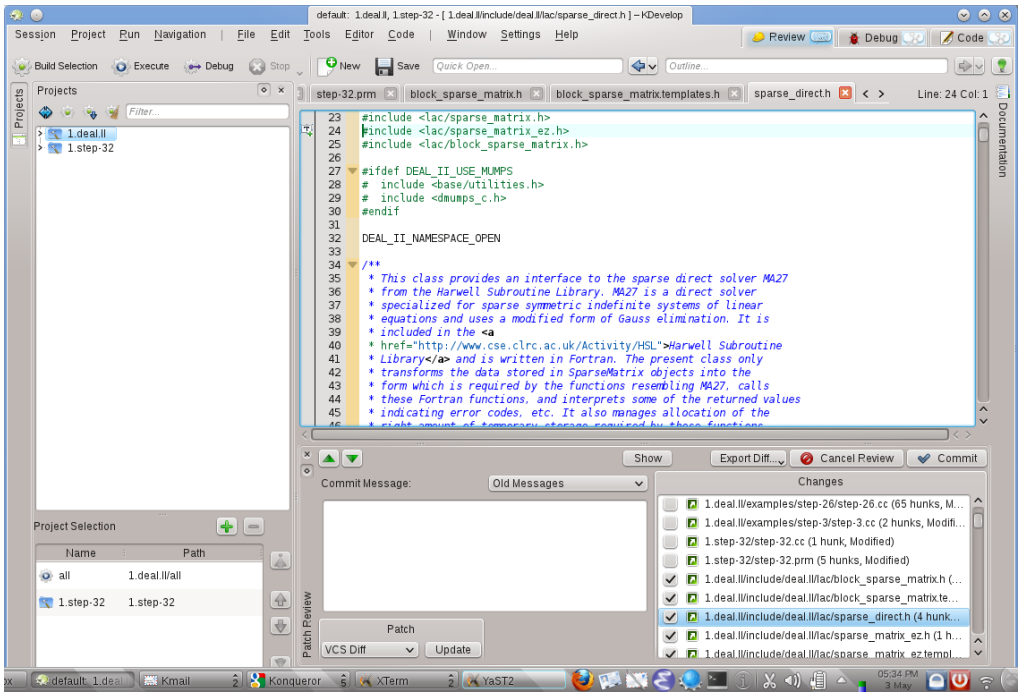
Kapitel 8

Arbeta med versionskontrollsystem

Om du arbetar med större projekt, är det troligt att källkoden hanteras av ett versionskontrollsystem såsom subversion eller git. Följande beskrivning är skriven med **subversion** i åtanke men är lika relevant om du använder **git** eller något annat versionskontrollsystem som stöds.

Observera först att om katalogen som ett projekt finns i hanteras av versionskontroll, märker KDevelop det automatiskt. Med andra ord: Det är inte nödvändigt att tala om för KDevelop att själv checka ut en kopia när projektet skapas, det går bra att peka KDevelop på en katalog där du tidigare har checkat ut en kopia från arkivet. Om du har en sådan katalog med versionskontroll, öppna verktygsvyn **Projekt**. Därefter finns det ett antal saker du kan göra:

- Om katalogen har blivit föråldrad, kan du uppdatera den från arkivet: Klicka på projektnamnet med höger musknapp, gå till menyn **Subversion** och välj **Uppdatera**. Det uppdaterar alla filer som hör till projektet i förhållande till arkivet.
- Om du vill begränsa åtgärden till individuella underkataloger eller filer, expandera trädvyn för projektet till nivån du vill och högerklicka på en underkatalog eller ett filnamn, och sedan göra samma sak som ovan.



- Om du har redigerat en eller flera filer, expandera projektvyn till katalogen där filerna finns och högerklicka på katalogen. Det ger dig menyalternativet **Subversion** som erbjuder olika alternativ. Välj **Jämför med bas** för att se skillnaderna mellan versionen du har redigerat och versionen i arkivet som du tidigare checkade ut (versionen 'bas'). Den resulterande vyn visar 'skillnaderna' för alla filer i katalogen.

- Om du bara redigerar en enda fil, kan du också få menyn **Subversion** för filen genom att helt enkelt klicka på det motsvarande filnamnet i projektvyn. Ännu enklare är att bara högerklicka på **Editor**-vyn där filen har öppnats ger också menyalternativet.

- Om du vill arkivera en eller flera redigerade filer, högerklicka antingen på en individuell fil, underkatalog, eller hela projektet och välj **Subversion** → **Arkivera**. Det går till läget **Granska**, det tredje läget förutom **Kod** och **Avlusa** som du kan se i övre högra hörnet i KDevelops huvudfönster. Bilden till höger visar hur det ser ut. Med **Granska**, visar den övre delen skillnaderna i hela underkatalogen eller projektet och varje individuell fil med ändringarna färglagda (se de olika flikarna på den sidan av fönstret). Normalt är alla ändringar i ändringsuppsättningen som ska arkiveras, men du kan avmarkera några av filerna om deras ändringar inte är relaterade till det du vill arkivera. I exemplet till höger har jag exempelvis avmarkerat `step-32.cc` och `step-32.prm` eftersom ändringarna i dessa filer inte har något att göra med de andra ändringarna jag har gjort i projektet, och jag inte vill arkivera dem ännu (jag kanske vill göra det senare i en separat arkivering). Efter att ha granskat ändringarna kan du skriva in ett arkiveringsmeddelande i textrutan och trycka på **Arkivera** till höger för att skicka iväg allt.

- Liksom med att se skillnader, om du vill arkivera en enda fil kan du bara högerklicka på editorfönstret för att få menyalternativet **Subversion** → **Arkivera**.

Kapitel 9

Anpassa KDevelop

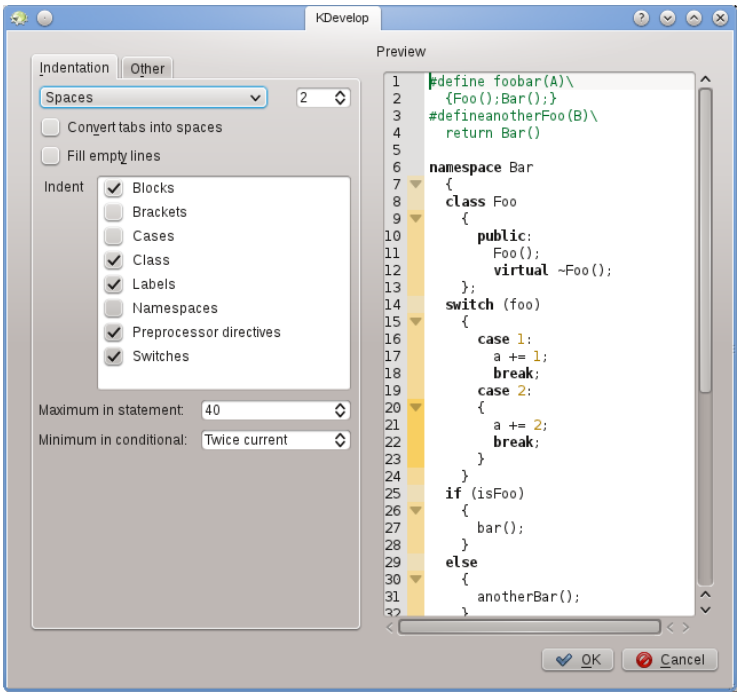
Det finns tillfällen när du vill ändra standardutseendet eller beteendet hos KDevelop, exempelvis eftersom du är van vid andra snabbtangenter eller eftersom projektet kräver en annan indexeringsstil av källkod. I följande avsnitt beskriver vi kortfattat de olika sätt som KDevelop kan anpassas för dessa behov.

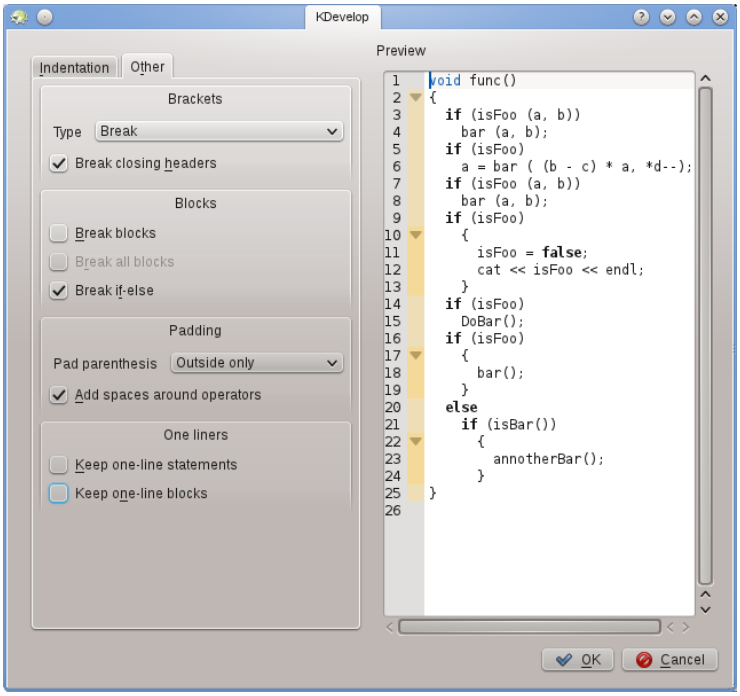
9.1 Anpassa editorn

Det finns ett antal användbara saker man kan anpassa i och omkring KDevelops inbyggda editor. Av mer generell användning är att sätta på radnumrering genom att använda menyalternativet **Editor** → **Visa** → **Visa radnummer**, vilket gör det enklare att matcha kompilatorfelmeddelanden eller avlusningsmeddelanden med positionen i koden. I samma undermeny kan du också vilja sätta på *ikonkant*: en kolumn till vänster om koden där KDevelop visar ikoner, såsom om det finns en brytpunkt på den aktuella raden.

9.2 Anpassa kodindentering

Många av oss tycker om kod formaterad på ett visst sätt. Många projekt kräver också en viss indenteringsstil. Ingendera kanske motsvarar KDevelops förvalda indenteringsstil. Det kan dock anpassas: Gå till menyalternativet **Inställningar** → **Anpassa KDevelop**, klicka sedan på **Källkodsformatering** till vänster. Du kan välja en av de fördefinierade indenteringsstilarna med omfattande användning, eller definiera din egen genom att lägga till en ny stil och sedan redigera den. Det kanske inte finns ett sätt att exakt återskapa stilen som projektets källkod har indenterats med i det förgångna, men du kan komma nära genom att använda inställningarna för en ny stil. Ett exempel visas i de två bilderna nedan.


KDevelop
Indentation
Other
Spaces
2
Convert tabs into spaces
Fill empty lines
Indent
Blocks
Brackets
Cases
Class
Labels
Namespaces
Preprocessor directives
Switches
Maximum in statement:
40
Minimum in conditional:
Twice current
Preview
#define foobar(A)\
{Foo();Bar();}
#defineanotherFoo(B)\
return Bar()
namespace Bar
{
class Foo
{
public:
Foo();
virtual ~Foo();
};
switch (foo)
{
case 1:
a += 1;
break;
case 2:
{
a += 2;
break;
}
}
if (isFoo)
{
bar();
}
else
{
anotherBar();
OK
Cancel


KDevelop
Indentation
Other
Brackets
Type
Break
Break closing headers
Blocks
Break blocks
Break all blocks
Break if-else
Padding
Pad parenthesis
Outside only
Add spaces around operators
One liners
Keep one-line statements
Keep one-line blocks
Preview
void func()
{
if (isFoo (a, b))
bar (a, b);
if (isFoo)
a = bar ((b - c) * a, *d--);
if (isFoo (a, b))
bar (a, b);
if (isFoo)
{
isFoo = false;
cat << isFoo << endl;
}
if (isFoo)
DoBar();
if (isFoo)
{
bar();
}
else
if (isBar())
{
annotherBar();
}
}
OK
Cancel

NOT
Med **KDevelop 4.2.2** kan du skapa en ny stil för en viss Mime-typ (t.ex. C++ deklarationsfiler) men stilen visas inte i listan över möjliga stilar för andra Mime-typer (t.ex. för C++ källkodsfiler) även om det naturligtvis skulle vara användbart att använda samma stil för båda filtyperna. Därför måste du definiera varje stil två gånger, en gång för deklarationsfiler och en för källkodsfiler. Det har rapporterats som KDevelop fel 272335.

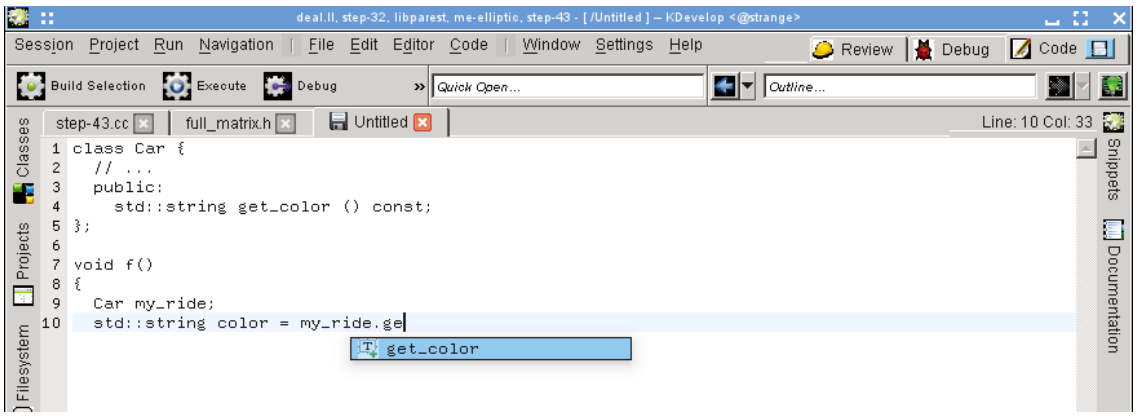
9.3 Anpassa snabbtangenter

KDevelop har en nästan gränslös lista över snabbtangenter (några av dem listas i avsnittet 'Användbara snabbtangenter' i flera kapitel i handboken) som kan ändras enligt tycke och smak med menyn **Inställningar** → **Anpassa genvägar**. Längst upp i dialogrutan kan du skriva in ett sökbegrepp så visas bara de kommandon som matchar. Därefter kan du redigera vilken tangentkombination som är kopplad till kommandot.

Två som har funnits vara mycket användbara att ändra är att ställa in **Justera** till tabulatortangenten (många skriver oftast inte in tabulatortecken för hand och föredrar istället att editorn väljer kodens layout: med den ändrade genvägen indenterar/avindenterar/justerar KDevelop koden när tabulator används). Den andra är att placera **Växla brytpunkt** på **Ctrl-B** eftersom det är en ofta använd åtgärd.

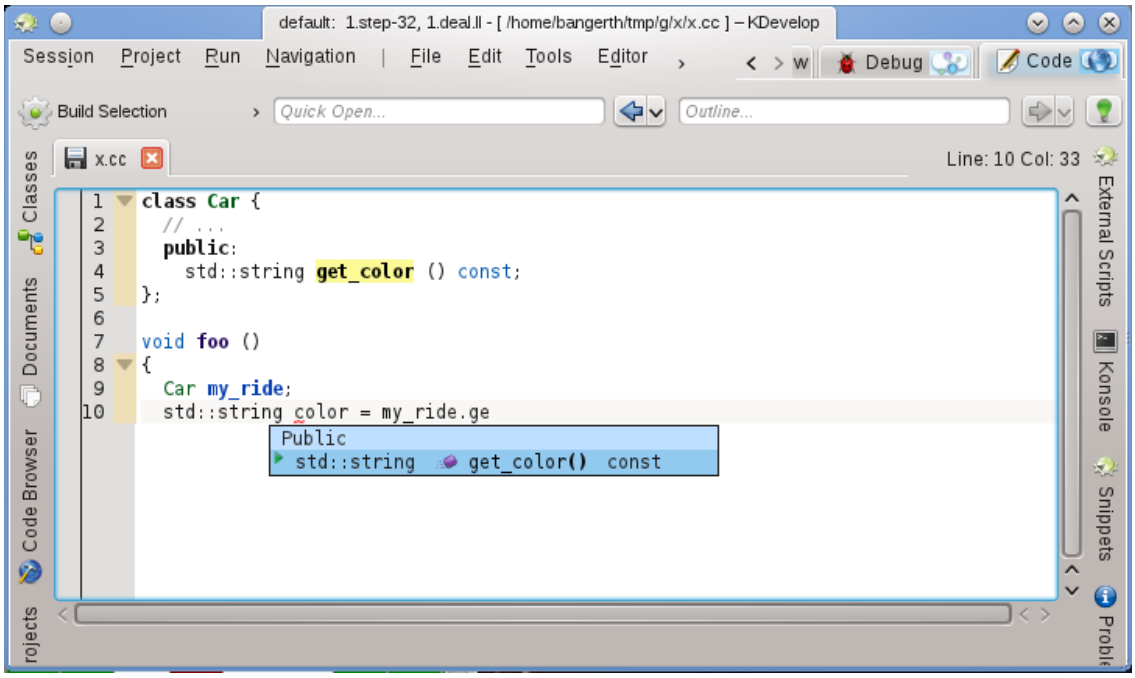
9.4 Automatisk kodkomplettering

Kodkomplettering beskrivs i handbokens avsnitt om att skriva källkod. I KDevelop kommer den från två källor: editorn, och tolkningsgränssnittet. Editorn (Kate) är en komponent i den mer omfattande KDE-miljön och erbjuder automatisk komplettering baserad på ord som den redan har sett i andra delar av samma dokument. Sådan automatisk komprimering kan identifieras i verktygstipset av ikonen som inleder den:



Editorns kodkomplettering kan anpassas via **Inställningar** → **Anpassa editor** → **Redigering** → **Automatisk komplettering**. I synnerhet kan du välja hur många tecken som du måste skriva i ett ord innan automatisk komprimering sätter igång.

Å andra sidan är KDevelops egen automatiska komplettering mycket kraftfullare eftersom den tar hänsyn till semantisk information om innehållet. Exempelvis vet den vilka medlemsfunktioner som ska erbjudas när du skriver `object.`, etc. som visas här:



Den sammanhangsberoende informationen kommer från diverse insticksprogram för språkstöd, som kan användas efter en given fil har sparats (så att filtypen kan kontrolleras och korrekt språkstöd kan användas).

KDevelops komplettering är uppsatt för att visas medan du skriver, direkt, nästan överallt där något kan kompletteras på något sätt. Det går att ställa in med **Inställningar** → **Anpassa KDevelop** → **Språkstöd**. Om det inte redan är aktiverat (vilket det bör vara, som förval), säkerställ att **Aktivera automatisk start** är aktiverat.

KDevelop har två sätt att visa en komplettering: **Minimal automatisk komplettering** visar bara den grundläggande information i verktygstips för komplettering (dvs. namnrymden, klass, funktion, eller variabelnamn). Det ser liknande ut som komplettering i Kate (utom för ikonerna).

Å andra sidan, visar **Fullständig komplettering** dessutom typen för varje post, och i fallet med funktioner, också argumenten de har. Förutom det, om du för närvarande håller på att fylla i argumenten för en funktion, har fullständig komplettering ytterligare en informationsruta ovanför markören som visar dig det aktuella argumentet som du arbetar på.

KDevelops kodkomplettering ska också placera alla kompletteringsobjekt som matchar den aktuella förväntade typen i både minimal och fullständig komplettering överst och färglägga dem med grönt, känt som ‘bästa träffar’.

De tre möjliga val för kompletteringsnivå i inställningsdialogen är:

- **Alltid minimal komplettering**: Visa aldrig ‘Fullständig komplettering’
- **Minimal automatisk komplettering**: Visa bara ‘Fullständig komplettering’ när automatisk komplettering har utlösts manuellt (dvs. när du trycker på **Ctrl-Mellanslag**).
- **Alltid fullständig komplettering**: Visa alltid ‘Fullständig komplettering’

Kapitel 10

Tack till och licens

Översättning Stefan Asserhäll stefan.asserhall@bredband.net