KDevelop Handbook

This documentation was converted from the KDE UserBase KDevelop4/Manual page.

Contents

Abstract

KDevelop is an Integrated Development Environment to be used for a wide variety of programming tasks.

Chapter 1

What is KDevelop?

KDevelop is a modern integrated development environment (IDE) for C++ (and other languages) that is one of many KDE applications. As such it runs on Linux® (even if you run one of the other desktops, such as GNOME) but it is also available for most other variants of UNIX® and for Windows as well.

KDevelop offers all amenities of modern IDEs. For large projects and applications, the most important feature is that KDevelop *understands* C++: it parses the entire source base and remembers which classes have which member functions, where variables are defined, what their types are, and many other things about your code. For example, let's say one of your project's header files declares a class

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

and later on in your program you have

```
Car my_ride;
// ...do something with this variable...
std::string color = my_ride.ge
```

it will have remembered that `my_ride` in the last line is a variable of type `Car` and offer you to complete `ge` as `get_color()` since this is the only member function of the `Car` class that starts like this. Instead of continuing to type you just hit **Enter** to get the full word; this saves typing, avoids typos, and doesn't require you to remember the exact names of the hundreds or thousands of functions and classes that make up large projects.

As a second example, assume you have code like this:

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

If you hover the mouse over the symbol `var` in function `bar` you get the option to see all uses of this symbol. Clicking on it will only show you the uses of this variable in function `bar` because KDevelop understands that the variable `var` in function `foo` has nothing to do with it. Similarly, right clicking on the variable name allows you to rename the variable; doing so will only touch the variable in `bar` but not the one with the same name in `foo`.

But KDevelop is not just an intelligent code editor; there are other things KDevelop does well. Obviously, it highlights the source code in different colors; it has a customizable indenter; it has an integrated interface to the GNU debugger gdb; it can show you the documentation for a function if you hover the mouse over a use of this function; it can deal with different kinds of build environments and compilers (e.g. with **make** and **cmake**-based project), and many other neat things that are discussed in this manual.

Chapter 2

Sessions and projects: The basics of KDevelop

In this section, we will go over some of the terminology of how KDevelop sees the world and how it structures work. In particular, we introduce the concept of *sessions* and *projects* and explain how you can set up the projects you want to work on in KDevelop.

2.1 Terminology

KDevelop has the concept of *sessions* and *projects*. A session contains all projects that have something to do with each other. For the examples that follow, assume you are the developer of both a library and an application that uses it. You can think of the core KDE libraries as the former and KDevelop as the latter. Another example: Let's say you are a Linux® kernel hacker but you are also working on a device driver for Linux® that hasn't been merged into the kernel tree yet.

So taking the latter as an example, you would have a session in KDevelop that has two projects: the Linux® kernel and the device driver. You will want to group them into a single session (rather than having two sessions with a single project each) because it will be useful to be able to see the kernel functions and data structures in KDevelop whenever you write source code for the driver — for example so that you can get kernel function and variable names auto-expanded, or so that you can see kernel function documentation while hacking on the device driver.

Now imagine you also happen to be a KDE developer. Then you would have a second session that contains KDE as a project. You could in principle have just one session for all of this, but there is no real reason for this: in your KDE work, you don't need to access kernel or device driver functions; and you don't want KDE class names autoexpanded while working on the Linux® kernel. Finally, building some of the KDE libraries is independent of re-compiling the Linux® kernel (whereas whenever you compile the device driver it would also be good to re-compile the Linux® kernel if some of the kernel header files have changed).

Finally, another use for sessions is if you work both on the current development version of a project, as well as on a branch: in that case, you don't want KDevelop to confuse classes that belong to mainline and the branch, so you'd have two sessions, with the same set of projects but from different directories (corresponding to different development branches).

2.2 Setting up a session and importing an existing project

Let's stick with the Linux® kernel and device driver example — you may want to substitute your own set of libraries or projects for these two examples. To create a new session that contains these two projects go to the **Session** → **Start new session** menu at the top left (or, if this is the first time you use KDevelop: simply use the default session you get on first use, which is empty).

We next want to populate this session with projects that for the moment we assume already exist somewhere (the case of starting projects from scratch is discussed elsewhere in this manual). For this, there are essentially two methods, depending on whether the project already is somewhere on your hard drive or whether it needs to be downloaded from a server.

2.2.1 Option 1: Importing a project from a version control system server

Let's first assume that the project we want to set up — the Linux® kernel — resides in some version control system on a server, but that you haven't checked it out to your local hard drive yet. In this case, go to the **Project** menu to create the Linux® kernel as a project inside the current session and then follow these steps:

- Go to **Project** → **Fetch Project** to import a project
- You then have multiple options to start a new project in the current session, depending on where the source files should come from: You can just point KDevelop at an existing directory (see option 2 below), or you can ask KDevelop to get the sources from a repository.
- Assuming you don't already have a version checked out:
  - In the dialog box, under **Select Source**, choose to use **From File System**, **Subversion**, **Git**, **GitHub**, or **KDE**
  - Choose a working directory as destination into which the sources should be checked out
  - Choose an URL for the location of the repository where the source files can be obtained
  - Hit **Get**. This can take quite a long while; depending on the speed of your connection and the size of the project. Unfortunately, in KDevelop 4.2.x the progress bar does not actually show anything, but you can track progress by periodically looking at the output of the command line command

    ```
    du -sk /path/to/KDevelop/project
    ```

    to see how much data has already been downloaded.

> **NOTE**
> The problem with the progress bar has been reported as KDevelop bug 256832.

> **NOTE**
> In this process, I also get the error message *You need to specify a valid location for the project* which can be safely ignored.

- It asks you to select a KDevelop project file in this directory. Since you probably don't have one yet, simply hit **Next**
- Hit **Next** again

- KDevelop will then ask you to choose a project manager. If this project uses standard UNIX® make files, choose the custom makefile project manager
- KDevelop will then start to parse the entire project. Again, it will take quite a while to go through all files and index classes etc. At the bottom right of the main window, there is a progress bar that shows how long this process has come along. (If you have several processor cores, you can accelerate this process by going to the **Settings** → **Configure KDevelop** menu item, then selecting **Background parser** on the left, and increasing the number of threads for background parsing on the right.)

2.2.2 Option 2: Importing a project that is already on your hard drive

Alternatively, if the project you want to work with already exists on your hard drive (for example, because you have downloaded it as a tar file from an FTP server, because you already checked out a version of the project from a version control system, or because it is your own project that exists *only* on your own hard drive), then use **Projects** → **Open/Import project** and in the dialog box choose the directory in which your project resides.

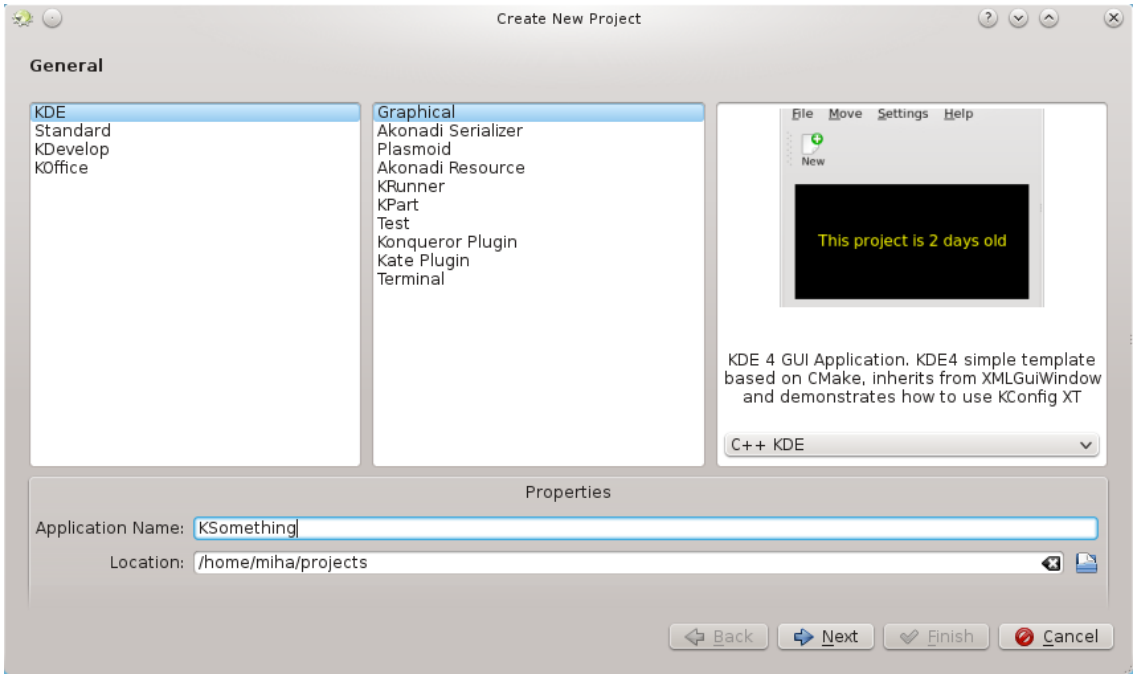
2.3 Setting up an application as a second project

The next thing you want to do is set up other projects in the same session. In the example above, you would want to add the device driver as the second project, which you can do using exactly the same steps.

If you have multiple applications or libraries, simply repeat the steps to add more and more projects to your session.

2.4 Creating projects from scratch

There is of course also the possibility that you want to start a new project from scratch. This can be done using the **Projects** → **New from Template...** menu item, which presents you with a template selection dialog. Some project templates are provided with KDevelop, but even more are available by installing the KAppTemplate application. Choose the project type and programming language from the dialog, enter a name and location for you project, and click **Next**.



The second page of the dialog allows you to set up a version control system. Choose the system you wish the use, and fill in the system-specific configuration if needed. If you do not wish to use a version control system, or want to set it up manually later, choose **None**. When you are happy with your choice, press **Finish**.

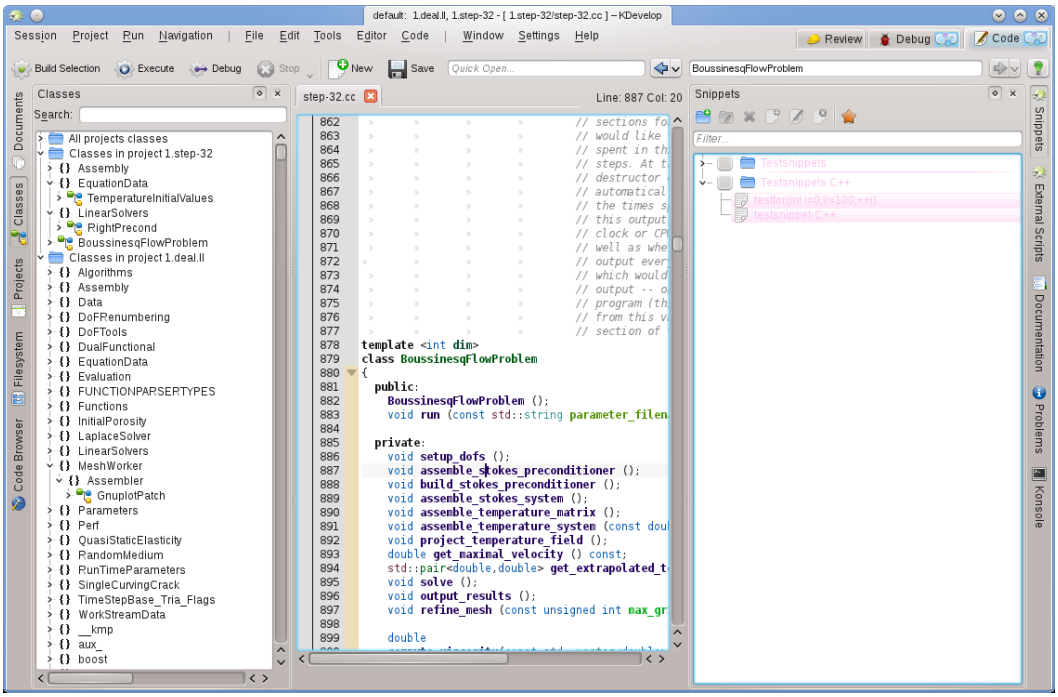
Your project is now created, so you can try building or installing it. Some templates will include comments within the code, or even a separate README file, and it is recommended that you read those first. Then, you can start working on your project, by adding whatever features you want.

Chapter 3

Working with source code

Besides debugging, reading through and writing source code is what you will spend the most time with when developing software. To this end, KDevelop offers you many many different ways to explore source codes and to make writing it more productive. As discussed in more detail in the following sections, KDevelop is not just a source editor — rather, it is a source management system that gives you different views of extracted information on the files that collectively make up your session's source code.

3.1 Tools and views



In order to work with projects, KDevelop has the concept of *tools*. A tool provides a particular view of the source, or an action that can be taken with it. Tools are represented by buttons around the perimeter of your window (in vertical text along the left and right margins, or horizontally

along the bottom margin). If you click on them, they expand to a subwindow — a *view* — within the main window; if you click on the tool button again, the subwindow disappears again.

To make a subwindow disappear, you can also click at the **x** at the top right of the subwindow

The picture above shows a particular selection of tools, aligned on the left and right margins; in the picture, the **Classes** tool is open on the left and the **Snippets** tool on the right, along with an editor for a source file in the middle. In practice, most of the time you will probably only have the editor and maybe the **Classes** or **Code Browser** tool open at the left. Other tool view will likely only be open temporarily as you use the tool, leaving more space for the editor most of the time.

When you run KDevelop the first time, you should already have the **Projects** tool button. Click on it: it will open a subwindow that shows the projects you have added to the session at the bottom, and a file system view of the directories of your projects at the top.

There are many other tools you can use with KDevelop, not all of which are initially present as buttons on the perimeter. To add some, go to the **Windows** → **Add tool view** menu entry. Here are some that you will likely find useful:

- **Classes**: A complete list of all classes that are defined in one of the projects or your session with all of their member functions and variables. Clicking on any of the members opens a source editor window at the location of the item you clicked on.
- **Documents**: Lists some of the more recently visited files, by kind (e.g. source files, patch files, plain text documents).
- **Code Browser**: Depending on your cursor position in a file, this tool shows things that are related. For example, if you are on an `#include` line, it shows information about the file you are including such as what classes are declared in that file; if you are on an empty line at file scope, it shows the classes and functions declared and defined in the current file (all as links: clicking on them brings you to the point in the file where the declaration or definition actually is); if you are in a function definition, it shows where the declaration is and offers a list of places where the function is used.
- **File system**: Shows you a tree view of the file system.
- **Documentation**: Allows you to search for man pages and other help documents.
- **Snippets**: This provides sequences of text that one uses over an over and doesn't want to write every time. For example, in the project from which the picture above was created, there is a frequent need to write code like

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

This is an awkward expression but it will look almost exactly like this every time you need such a loop — which would make it a good candidate for a snippet.

- **Konsole**: Opens a command line window inside KDevelop's main window, for the occasional command you may want to enter (e.g. to run `./configure`).

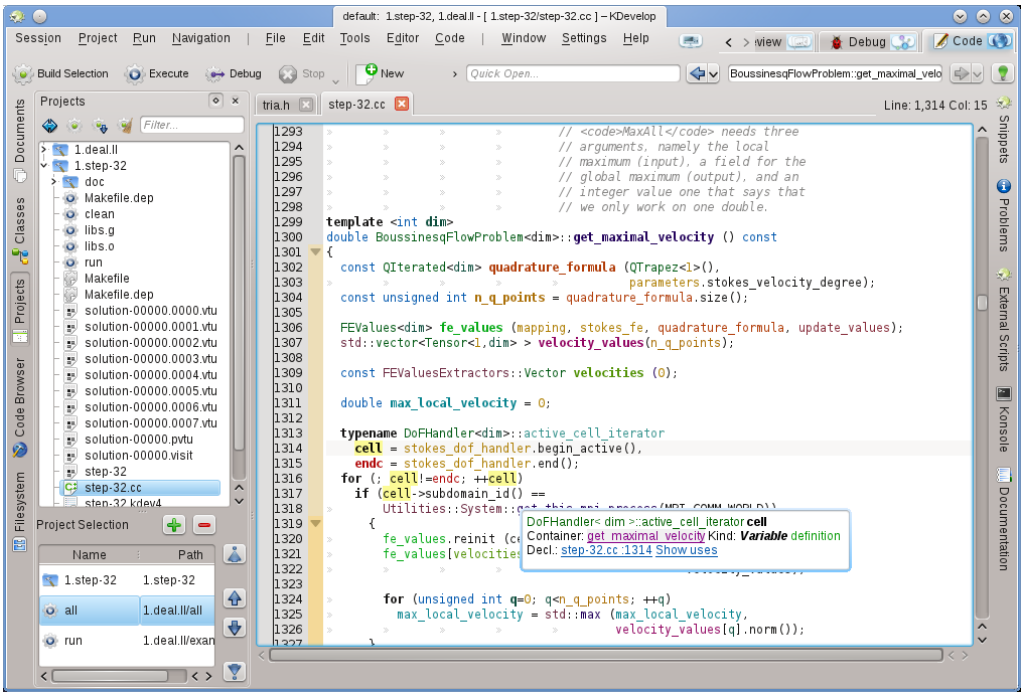
A complete list of tools and views is given here.

For many programmers, vertical screen space is the most important. To this end, you can arrange your tool views at the left and right margin of the window: to move a tool, click on its symbol with the right mouse button and select a new position for it.

3.2 Exploring source code

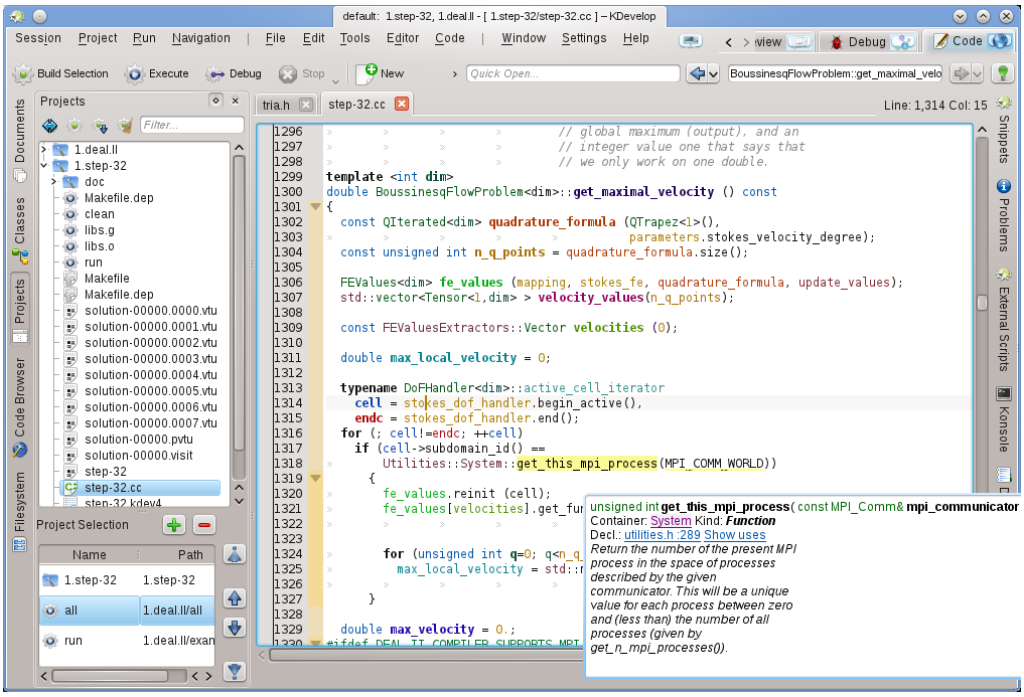
3.2.1 Local information

KDevelop *understands* source code, and as a consequence it is really good at providing you information about variables or functions that may appear in your program. For example, here is a snapshot of working with a piece of code and hovering the mouse over the symbol `cell` in line 1316 (if you are working keyboard oriented, you can achieve the same effect by holding down the **Alt** key for a while):



KDevelop shows me a tooltip that includes the type of the variable (here: `DoFHandler<dim>active_cell_iterator`), where this variable is declared (the *container*, which here is the surrounding function `get_maximal_velocity` since it is a local variable), what it is (a variable, not a function, class or namespace) and where it is declared (in line 1314, just a few lines up in the code).

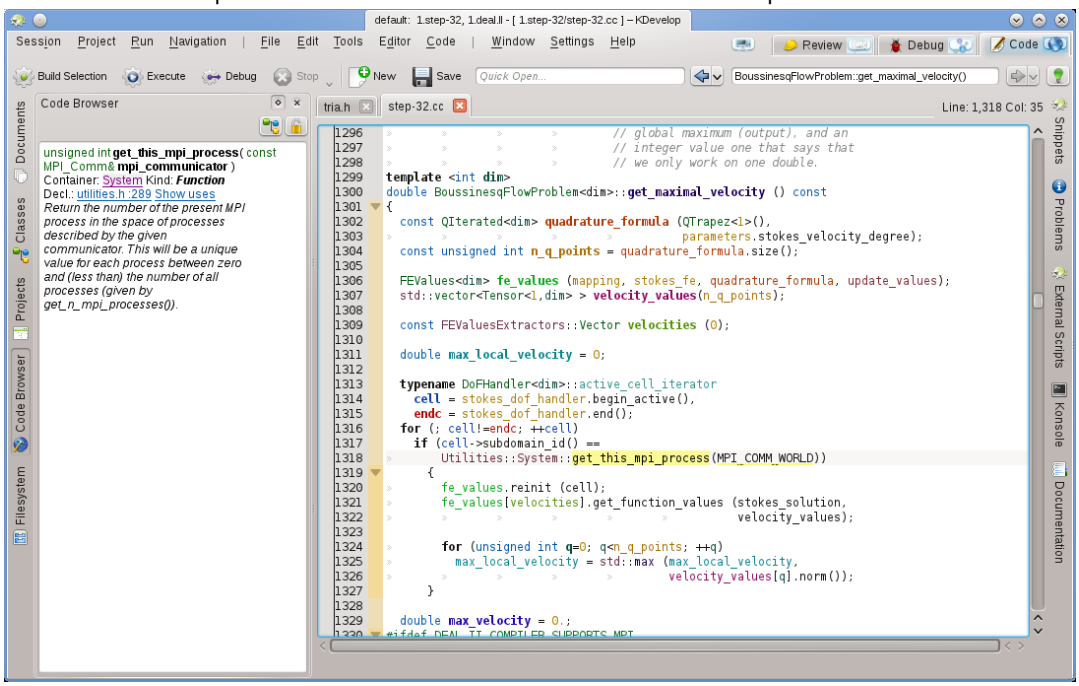
In the current context, the symbol over which the mouse was hovering has no associated documentation. In this example, had the mouse hovered over the symbol `get_this_mpi_process` in line 1318, the outcome would have been this:



Here, KDevelop has cross-referenced the declaration from an entirely different file (`utilities.h`, which in fact even resides in a different project of the same session) together with the doxygen-style comment that accompanies the declaration there.

What makes these tooltips even more useful is that they are dynamic: I can click on the container to get information about the context in which the variable is declared (i.e. about the namespace `System`, such as where it is declared, defined, used, or what its documentation is) and I can click on the blue links that will reset the cursor position to the location of the declaration of the Symbol (e.g. in `utilities.h`, line 289) or give me a list of places where this symbol is used in the current file or throughout all the projects of the current session. The latter is often useful if you want to explore how, for example, a particular function is used in a large code basis.

NOTE
The information in a tooltip is fleeting — it depends on you holding the **Alt** key down or hovering your mouse. If you want a more permanent place for it, open the **Code Browser** tool view in one of the sub-windows. For example, here the cursor is on the same function as in the example above, and the tool view on the left presents the same kind of information as in the tooltip before:

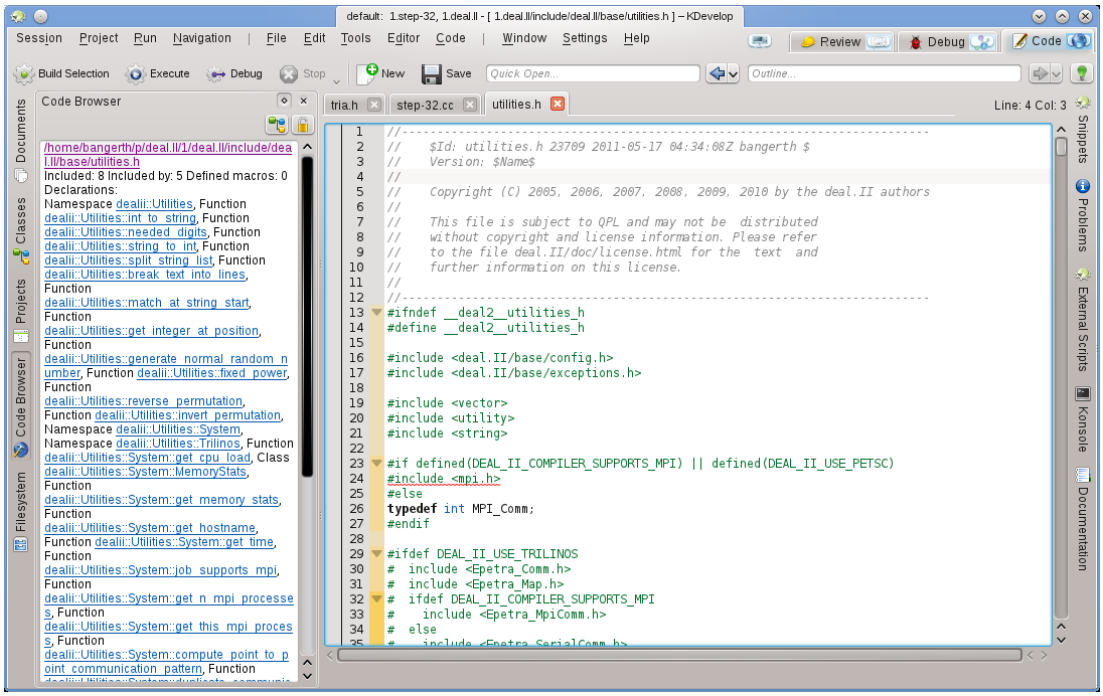


Moving the cursor around on the right changes the information presented on the left. What's more, clicking on the **Lock current view** button at the top right allows you to lock this information, making it independent of the cursor movement while you explore the information presented there.

NOTE
This sort of context information is available in many other places in KDevelop, not just in the source editor. For example, holding down the **Alt** key in a completion list (e.g. when doing a quick-open) also yields context information about the current symbol.

3.2.2 File scope information

The next level up is to obtain information about the entire source file you are currently working on. To this end, position the cursor at file scope in the current file and look at the what the **Code Browser** tool view shows:

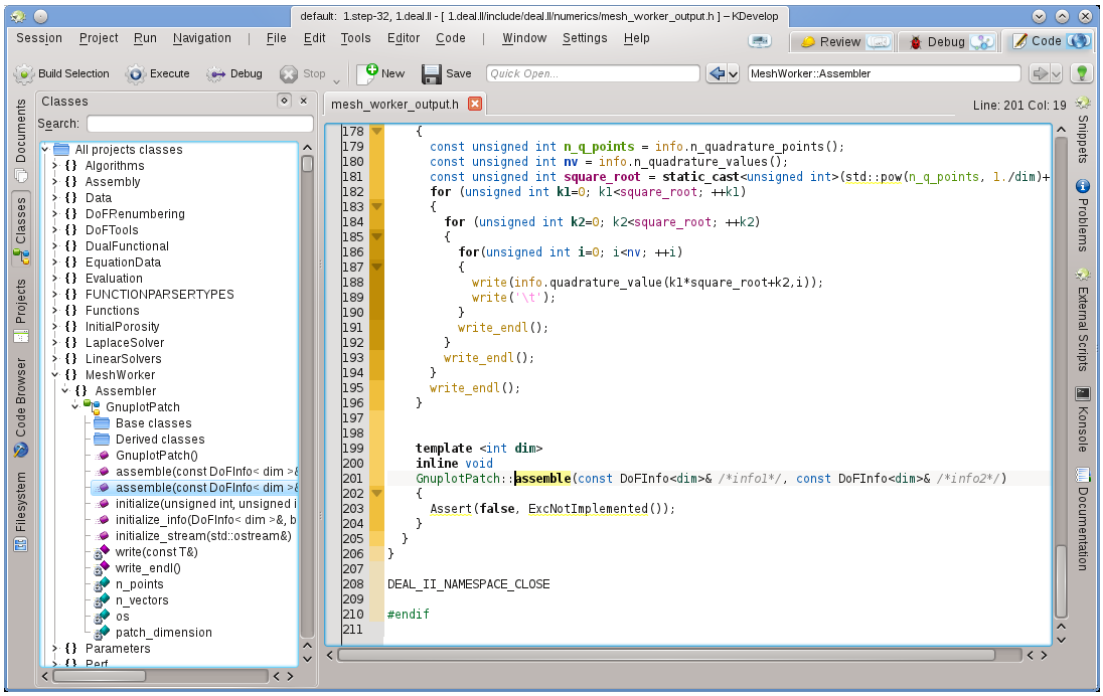


Here, it shows a list of namespaces, classes and functions declared or defined in the current file, giving you an overview of what's happening in this file and a means to jump directly to any of these declarations or definitions without scrolling up and down in the file or searching for a particular symbol.

> NOTE
> The information shown for file scope is the same as presented in the 'Outline' mode discussed below for navigating source code; the difference is that outline mode is only a temporary tooltip.

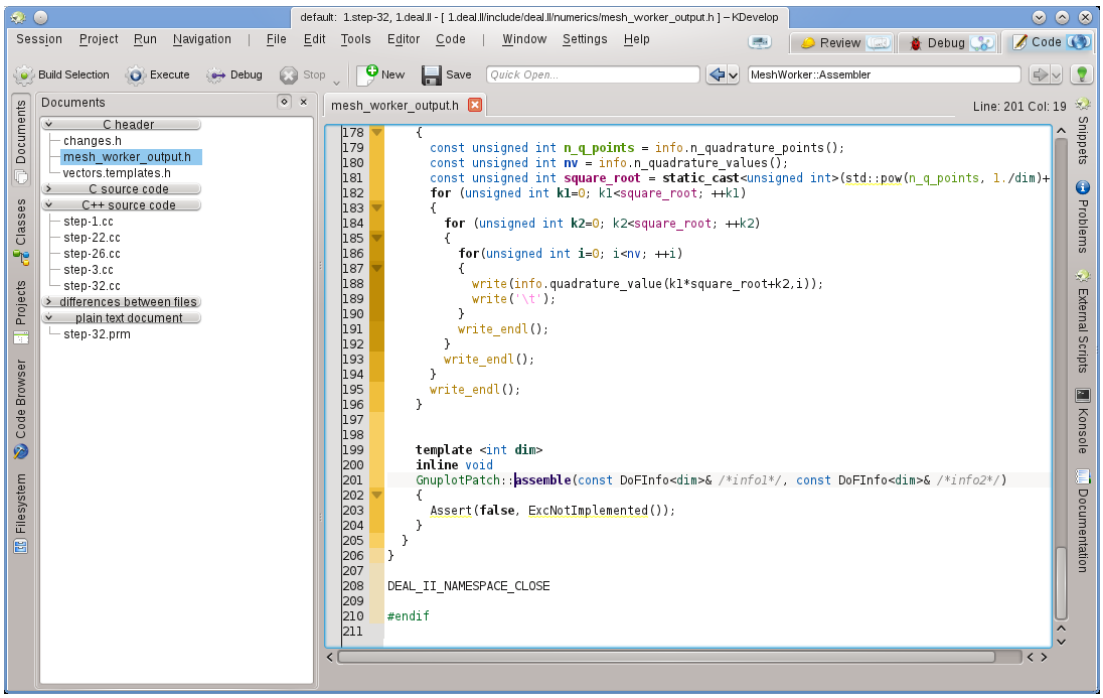
3.2.3 Project and session scope information

There are many ways to obtain information about an entire project (or, in fact, about all projects in a session). This sort of information is typically provided through various tool views. For example, the **Classes** tool view provides a tree structure of all classes and surrounding namespaces for all projects in a session, together with the member functions and variables of each of these classes:



Hovering over an entry provides again information about the symbol, its location of declaration and definition, and its uses. Double-clicking on an entry in this tree view opens an editor window at the location where the symbol is declared or defined.

But there are other ways of looking at global information. For example, the **Documents** tool provides a view of a project in terms of the kinds of files or other documents this project is comprised of:



3.2.4 Rainbow color highlighting explained

KDevelop uses a variety of colors to highlight different objects in source code. If you know what the different colors mean, you can very quickly extract a lot of information from source code just by looking at the colors, without reading a single character. The highlighting rules are as follows:

- Objects of type Class / Struct, Enum (the values and the type), (global) functions, and class members each have their own color assigned (classes are green, enums are dark red, and members are dark yellow or violet, (global) functions are always violet).
- All global variables are colored in dark green.
- Identifiers which are typedefs for another type are colored in teal.
- All declarations and definitions of objects are in bold.
- If a member is accessed from within the context where it is defined (base or derived class) it appears in yellow, otherwise it appears in violet.
- If a member is private or protected, it gets colored in a slightly darker color when used.
- For variables local to a function body scope, rainbow colors are picked based on a hash of the identifier. This includes the parameters to the function. An identifier always will have the same color within its scope (but the same identifier will get a different color if it represents a different object, i.e. if it is redefined in a more nested scope), and you will usually get the same color for the same identifier name in different scopes. Thus, if you have multiple functions taking parameters with the same names, the arguments will all look the same color-wise. These rainbow colors can be turned off separately from the global coloring in the settings dialog.
- Identifiers for which KDevelop could not determine the corresponding declaration are colored in white. This can sometimes be caused by missing `#include` directives.
- In addition to that coloring, the normal editor syntax highlighting will be applied, as known from Kate. KDevelop's semantic highlighting will always override the editor highlighting if there is a conflict.

3.3 Navigating in source code

In the previous section, we have discussed exploring source code, i.e. getting information about symbols, files and projects. The next step is then to jump around your source base, i.e. to navigate in it. There are again various levels at which this is possible: local, within a file, and within a project.

> NOTE
> Many of the ways to navigate through code are accessible from the **Navigate** menu in the KDevelop main window.

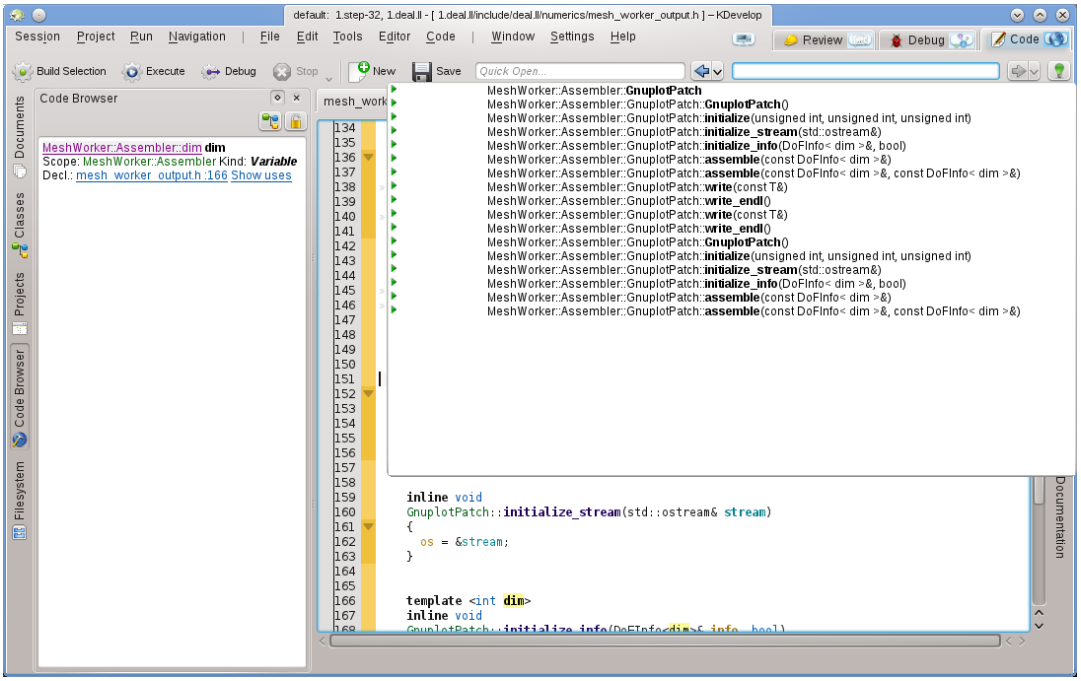
3.3.1 Local navigation

KDevelop is much more than an editor, but it is *also* a source editor. As such, you can of course move the cursor up, down, left or right in a source file. You can also use the **PageUp** and **PageDown** keys, and all the other commands you are used from any useful editor.

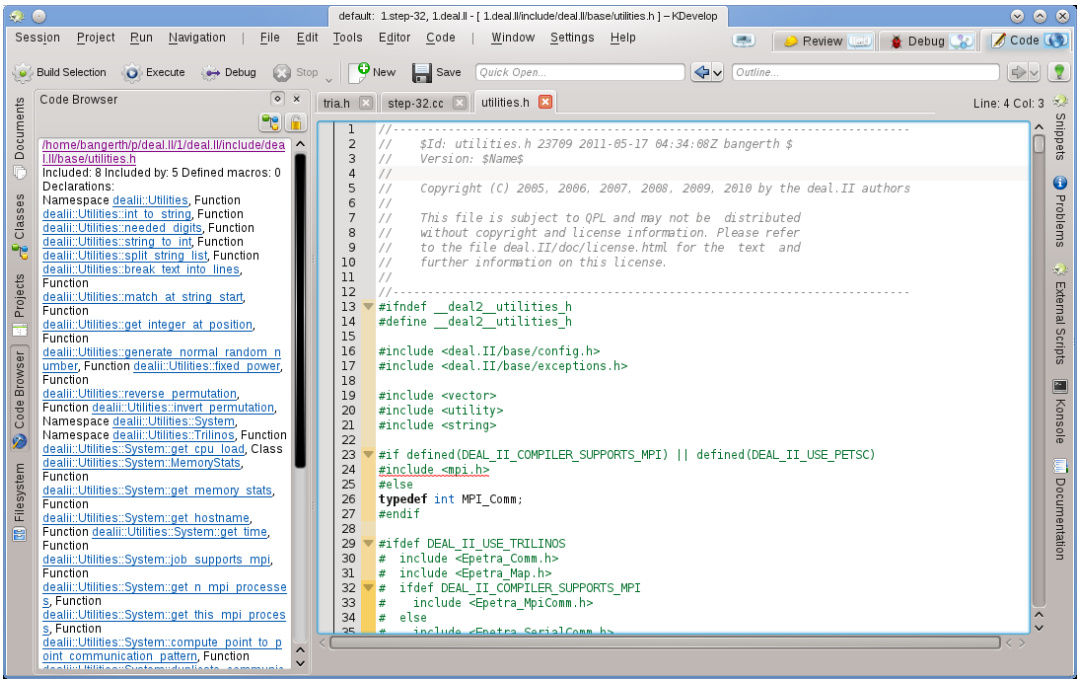
3.3.2 File scope navigation and outline mode

At the file scope, KDevelop offers many possible ways to navigate through source code. For example:

- **Outline**: You can get an outline of what's in the current file in at least three different ways:
  - Clicking into the **Outline** textbox at the top right of the main window, or hitting **Alt-Ctrl-N** opens a drop-down menu that lists all function and class declarations:



    You can then just select which one to jump to, or — if there are a lot — start typing any text that may appear in the names shown; in that case, as you keep typing, the list becomes smaller and smaller as names are removed that don't match the text already typed until you are ready to select one of the choices.
  - Positioning the cursor at file scope (i.e. outside any function or class declarations or definitions) and having the **Code Browser** tool open:



This also provides you an outline of what is happening in the current file, and allows you to select where you want to jump to.

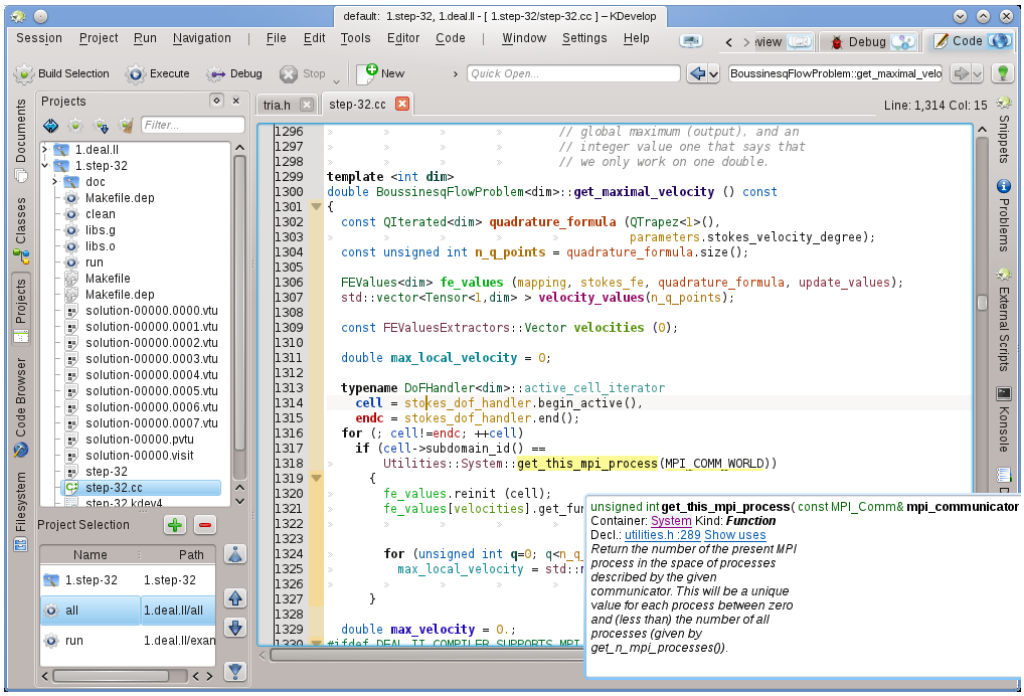
  - Hovering the mouse over the tab for one of the open files also yields an outline of the file in that tab.

- Source files are organized as a list of function declarations or definitions. Hitting **Alt-Ctrl-PgUp** and **Alt-Ctrl-PgDown** jumps to the previous or next function definition in this file.

3.3.3 Project and session scope navigation: Semantic navigation

As mentioned in other places, KDevelop does not usually consider individual source files but rather looks at projects as a whole (or, rather, at all projects that are part of the current session). As a consequence, it offers many possibilities for navigating through entire projects. Some of these are derived from what we have already discussed in the section on Exploring source code while others are genuinely different. The common theme is that these navigation features are based on a *semantic understanding* of the code, i.e. they offer you something that requires parsing entire projects and connecting data. The following list shows some ways how to navigate through source code that is scattered throughout a potentially very large number of files:

- As seen in the section on Exploring source code, you can get a tooltip explaining individual namespace, class, function or variable names by hovering your mouse over it or keeping the **Alt** key pressed for a while. Here is an example:



Clicking on the links for the declaration of a symbol or expanding the list of uses allows you to jump to these locations, if necessary opening the respective file and placing the cursor at the corresponding location. A similar effect can be achieved by using the **Code Browser** tool view also discussed previously.

- A quicker way to jump to the declaration of a symbol without having to click on the links in the tooltip is to temporary enabling **Source Browse Mode** by holding down the **Alt** or **Ctrl** key. In this mode, it is possible to directly click on any symbol in the editor to jump to its declaration.

- **Quick open**: A very powerful way of jumping to other files or locations is to use the various *quick open* methods in KDevelop. There are four versions of these:

  – **Quick open class** (**Navigate** → **Quick open class** or **Alt-Ctrl-C**): You will get a list of all classes in this session. Start typing (a part of) the name of a class and the list will continue to whittle down to only those that actually match what you've typed so far. If the list is short enough, select an element using the up and down keys and KDevelop will get you to the place where the class is declared.

  – **Quick open function** (**Navigate** → **Quick open function** or **Alt-Ctrl-M**): You will get a list of all (member) functions that are part of the projects in the current session, and you can select from it in the same way as above. Note that this list may include both function declarations and definitions.

  – **Quick open file** (**Navigate** → **Quick open file** or **Alt-Ctrl-O**): You will get a list of all files that are part of the projects in the current session, and you can select from it in the same way as above.

  – **Universal quick open** (**Navigate** → **Quick open** or **Alt-Ctrl-Q**): If you forget which key combination is bound to which of the above commands, this is the universal swiss army knife — it simply presents you with a combined list of all files, functions, classes, and other things from which you can select.

- **Jump to declaration/definition**: When implementing a (member) function, one often needs to switch back to the point where a function is declared, for example to keep the list of function arguments synchronised between declaration and definition, or to update the documentation. To do so, place the cursor onto the function name and select **Navigation** → **Jump to declaration** (or hit **Ctrl-.**) to get to the place where the function is declared. There are multiple ways to get back to the original place:

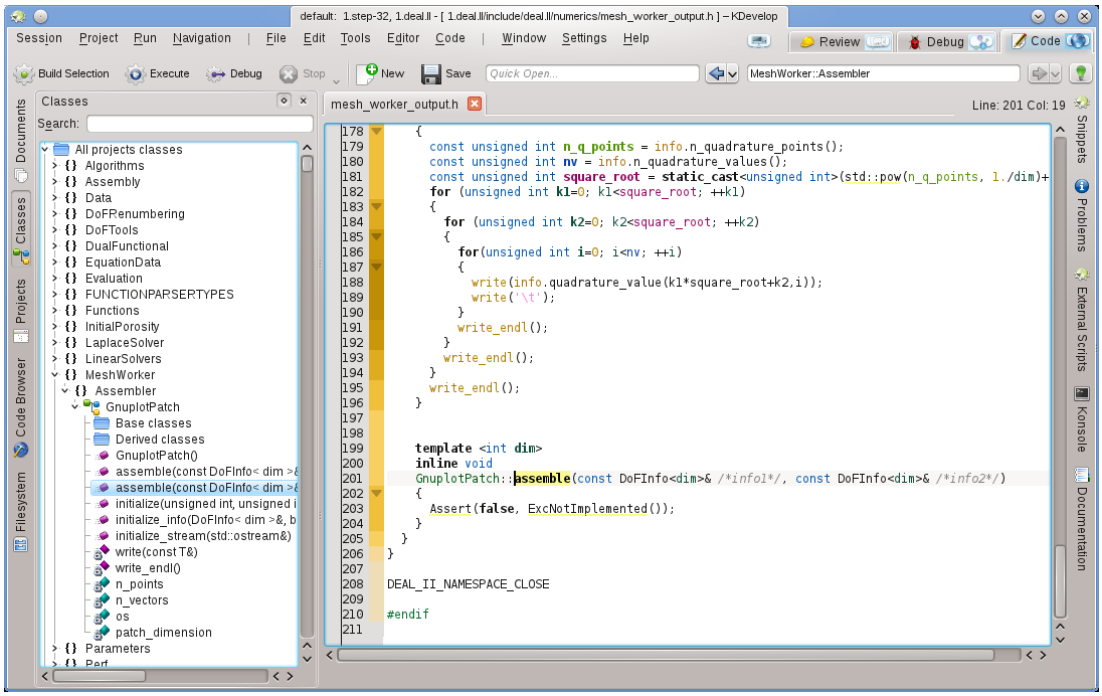
  - Selecting **Navigation** → **Jump to definition** (or hitting **Ctrl-,**).
  - Selecting **Navigation** → **Previous visited context** (or hit **Meta-Left**), as described below.

> NOTE
> Jumping to the declaration of a symbol is something that does not only work when placing the cursor on the name of the function you are currently implementing. Rather, it also works for other symbols: Putting the cursor on a (local, global, or member) variable and jumping to its declaration also transports you to its location of declaration. Similarly, you can place the cursor on the name of a class, for example in a variable of function declaration, and jump to the location of its declaration.

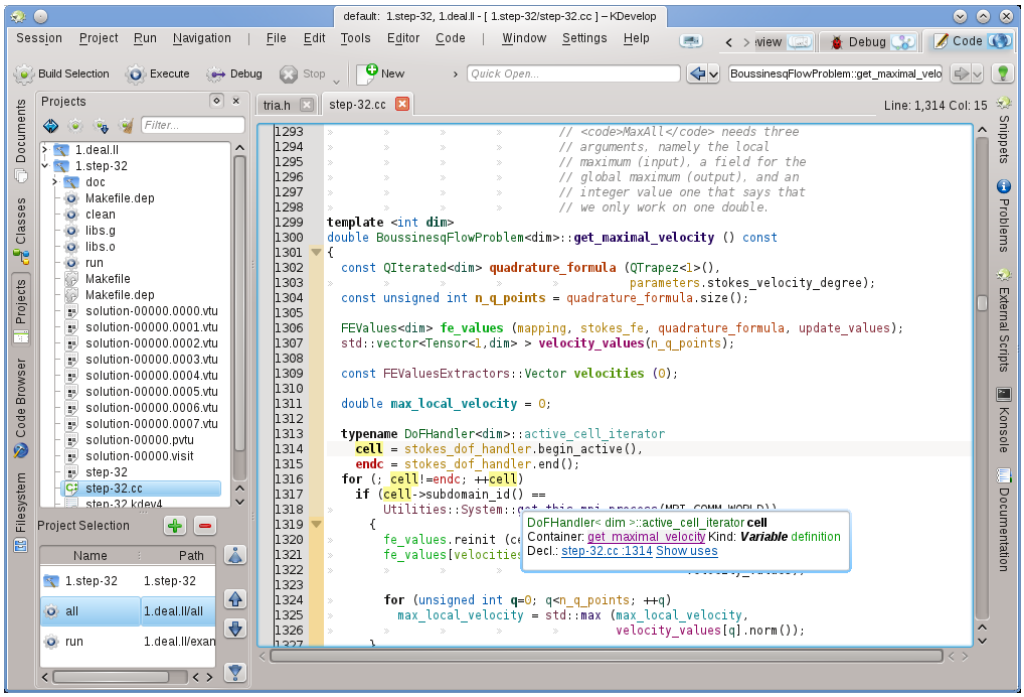
- **Switch between declaration/definition**: In the example above, to jump to the site of the declaration of the current function, you need to first place the cursor on the function name. To avoid this step, you can select **Navigation** → **Switch definition/declaration** (or hit **Shift-Ctrl-C**) to jump to the declaration of the function within which the cursor currently is. Selecting the same menu entry a second time transports you back to the place where the function is defined.
- **Previous/Next use**: Placing the cursor on the name of a local variable and selecting **Navigation** → **Next use** (or hitting **Meta-Shift-Right**) transports you to the next use of this variable in the code. (Note that this doesn't just search for the next occurrence of the variable name but in fact takes into account that variables with the same name but in different scopes are different.) The same works for the use of function names. Selecting **Navigation** → **Previous use** (or hitting **Meta-Shift-Left**) transports you to the previous use of a symbol.

> NOTE
> To see the list of all uses of a name through which these commands cycle, place the cursor onto it and open the **Code Browser** tool view or press and hold the **Alt** button. This is explained in more detail in the section on Exploring code.

- The **context list**: Web browsers have this feature where you can go backward and forward in the list of most recently visited web pages. KDevelop has the same kind of features, except that instead of web pages you visit *contexts*. A context is the current location of the cursor, and you change it by navigating away from it using anything but cursor commands — for example, by clicking on a location provided by a tooltip, in the **Code Browser** tool view, one of the options given in the **Navigation** menu, or any other navigation command. Using the **Navigation** → **Previous Visited Context** (**Meta-Left**) and **Navigation** → **Next Visited Context** (**Meta-Right**) transports you along this list of visited contexts just like the **back** and **forward** buttons of a browser transports you to the previous or next webpage in the list of visited pages.
- Finally, there are tool views that allow you to navigate to different places in your code base. For example, the **Classes** tool provides you with a list of all namespaces and classes in all projects of the current session, and allows you to expand it to see member functions and variables of each of these classes:



Double-clicking on an item (or going through the context menu using the right mouse button) allows you to jump to the location of the declaration of the item. Other tools allow similar things; for example, the **Projects** tool view provides a list of files that are part of a session:



Again, double-clicking on a file opens it.

3.4 Writing source code

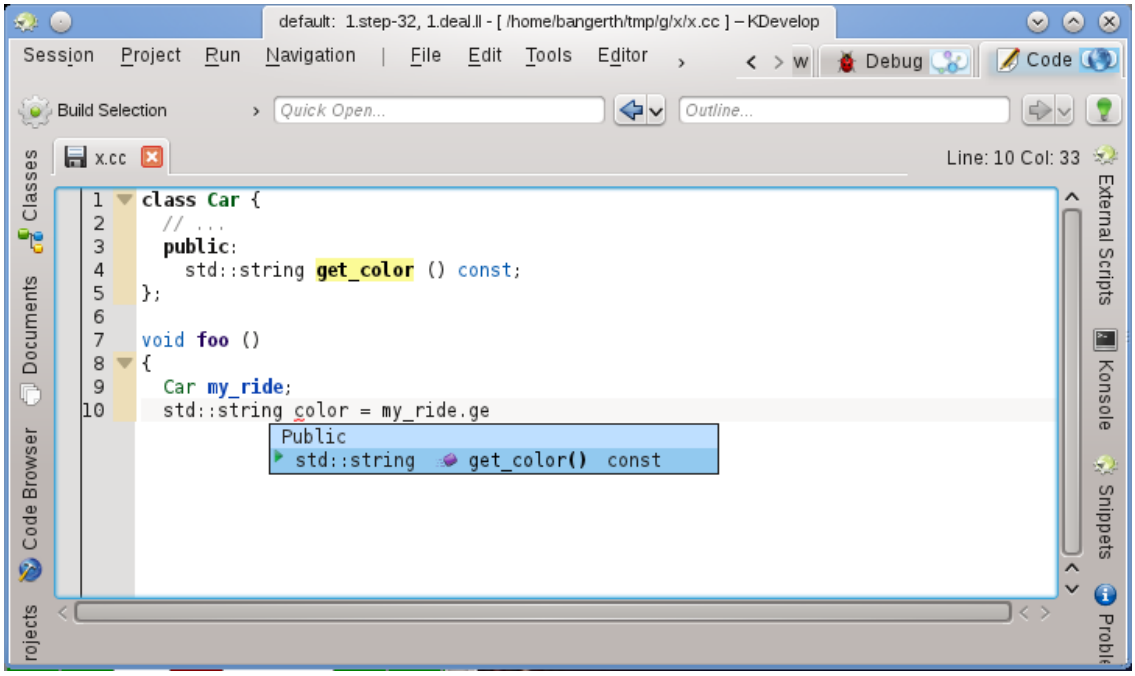
Because KDevelop understands your projects' source code, it can assist in writing more code. The following outlines some of the ways in which it does that.

3.4.1 Auto-completion

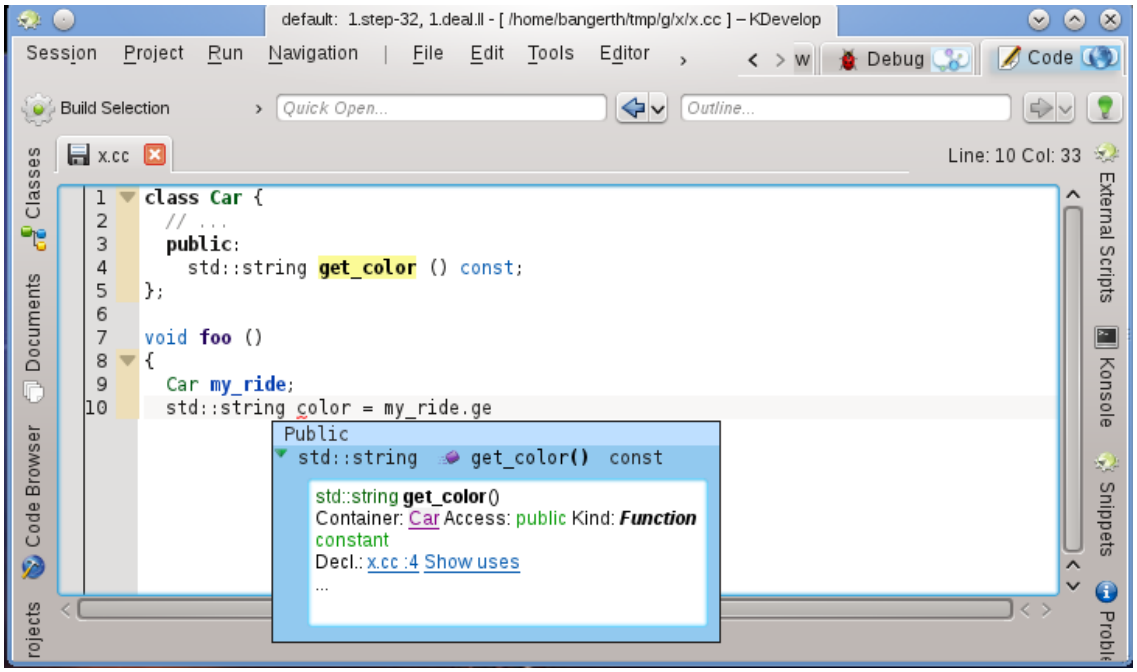
Probably the most useful of all features in writing new code is auto-completion. Consider, for example, the following piece of code:

```
class Car {
  // ...
  public:
    std::string get_color () const;
};
void foo()
{
  Car my_ride;
  // ...do something with this variable...
  std::string color = my_ride.ge
```

In the last line, KDevelop will remember that the variable `my_ride` is of type `Car`, and will automatically offer to complete the name of the member function `ge` as `get_color`. In fact, all you have to do is to keep typing until the auto-completion feature has reduced the number of matches to one, and then hit the **Enter** key:



Note that you can click on the tool-tip to get more information about the function apart from its return type and whether it is public:



Auto-completion can save you a lot of typing if your project uses long variable and function names; furthermore, it avoids mis-spelling names (and the resulting compiler errors) and it makes it much simpler to remember the exact names of functions; for example, if all of your getters start with `get_`, then the auto-completion feature will be able to only present you a list of possible getters when you have typed the first four letters, likely reminding you in the process which of the functions is the correct one. Note that for auto-completion to work, neither the declaration of the `Car` class nor of the `my_ride` variable need to be in the same file as where you are currently writing code. KDevelop simply has to know that these classes and variables are connected, i.e. the files in which these connections are made need to be part of the project you are currently working on.

> **NOTE**
> KDevelop doesn't always know when to assist you in completing code. If the auto-completion tooltip doesn't automatically open, hit **Ctrl**-**Space** to open a list of completions manually. In general, in order for auto-completion to work, KDevelop needs to parse your source files. This happens in the background for all files that are part of the projects of the current session after you start KDevelop, as well as while after you stop typing for a fraction of a second (the delay can be configured).

> **NOTE**
> KDevelop only parses files that it considers source code, as determined by the MIME-type of the file. This type isn't set before the first time a file is saved; consequently, creating a new file and starting to write code in it will not trigger parsing for auto-completion until after it is saved for the first time.

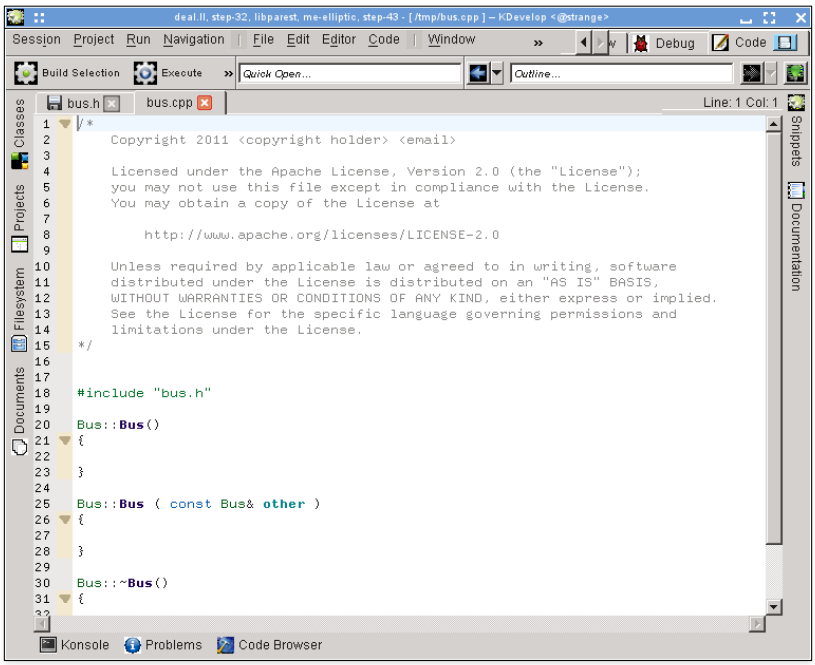
> **NOTE**
> As in the previous note, for auto-completion to work, KDevelop must be able to find declarations in header files. For this, it searches in a number of default paths. If it doesn't automatically find a header file, it will underline the name of a header file in red; in that case, right click on it to tell KDevelop explicitly where to find these files and the information they provide.

> **NOTE**
> Configuring auto-completion is discussed in this section of this manual.

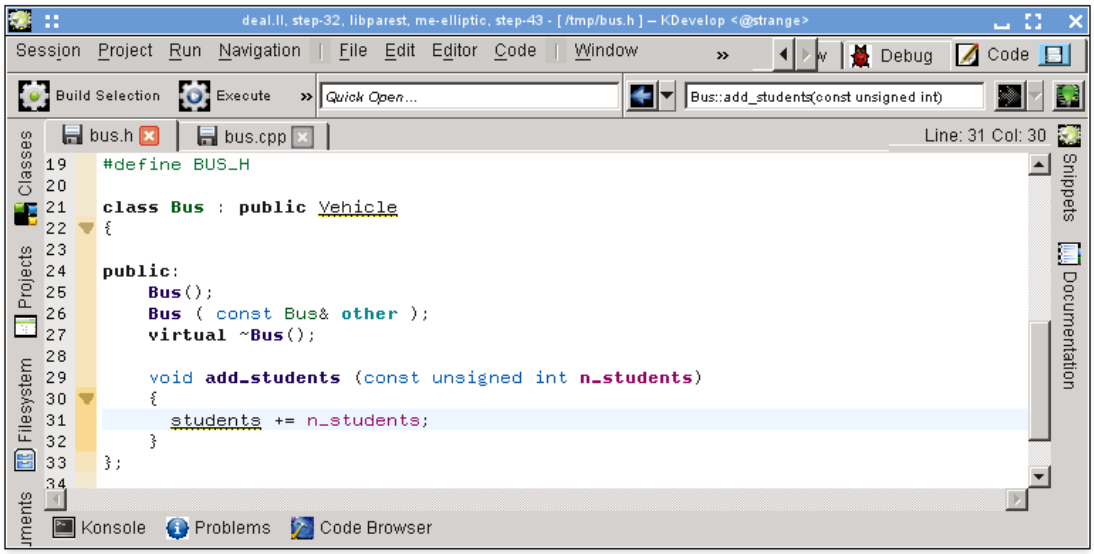
3.4.2 Adding new classes and implementing member functions

KDevelop has an assistant for adding new classes. The procedure is described in Creating a new class. A simple C++ class can be created by choosing the Basic C++ template from the `Class` category. In the assistant, we can choose some predefined member functions, for example an empty constructor, a copy constructor and a destructor.

After completing the assistant, the new files are created and opened in the editor. The header file already has include guards and the new class has all the member functions we selected. The next two steps would be to document the class and its member functions and to implement them. We will discuss aids for documenting classes and functions below. To implement the special functions already added, simply go to the **bus.cpp** tab where the skeleton of functions are already provided:

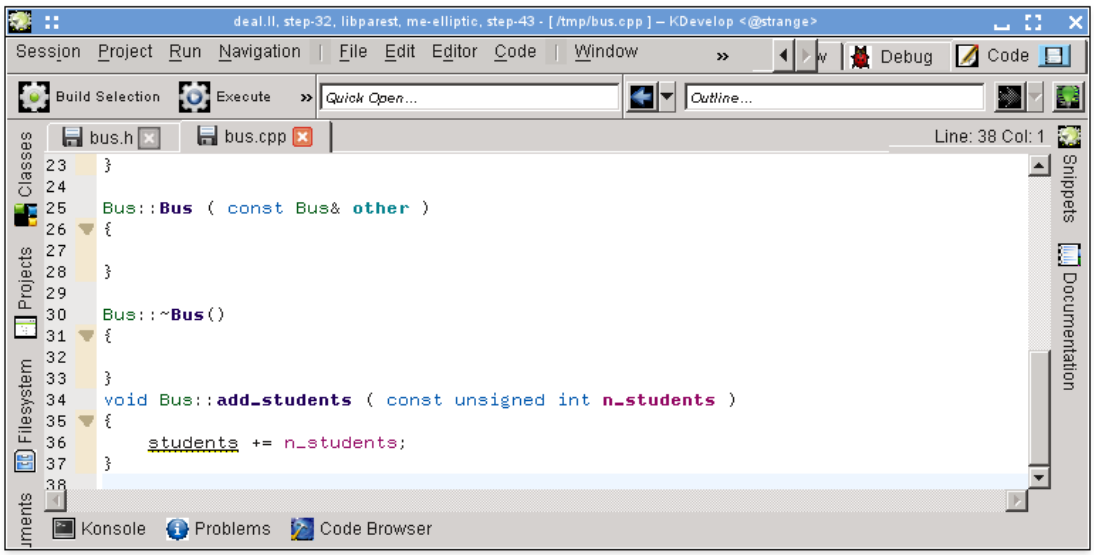


To add new member functions, go back to the **bus.h** tab and add the name of a function. For example, let us add this:



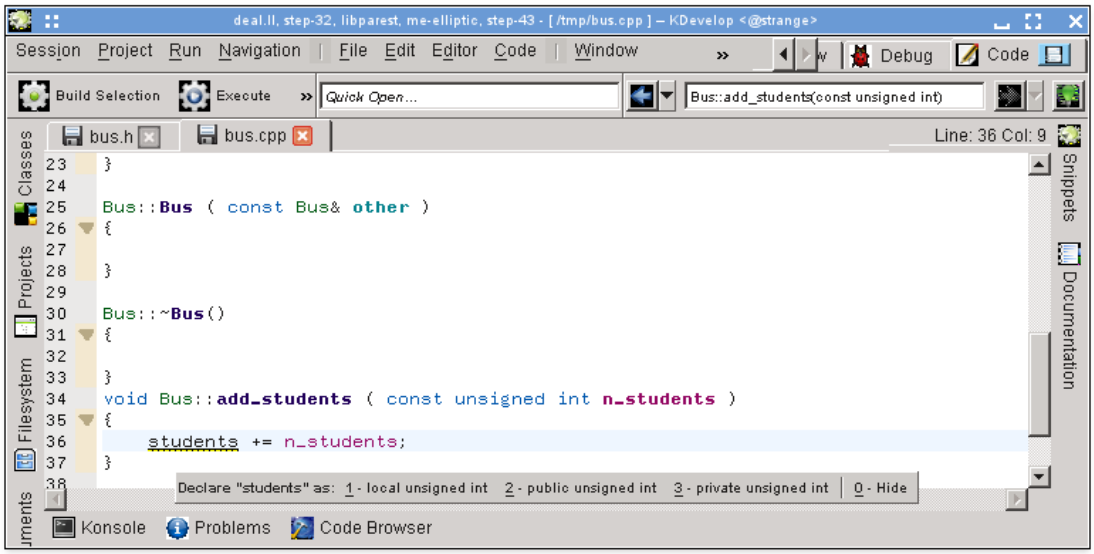

```
19  #define BUS_H
20
21  class Bus : public Vehicle
22  {
23
24  public:
25      Bus();
26      Bus ( const Bus& other );
27      virtual ~Bus();
28
29      void add_students (const unsigned int n_students)
30      {
31        students += n_students;
32      }
33  };
34
```

Note how I have already started with the implementation. However, in many coding styles, the function shouldn't be implemented in the header file but rather in the corresponding .cpp file. To this end, locate the cursor on the name of the function and select **Code** → **Move to source** or hit **Ctrl-Alt-S**. This remove the code between curly braces from the header file (and replaces it by a semicolon as necessary to terminate the function declaration) and moves it into the source file:

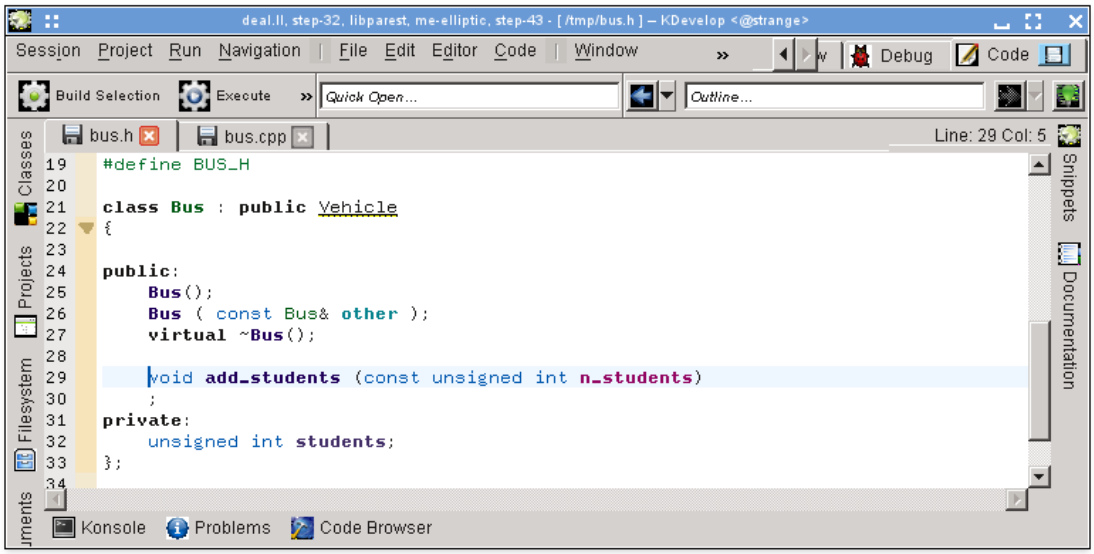



```
23  }
24
25  Bus::Bus ( const Bus& other )
26  {
27
28  }
29
30  Bus::~Bus()
31  {
32
33  }
34  void Bus::add_students ( const unsigned int n_students )
35  {
36      students += n_students;
37  }
38
```

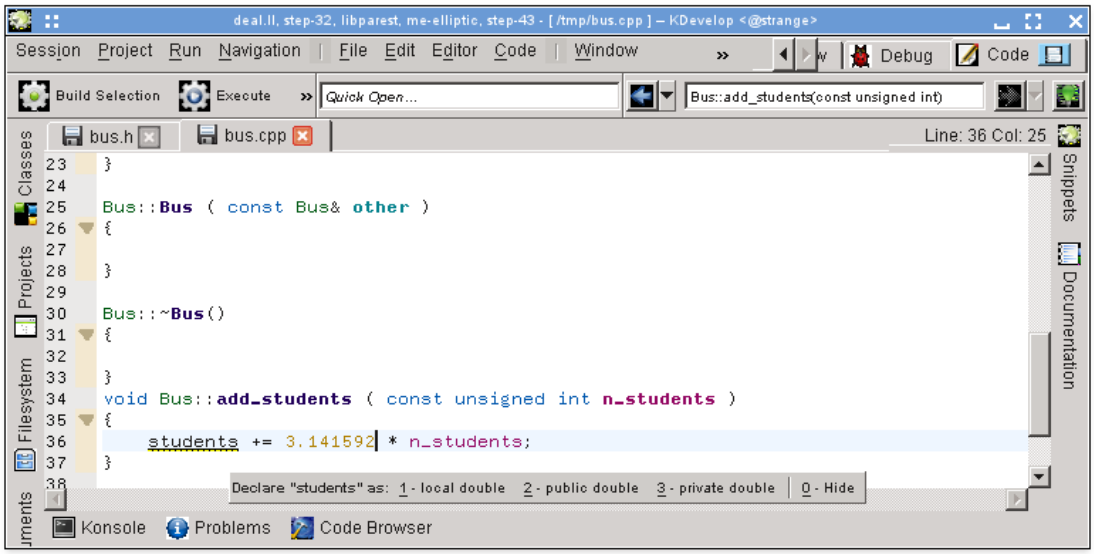
Note how I have just started typing and that I meant to imply that the `students` variable should probably be a member variable of class `Bus` but that I haven't yet added it. Note also how KDevelop underlines it to make clear that it doesn't know anything about the variable. But this problem can be solved: Clicking on the variable name yields the following tooltip:



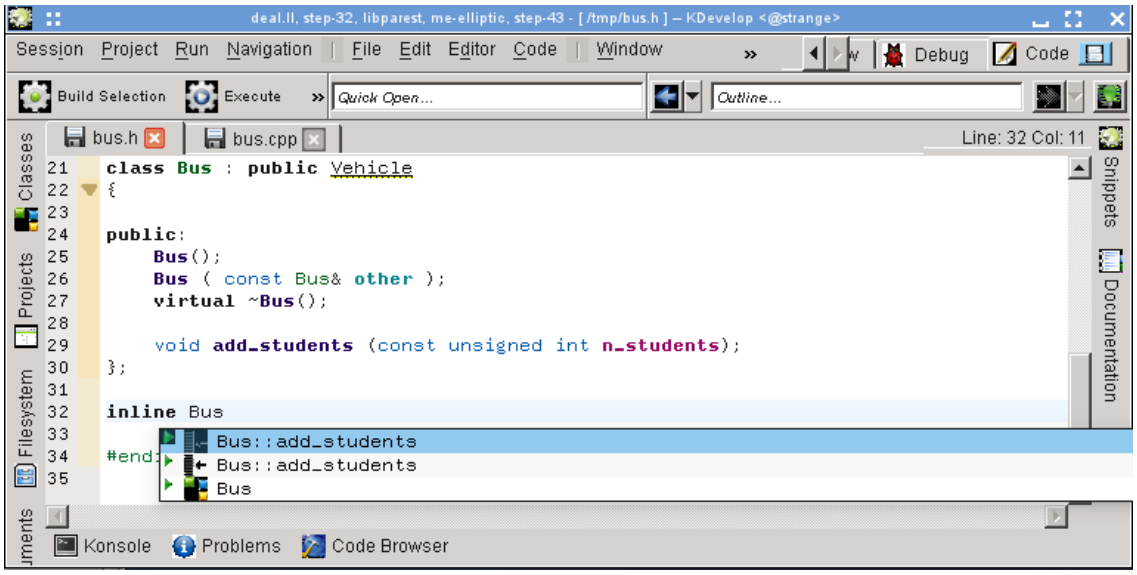
(The same can be achieved by right clicking on it and selecting **Solve: Declare As**.) Let me select '3 - private unsigned int' (either by mouse, or by hitting **Alt-3**) and then see how it comes out in the header file:



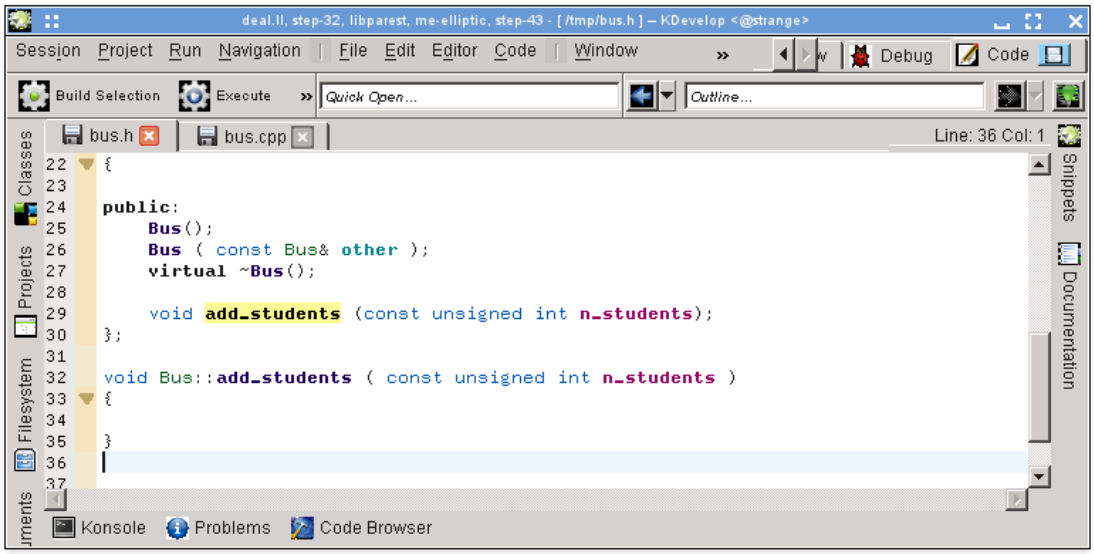
It is worth noting that KDevelop extracts the type of the variable to be declared from the expression used to initialize it. For example, if we had written the addition in the following rather dubious way, it would had suggested to declare the variable as type `double`:



As a final point: The method using **Code** → **Move to source** does not always insert the new member function where you may want it. For example, you may want it to be marked as `inline` and place it at the bottom of the header file. In a case like this, write the declaration and the start writing the definition of the function like this:



KDevelop automatically offers all possible completions of what might come here. Selecting one of the two `add_students` entries yields the following code that already fills in the complete argument list:




```
{

public:
    Bus();
    Bus ( const Bus& other );
    virtual ~Bus();

    void add_students (const unsigned int n_students);
};

void Bus::add_students ( const unsigned int n_students )
{

}
```

NOTE

In the example, accepting one of the choices the auto-completion tool offers yields the correct signature but unfortunately deletes the `inline` marker already written. This has been reported as KDevelop Bug 274245.

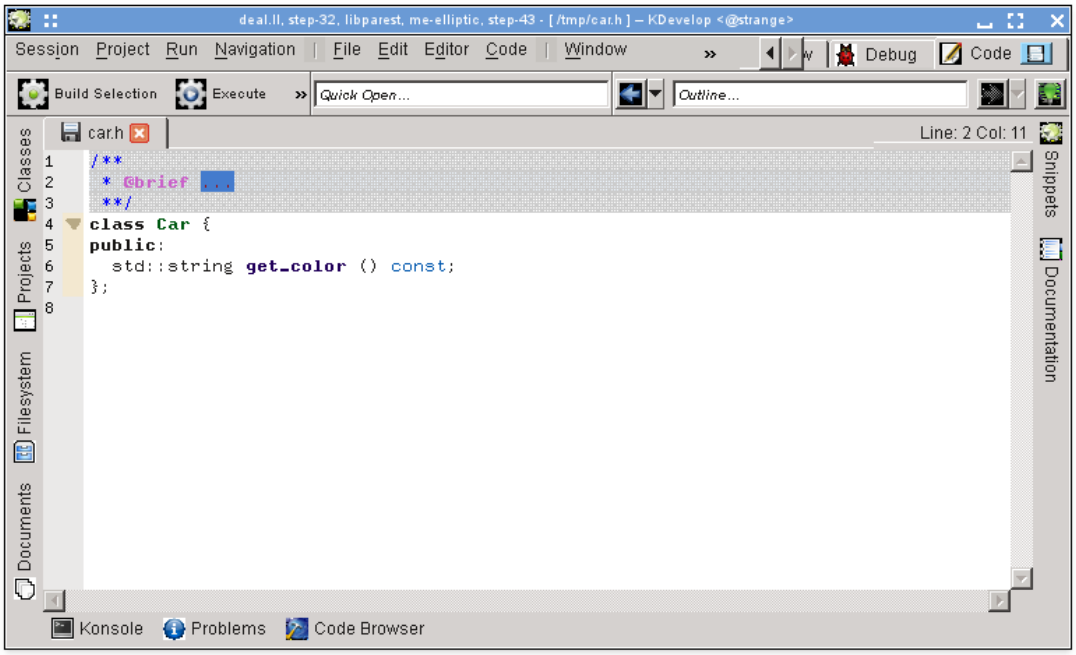
3.4.3 Documenting declarations

Good code is well documented, both at the level of the implementation of algorithms within in functions as well as at the level of the interface — i.e., classes, (member and global) functions, and (member or global) variables need to be documented to explain their intent, possible values of arguments, pre- and postconditions, etc. As far as documenting the interface is concerned, doxygen has become the de facto standard for formatting comments that can then be extracted and displayed on searchable webpages.

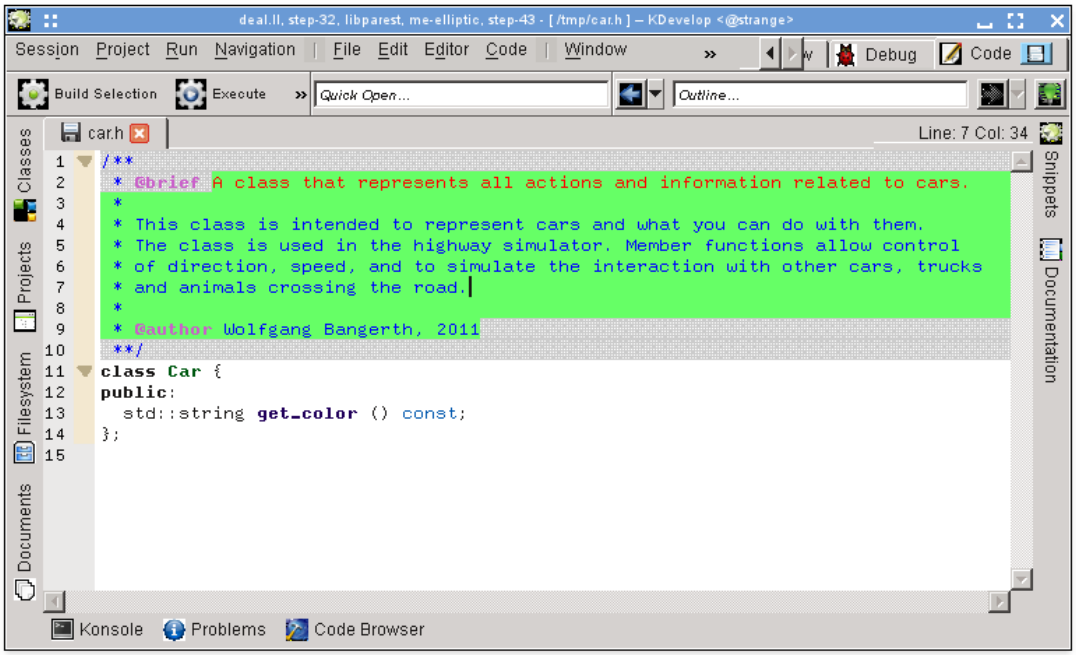
KDevelop supports this style of comments by providing a short cut to generate the framework of comments that document a class or member function. For example, assume you have already written this code:

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

You now want to add documentation to both the class and the member function. To this end, move the cursor onto the first line and select **Code** → **Document Declaration** or hit **Alt-Shift-D**. KDevelop will respond with the following:

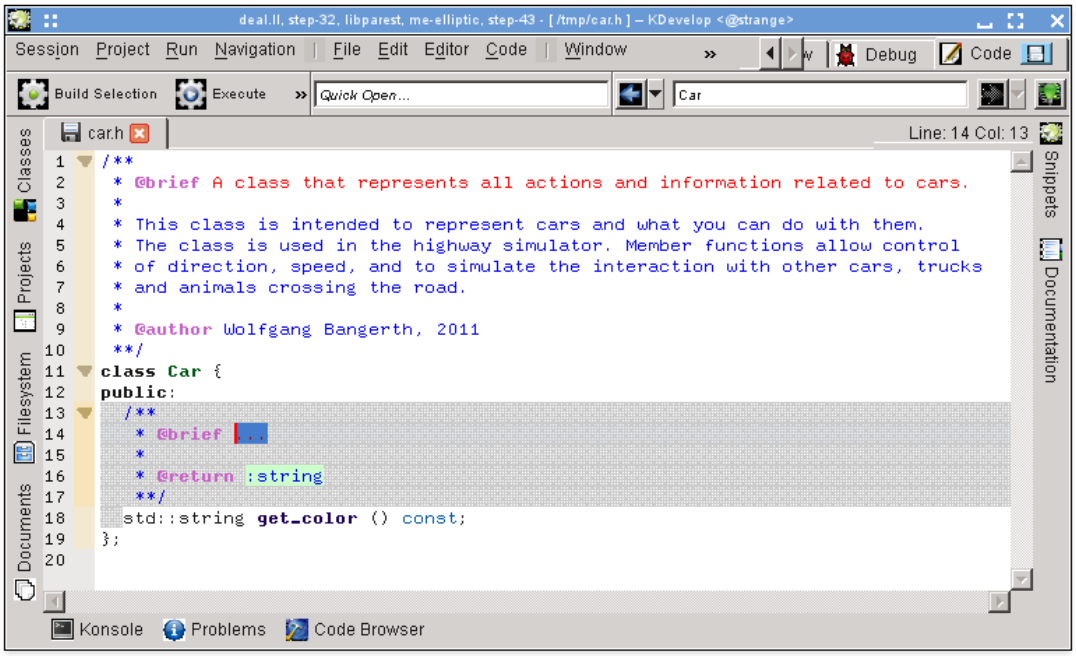


The cursor is already in the grayed out area for you to fill in the short description (after the doxygen keyword `@brief`) of this class. You can then continue to add documentation to this comment that gives a more detailed overview of what the class does:

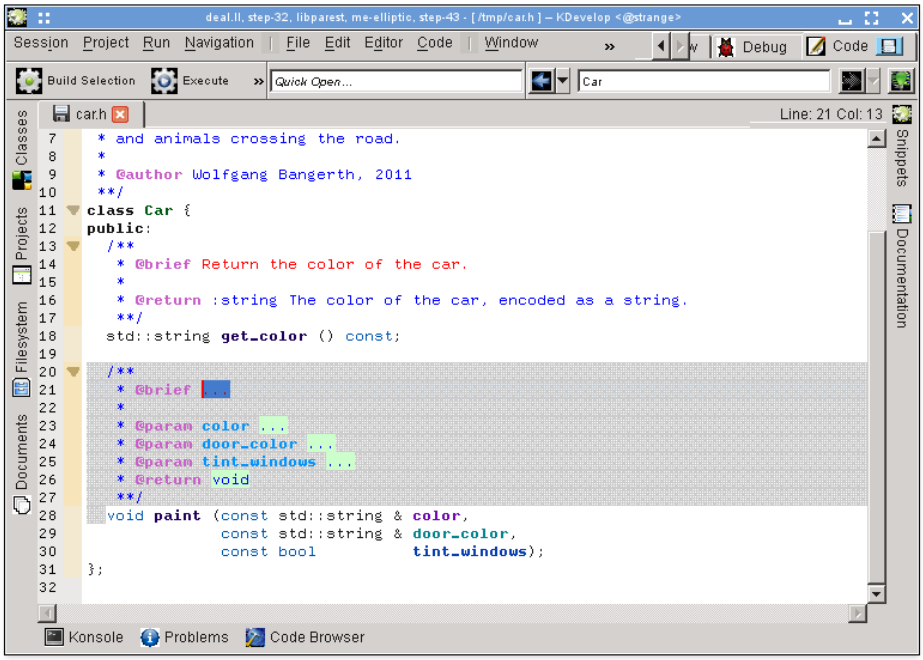


While the editor is inside the comment, the comment text is highlighted in green (the highlighting disappears once you move the cursor out of the comment). When you get to the end of a line, hit **Enter** and KDevelop will automatically start a new line that starts with an asterisk and place the cursor one character indented.

Now let's document the member function, again by putting the cursor on the line of the declaration and selecting **Code** → **Document Declaration** or hitting **Alt-Shift-D**:



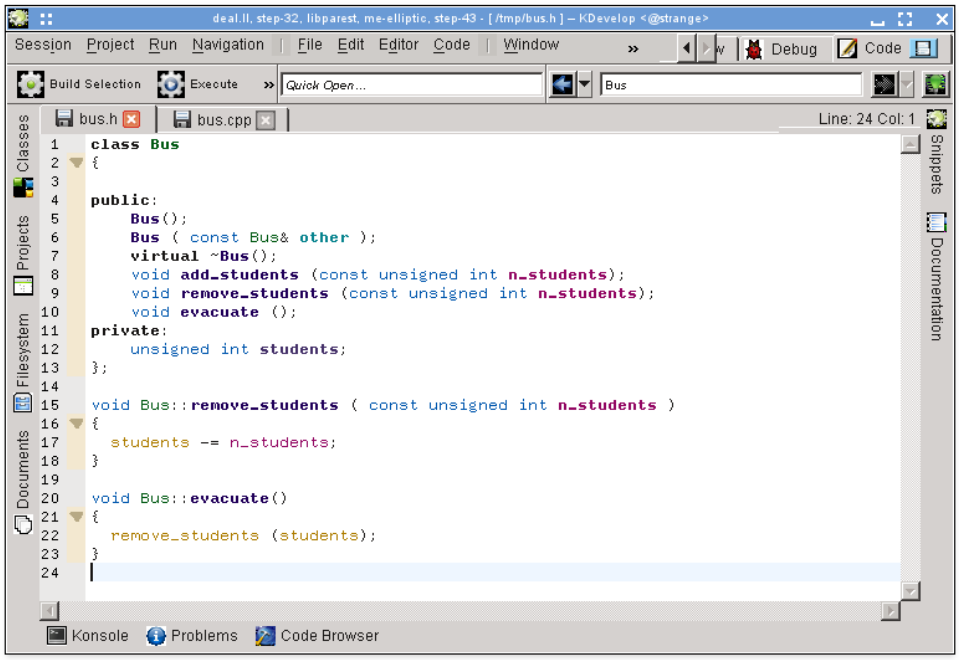
Again, KDevelop automatically generates the skeleton of a comment, including documentation for the function itself, as well as its return type. In the current case, the name of the function is pretty much self-explanatory, but oftentimes function arguments may not be and should be documented individually. To illustrate this, let's consider a slightly more interesting function and the comment KDevelop automatically generates:



Here, the suggested comment already contains all the Doxygen fields for the individual parameters, for example.

3.4.4 Renaming variables, functions and classes

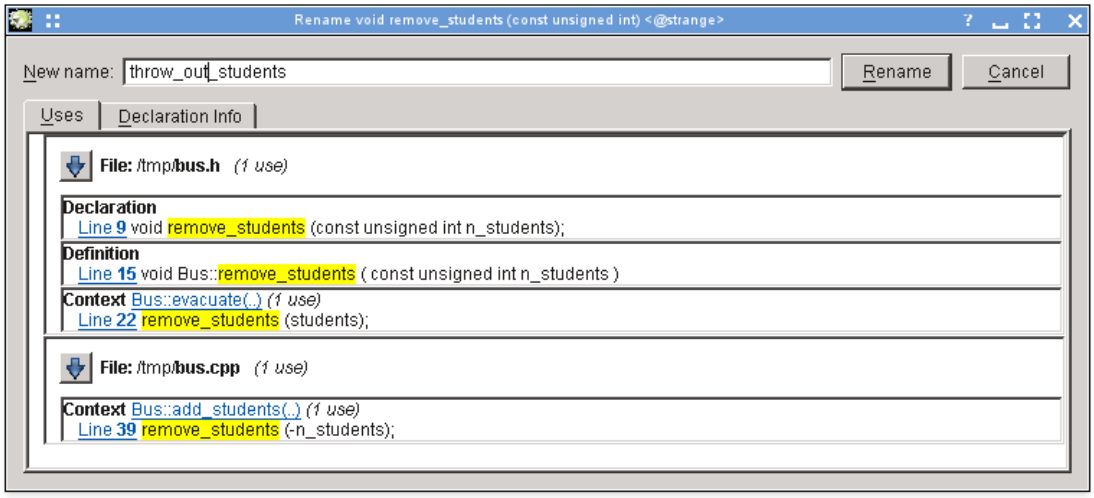
Sometimes, one wants to rename a function, class or variable. For example, let's say we already have this:



We then realize that we're unhappy with the name `remove_students` and would have rather called it, say, `throw_out_students`. We could do a search-replace for the name, but this has two drawbacks:

- The function may be used in more than one file.
- We really only want to rename this function and not touch functions that may have the same name but are declared in other classes or namespaces.

Both these problems can be solved by moving the cursor on any of the occurrences of the name of the function and selecting **Code** → **Rename declaration** (or right clicking on the name and selecting **Rename Bus::remove_students**). This brings up a dialog box where you can enter the new name of the function and where you can also see all the places where the function is actually used:

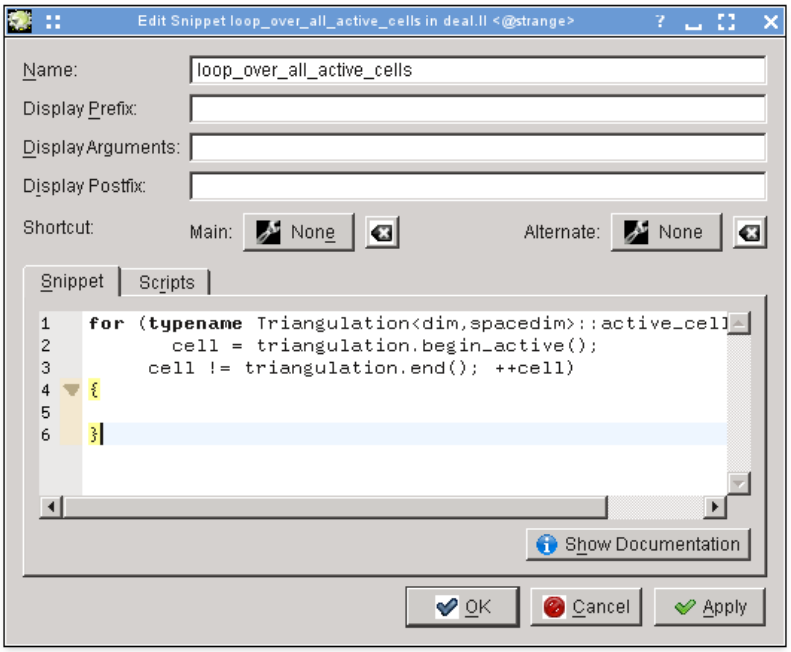


3.4.5 Code snippets

Most projects have pieces of code that one frequently has to write in source code. Examples are: for compiler writers, a loop over all instructions; for user interface writers, checks that user input is valid and if not to open an error box; in the project of the author of these lines, it would be code of the kind

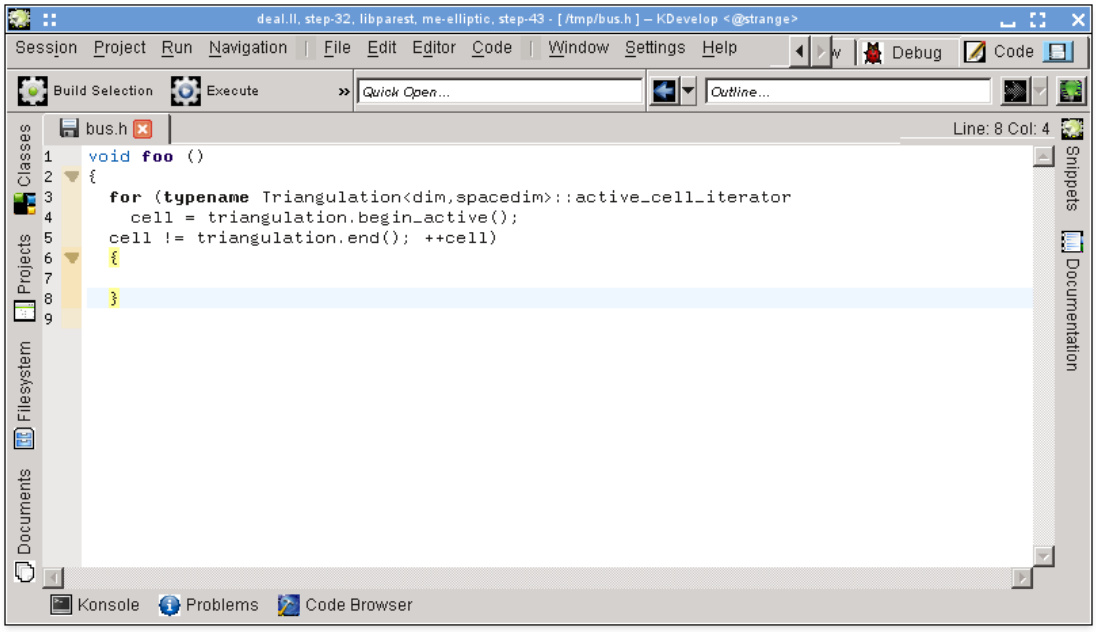
```
for (typename Triangulation::active_cell_iterator
       cell = triangulation.begin_active();
     cell != triangulation.end(); ++cell)
  ... do something with the cell ...
```

Rather than typing this kind of text over and over again (with all the concomitant typos one introduces), the **Snippets** tool of KDevelop can help here. To this end, open the tool view (see Tools and views if the corresponding button isn't already on the perimeter of your window). Then click on the 'Add repository' button (a slight misnomer — it allows you to create a named collection of snippets for source codes of a particular kind, e.g. C++ sources) and create an empty repository. Then click to add a snippet, to get a dialog like the following:



> **NOTE**
> The name of a snippet may not have spaces or other special characters because it must look like a normal function or variable name (for reasons that will become clear in the next paragraph).

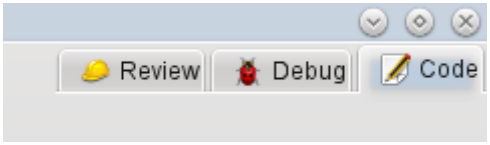
To use a snippet so defined, when you are editing code, you can just type the name of the snippet like you would any other function or variable name. This name will become available for auto-completion — which means that there is no harm in using a long and descriptive name for a snippet such as the one above — and when you accept the suggestion of the auto-completion tooltip (for example by just hitting **Enter**), the already entered part of the snippets' name will be replaced by the full expansion of the snippet and will be properly indented:



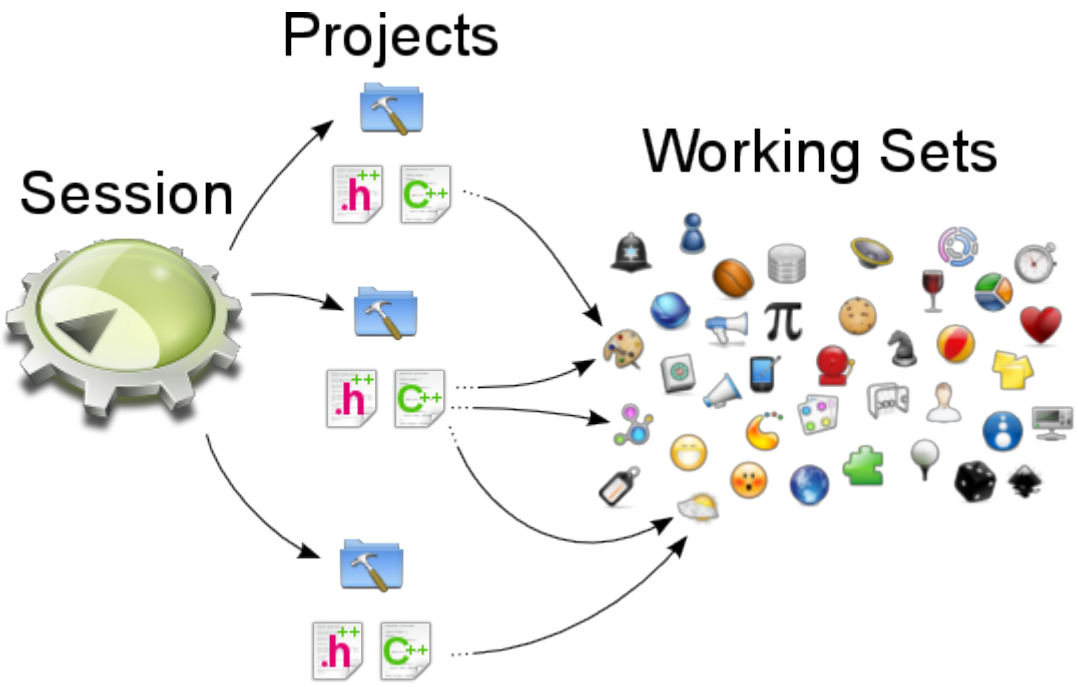
Note that for this to work, the **Snippets** tool view need not be open or visible: you only ever need the tool view to define new snippets. An alternative, if less convenient, way to expand a snippet is to simply click on it in the respective tool view.

> NOTE
> Snippets are much more powerful than just explained. For a full description of what you can do with them, see the detailed documentation of the Snippets tool.

3.5 Modes and working sets

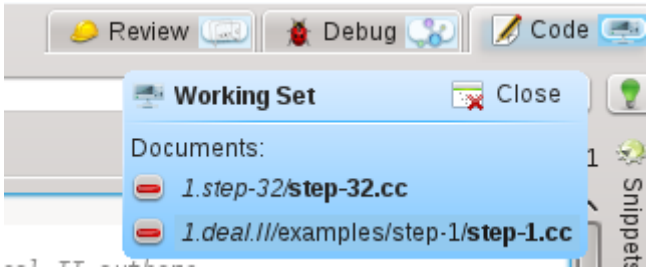


If you've gotten this far, take a look at the upper right of the KDevelop main window: As shown in the picture, you will see that there are three **modes** KDevelop can be in: **Code** (the mode we discuss in the current chapter on working with source code), **Debug** (see Debugging programs) and **Review** (see Working with version control systems).



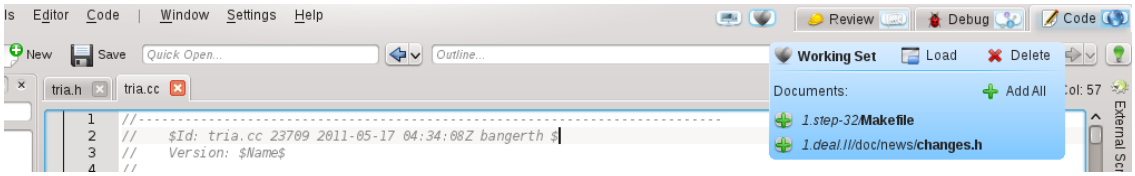
Each mode has its own set of tools that are stacked around the perimeter, and each mode also has a *working set* of currently open files and documents. Furthermore, each such working set is associated with a current session, i.e. we have the relationship shown above. Note that the files in the working set come from the same session, but they may come from different projects that are part of the same session.

If you open KDevelop the first time, the working set is empty — there are no open files. But as you open files for editing (or debugging, or reviewing in the other modes) your working set grows. The fact that your working set is non-empty is indicated by a symbol in the tab, as shown below. You will notice that whenever you close KDevelop and later start it again, the working set is saved and restored, i.e. you get the same set of open files.



If you hover your mouse over the symbol for the working set, you get a tooltip that shows you which files are currently open in this working set (here: the `step-32.cc` and `step-1.cc` files). Clicking on the red minus sign closes the tab for the corresponding file. Maybe more importantly, clicking on the correspondingly named button allows you to **close** the entire working set at once (i.e. to close all currently open files). The point about closing a working set, however, is that it doesn't just close all files, it actually saves the working set and opens a new, still empty one. You can see this here:

Note the two symbols to the left of the three mode tabs (the heart and the unidentifiable symbol to its left). Each of these two symbols represents a saved working set, in addition to the currently open working set. If you hover your mouse over the heart symbol, you'll get something like this:



It shows you that the corresponding working set contains two files and their corresponding project names: `Makefile` and `changes.h`. Clicking **Load** will close and save the current working set (which as shown here has the files `tria.h` and `tria.cc` open) and instead open the selected working set. You can also permanently delete a working set, which removes it from the set of saved working sets.

3.6 Some useful keyboard shortcuts

KDevelop's editor follows the standard keyboard shortcuts for all usual editing operations. However, it also supports a number of more advanced operations when editing source code, some of which are bound to particular key combinations. The following are frequently particularly helpful:

Jumping around in code	
Ctrl-Alt-O	Quick open file: enter part of a filename and select among all the files in the current session's projects' directory trees that match the string; the file will then be opened
Ctrl-Alt-C	Quick open class: enter part of a class name and select among all class names that match; the cursor will then jump to the class declaration
Ctrl-Alt-M	Quick open function: enter part of a (member) function name and select among all names that match; note that the list shows both declarations and definitions and the cursor will then jump to the selected item
Ctrl-Alt-Q	Universal quick open: type anything (file name, class name, function name) and get a list of anything that matches to select from
Ctrl-Alt-N	Outline: Provide a list of all things that are happening in this file, e.g. class declarations and function definitions
Ctrl-,	Jump to definition of a function if the cursor is currently on a function declaration
Ctrl-.	Jump to declaration of a function or variable if the cursor is currently in a function definition
Ctrl-Alt-PageDown	Jump to next function
Ctrl-Alt-PageUp	Jump to previous function
Ctrl-G	Goto line

Searching and replacing	
Ctrl-F	Find
F3	Find next
Ctrl-R	Replace
Ctrl-Alt-F	Find-Replace in multiple files

Other things	
Ctrl-_	Collapse one level: remove this block from view, for example if you want to focus on the bigger picture within a function
Ctrl-+	Expand one level: undo the collapsing
Ctrl-D	Comment out selected text or current line
Ctrl-Shift-D	Comment in selected text or current line
Alt-Shift-D	Document the current function. If the cursor is on a function or class declaration then hitting this key will create a doxygen-style comment pre-populated with a listing of all parameters, return values, etc.
Ctrl-T	Transpose the current and the previous character
Ctrl-K	Delete the current line (note: this is not just emacs' 'delete from here to the end of the line')

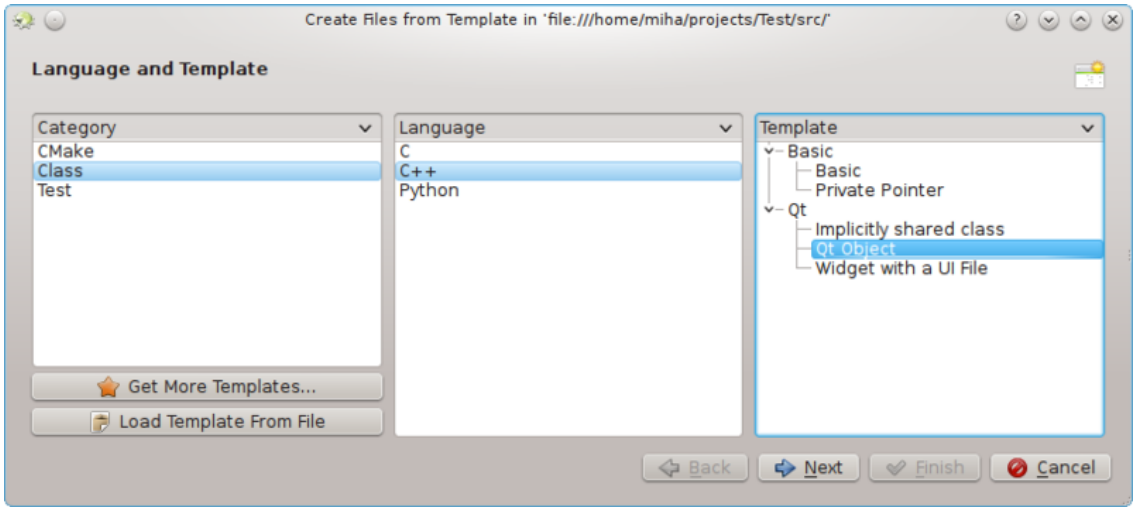
Chapter 4

Code generation with templates

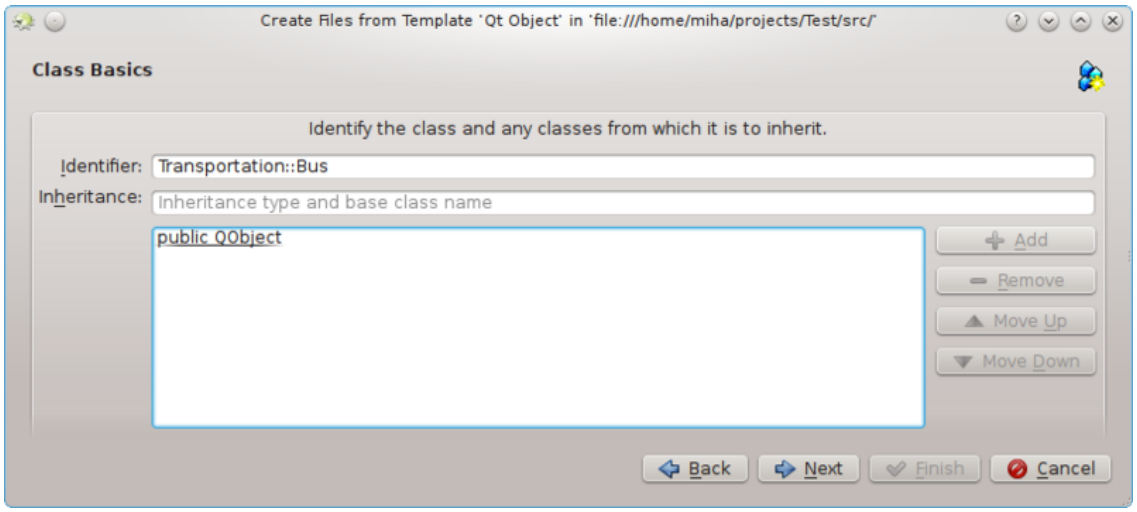
KDevelop uses templates for generating source code files and to avoid writing repeatable code.

4.1 Creating a new class

The most common use for code generation is probably writing new classes. To create a new class in an existing project, right click on a project folder and choose **Create from Template...**. The same dialog can be started from the menu by clicking **File** → **New from Template...**, but using a project folder has the benefit of setting a base URL for the output files. Choose `Class` in the category selection view, and the desired language and template in the other two views. After you have selected a class template, you will have to specify the details of the new class.

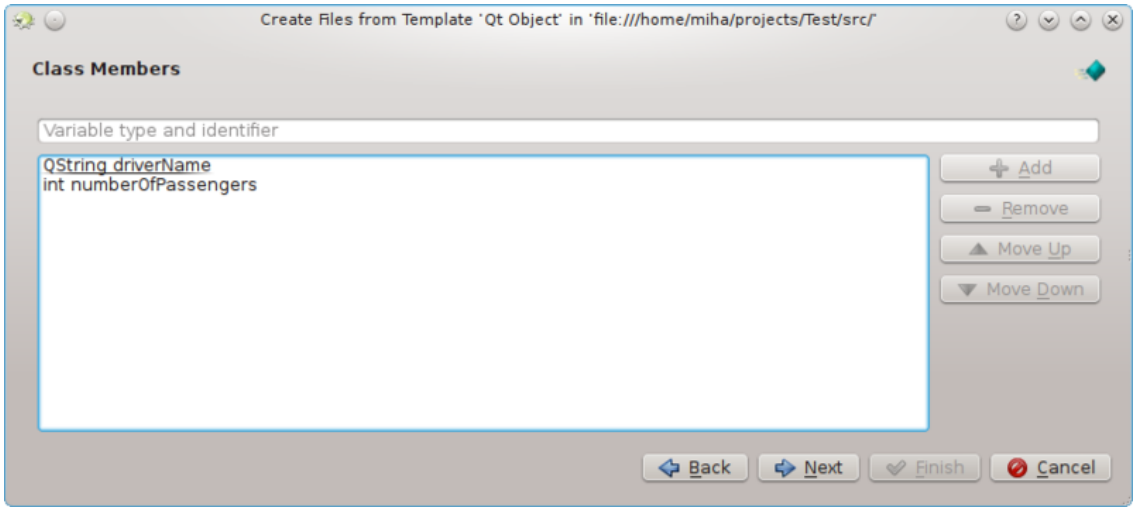


First you have to specify an identifier for the new class. This can be a simple name (like `Bus`) or a complete identifier with namespaces (like `Transportation::Bus`). In the latter case, KDevelop will parse the identifier and correctly separate the namespaces from the actual name. On the same page, you can add base classes for the new class. You may notice that some templates choose a base class on their own, you are free to remove it and/or add other bases. You should write the full inheritance statement here, which is language-dependent, such as `public QObject` for C++, `extends SomeClass` for PHP or simply the name of the class for Python.



In the next page, you are offered a selection of virtual methods from all inherited classes, as well as some default constructors, destructors and operators. Checking the check box next to a method signature will implement this method in the new class.

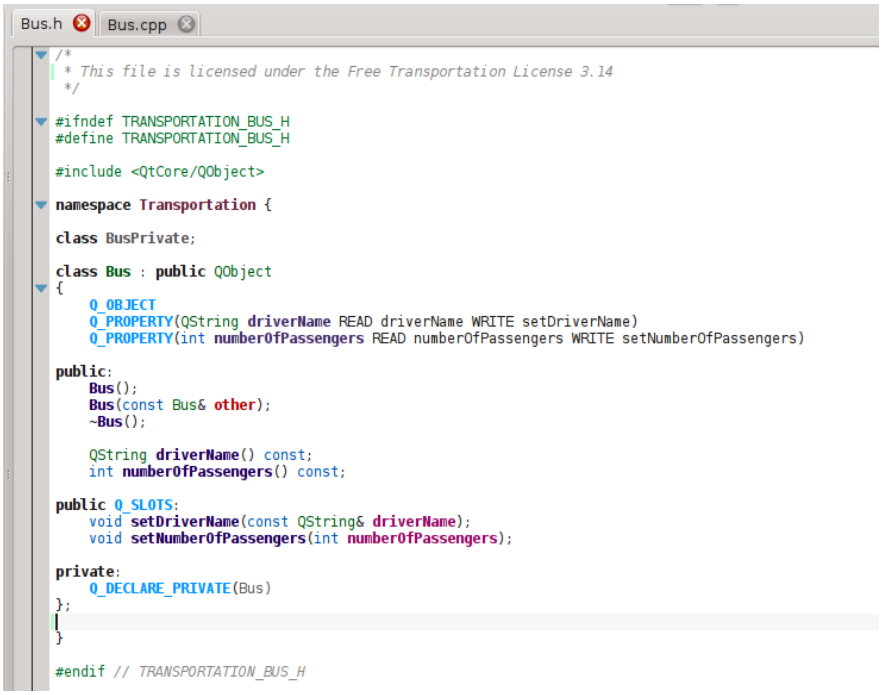
Clicking **Next** brings up a page where you can add members to a class. Depending on the selected template, these may appear in the new class as member variables, or the template may create properties with setters and getters for them. In a language where variable types have to be declared, such as C++, you have to specify both the type and the name of the member, such as `int number` or `QString name`. In other languages, you may leave out the type, but it is good practice to enter it anyway, because the selected template could still make some use of it.



In the following pages, you can choose a license for you new class, set any custom options required by the selected template, and configure output locations for all the generated files. By clicking **Finish**, you complete the assistant and create the new class. The generated files will be opened in the editor, so you can start adding code right away.

After creating a new C++ class, you will be given an option of adding the class to a project target. Choose a target from the dialog page, or dismiss the page and add the files to a target manually.

If you chose the `Qt Object` template, checked some of the default methods, and added two member variables, the output should look like on the following picture.



You can see that data members are converted into Qt properties, with accessor functions and the Q_PROPERTY macros. Arguments to setter functions are even passed by const-reference, where appropriate. Additionally, a private class is declared, and a private pointer created with Q_DECLARE_PRIVATE. All this is done by the template, choosing a different template in the first step could completely change the output.

4.2 Creating a new unit test

Even though most testing frameworks require each test to also be a class, KDevelop includes a method to simplify the creation of unit tests. To create a new test, right click on a project folder and choose **Create from Template...**. In the template selection page, choose `Test` as the category, then choose your programming language and template and click **Next**.

You will be prompted for the test name and a list of test cases. For the test cases, you only have to specify a list of names. Some unit testing frameworks, such as PyUnit and PHPUnit, require that test cases start with a special prefix. In KDevelop, the template is responsible for adding the prefix, so you do not have to prefix the test cases here. After clicking **Next**, specify the license and output locations for the generated files, and the test will be created.

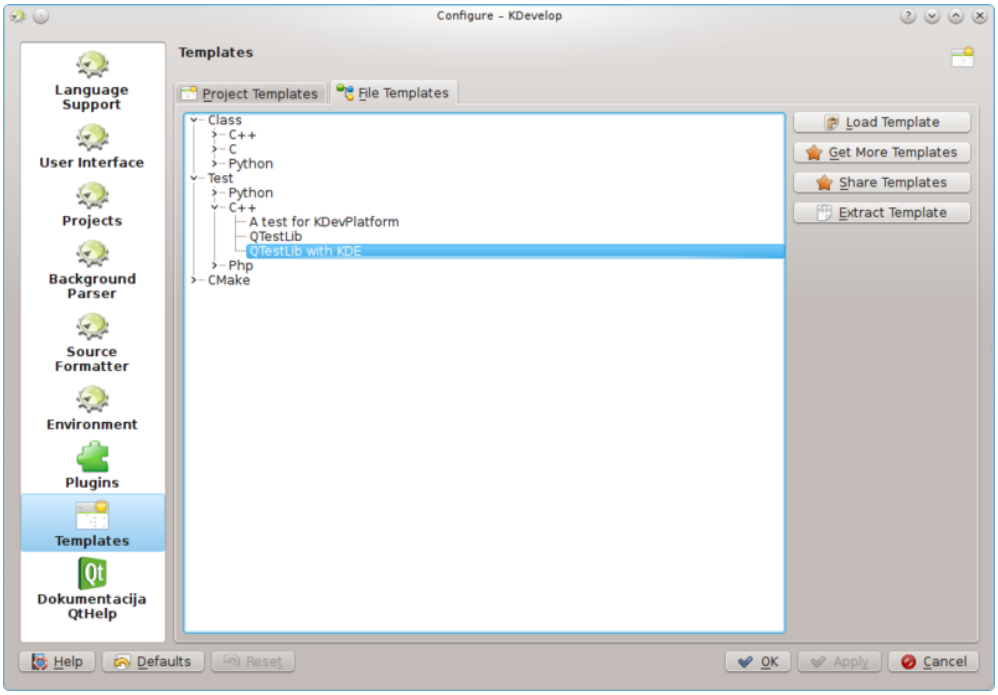
Unit tests created this way will not be added to any target automatically. If you are using CTest or some other testing framework, make sure to add the new files to a target.

4.3 Other files

While classes and unit tests receive special attention when generating code from templates, the same method can be used for any kind of source code files. For example, one could use a template for a CMake Find module or a .desktop file. This can be done by choosing **Create from Template...**, and selecting the wanted category and template. If the selected category is neither `C lass` nor `Test`, you will only have the option of choosing the license, any custom options specified by the template, and the output file locations. As with classes and tests, finishing the assistant will generate the files and open them in the editor.

4.4 Managing templates

From the **File** → **New from Template...** assistant, you can also download additional file templates by clicking the **Get more Templates...** button. This opens a Get Hot New Stuff dialog, where you can install additional templates, as well as update or remove them. There is also a configuration module for templates, which can be reached by clicking **Settings** → **Configure KDevelop** → **Templates**. From there, you can manage both file templates (explained above) and project templates (used for creating new projects).



Of course, if none of the available template suit your project, you can always create new ones. The easiest way is probably to copy and modify an existing template, while a short tutorial and a longer specification document are there to help you. To copy an installed template, open the template manager by clicking **Settings** → **Configure KDevelop...** → **Templates**, select the template you wish to copy, then click the **Extract Template** button. Select a destination folder, then click **OK**, and the contents of the template will be extracted into the selected folder. Now you can edit the template by opening the extracted files and modifying them. After you are done, you can import your new template into KDevelop by opening the template manager, activating the appropriate tab (either **Project Templates** or **File Templates**) and clicking **Load Template**. Open the template description file, which is the one with the suffix either `.kdevtemplate` or `.desktop`. KDevelop will compress the files into a template archive and import the template.

> **NOTE**
> When copying an existing template, make sure you rename it before importing it again. Otherwise, you will either overwrite the old template, or will end up with two templates with identical names. To rename a template, rename the description file to something unique (but keep the suffix), and change the `Name` entry in the description file.

If you want to write a template from scratch, you can start with a sample C++ class template by creating a new project and selecting the `C++ Class Template` project in category `KDevelop`.

Chapter 5

Building (compiling) projects with custom Makefiles

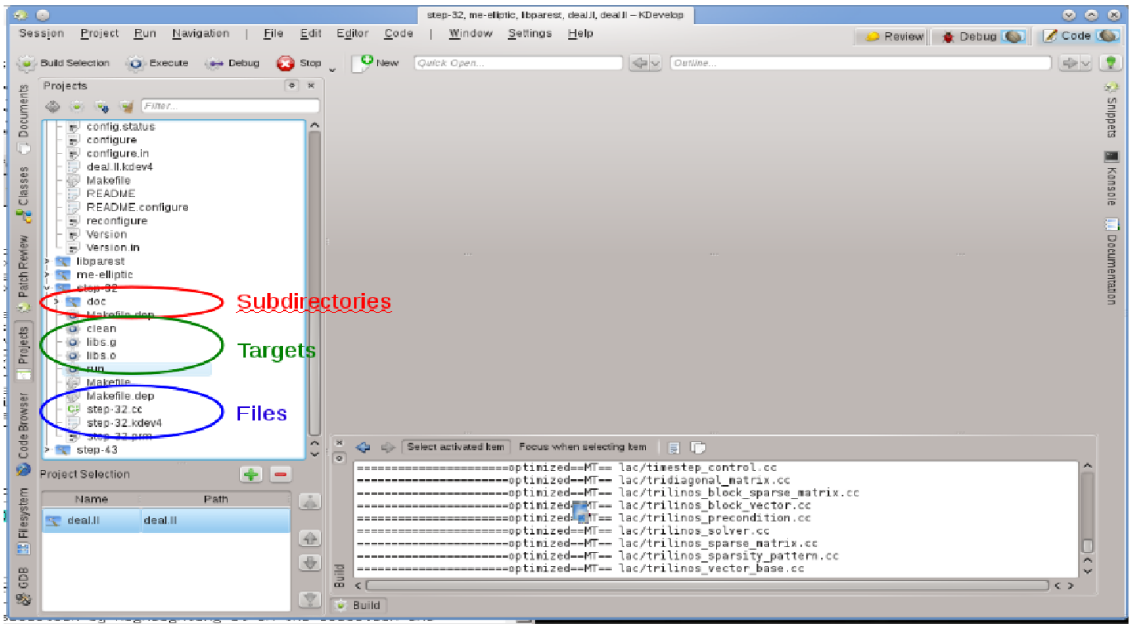
Many projects describe how source files have to be compiled (and which files have to be recompiled once a source or header file changes) using Makefiles that are interpreted by the **make** program (see, for example, GNU make). For simple projects, it is often very easy to set up such a file by hand. Larger projects often integrate their Makefiles with the **GNU autotools** (autoconf, autoheader, automake). In this section, let us simply assume that you have a Makefile for your project and you want to teach KDevelop how to interact with it.

> NOTE
> KDevelop 4.x doesn't know about the **GNU autotools** at the time this section is written. If your project uses them, you will have to run `./configure` or any of the other related commands by hand on a command line. If you want to do this within KDevelop, open the **Konsole** tool (if necessary add it to the perimeter of the main window using the menu **Windows** → **Add tool view**) that gives you a shell window view and run `./configure` from the command line in this view.

The first step is to teach KDevelop about targets in your Makefiles. There are two ways to do that: selecting individual Makefile targets, and choosing a set of targets you may want to build frequently. For both approaches, open the **Projects** tool by clicking on the **Projects** button on the perimeter of KDevelop's main window (if you don't have this button see above how to add a tool's button there). The **Projects** tool window has two parts: the top half — titled **Projects** — lists all of your projects and let's you expand the underlying directory trees. The bottom half — titled **Project Selection** — lists a subset of those projects that will be built if you choose the menu item **Project** → **Build selection** or hit **F8**; we'll come back to this part below.

5.1 Building individual Makefile targets

In the top part of the project view, expand the sub-tree for one project, let's say the one for which you want to run a particular Makefile target. This will give you icons for (i) directories under this project, (ii) files in the top-level directory for this project, (iii) Makefile targets KDevelop can identify. These categories are shown in the picture at right. Note that KDevelop *understands* Makefile syntax to a certain degree and therefore can offer you targets defined in this Makefile (though this understanding has its limits if targets are composed or implicit).



To build any of the targets listed there, click on it with the right mouse button and select **Build**. For example, doing this with the 'clean' target will simply execute 'make clean'. You can see this happening in the subwindow titled **Build** that opens up, showing the command and the output. (This window corresponds to the **Build** tool, so you can close and later re-open the window using the **Build** tool button on the perimeter of the main window. It is shown at the bottom right of the picture.)

5.2 Selecting a collection of Makefile targets for repeated building

Right-clicking on individual Makefile targets every time you want to build something will quickly get old. Rather, we'd like to have individual targets for one or more of the projects in the session that we can repeatedly build without much mouse work. This is where the concept of 'Build target selections' comes in: it is a collection of Makefile targets that are built one-after-the-other whenever you hit the **Build selection** button in the button list at the top, select the **Project** → **Build selection** menu item, or hit the **F8** function key.

The list of selected Makefile targets is shown in the bottom half of the **Projects** tool view.

By default, the selection contains all projects, but you can change that. For example, if your list of projects contains three projects (a base library L and two applications A and B), but you're currently only working on project A, then you may want to remove project B from the selection by highlighting it in the selection and hitting the button. Furthermore, you probably want to make sure that the library L is built before project A by moving entries in the selection up and down using the buttons to the right of the list. You can also get a particular Makefile target into the selection by right-clicking onto it and selecting **Add to buildset**, or just highlighting it and hitting the button just above the list of selected targets.

KDevelop allows you to configure what to do whenever you build the selection. To this end, use the menu item **Project** → **Open configuration**. There, you can for example select the number of simultaneous jobs 'make' should execute — if your computer has, say, 8 processor cores, then entering 8 in this field would be a useful choice. In this dialog, the **Default make target** is a Makefile target used for *all* targets in the selection.

5.3 What to do with error messages

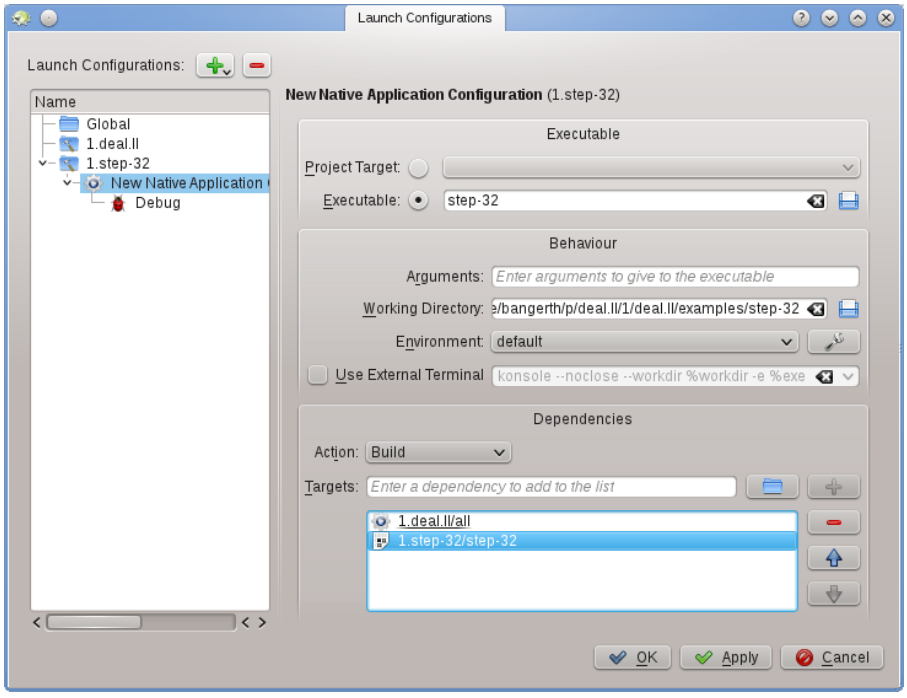
If the compiler encounters an error message, simply click on the line with the error message and the editor will jump to the line (and if available column) where the error was reported. Depending on the error message, KDevelop may also offer you several possible actions to fix the error, for example by declaring a previously undeclared variable if an unknown symbol was found.

Chapter 6

Running programs in KDevelop

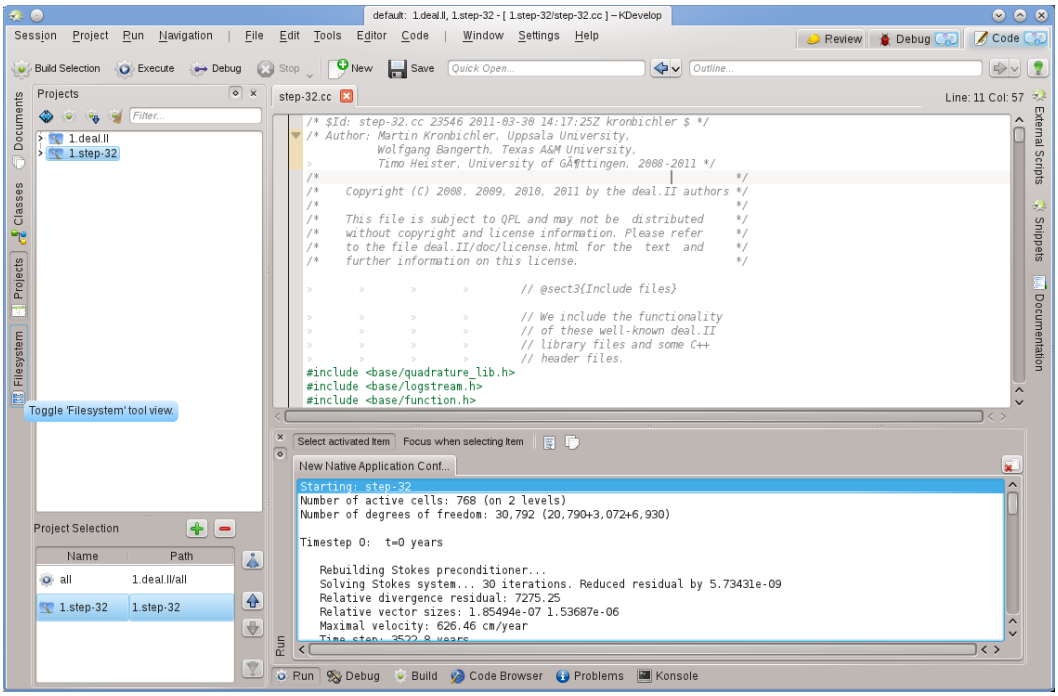
Once you have built a program, you will want to run it. To do this, need to configure *Launches* for your projects. A *Launch* consists of the name of an executable, a set of command line parameters, and an execution environment (such as 'run this program in a shell', or 'run this program in the debugger').

6.1 Setting up launches in KDevelop



To set this up go to menu item **Run** → **Configure launches**, highlight the project you want to add a launch for, and click on the button. Then enter the name of the executable, and the path where you want to run the program. If running the executable depends on building the executable and/or other libraries first, then you may want to add them to the list at the bottom: select **Build** from the dropdown menu, then hit the symbol to the right of the

textbox and select whatever target you want to have built. In the example above, I have selected the target **all** from project *1.deal.II* and *step-32* from project *1.step-32* to make sure both the base library and the application program have been compiled and are up to date before the program is actually executed. While you're there, you may as well also configure a debug launch by clicking on the **Debug** symbol and adding the name of the debugger program; if this is the system's default debugger (e.g. gdb on Linux®), then you don't need to do this step.



You can now try to run the program: Select **Run** → **Execute Launch** from KDevelop's main window menu (or hit **Shift-F9**) and your program should run in a separate subwindow of KDevelop. The picture above shows the result: The new **Run** tool subwindow at the bottom shows the output of the program that is being run, in this case of the *step-32* program.

> NOTE
> If you have configured multiple launches, you can choose which one should run when you hit **Shift-F9** by going to **Run** → **Current Launch Configuration**. There is a non-obvious way to edit the name of a configuration, however: in the dialog box you get when you select **Run** → **Current Launch Configuration**, double-click on the name of the configuration in the tree view on the left, which will allow you to edit the configuration's name.

6.2 Some useful keyboard shortcuts

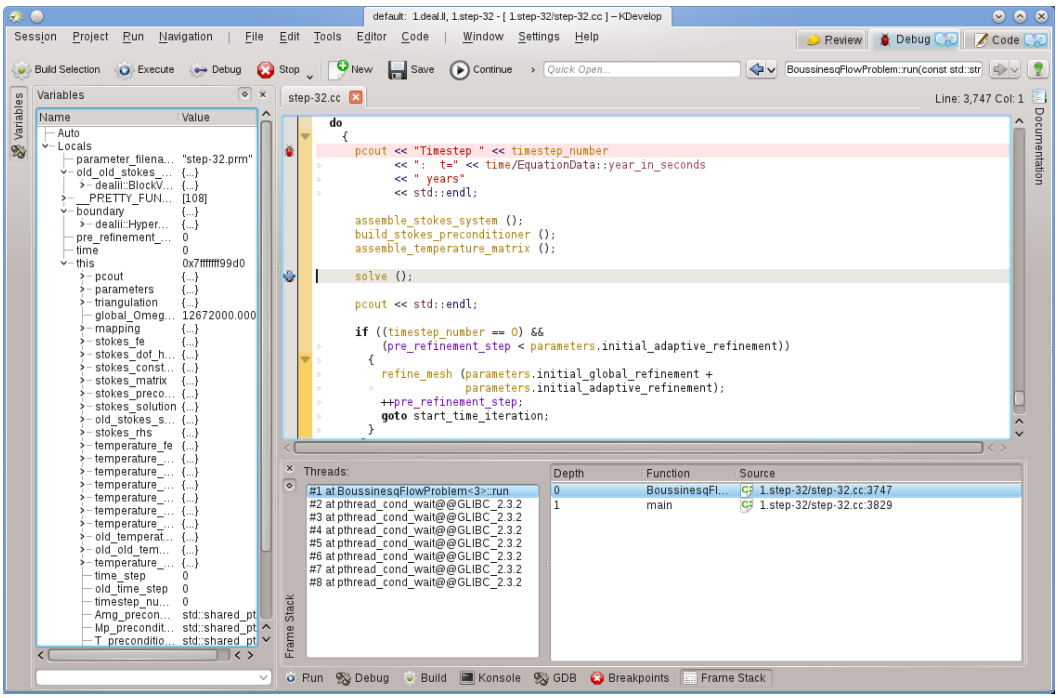
Running a program	
F8	Build (call make)
Shift-F9	Run
Alt-F9	Run program in the debugger; you may want to set breakpoints beforehand, for example by right-clicking with the mouse on a particular line in the source code

Chapter 7

Debugging programs in KDevelop

7.1 Running a program in the debugger

Once you have a launch configured (see Running programs), you can also run it in a debugger: Select the menu item **Run** → **Debug Launch**, or hit **Alt-F9**. If you are familiar with gdb, the effect is the same as starting gdb with the executable specified in the launch configuration and then saying `Run`. This means that if the program calls `abort()` somewhere (e.g. when you run onto a failing assertion) or if there is a segmentation fault, then the debugger will stop. On the other hand, if the program runs to the end (with or without doing the right thing) then the debugger will not stop by itself before the program is finished. In the latter case, you will want to set a breakpoint on all those lines of your code base where you want the debugger to stop before you run the debug launch. You can do that by moving the cursor on such a line and selecting the menu item **Run** → **Toggle breakpoint**, or right-clicking on a line and selecting **Toggle Breakpoint** from the context menu.



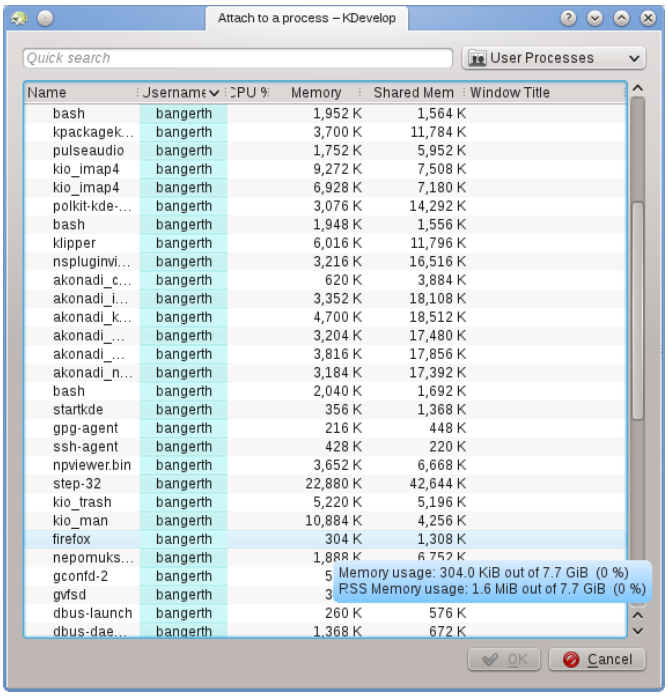
Running a program in the debugger will put KDevelop in a different mode: it will replace all the

'Tool' buttons on the perimeter of the main window by ones that are appropriate for debugging, rather than for editing. You can see which of the mode you are in by looking at the top right of the window: there are tabs named **Review**, **Debug**, and **Code**; clicking on them allows you to switch back and forth between the three modes; each mode has a set of tool views of its own, which you can configure in the same way as we configured the **Code** tools in the section Tools and views.

Once the debugger stops (at a breakpoint, or a point where `abort()` is called) you can inspect a variety of information about your program. For example, in the image above, we have selected the **Frame Stack** tool at the bottom (roughly equivalent to gdb's 'backtrace' and 'info threads' commands) that shows the various threads that are currently running in your program at the left (here a total of 8) and how execution got to the current stopping point at the right (here: `main()` called `run()`; the list would be longer had we stopped in a function called by `run()` itself). On the left, we can inspect local variables including the current object (the object pointed to by the `this` variable).

From here, there are various possibilities you can do: You can execute the current line (**F10**, gdb's 'next' command), step into the functions (**F11**, gdb's 'step' command), or run to the end of the function (**F12**, gdb's 'finish' command). At every stage, KDevelop updates the variables shown at the left to their current values. You can also hover the mouse over a symbol in your code, e.g. a variable; KDevelop will then show the current value of that symbol and offer to stop the program during execution the next time this variable's value changes. If you know gdb, you can also click on the **GDB** tool button at the bottom and have the possibility to enter gdb commands, for example in order to change the value of a variable (for which there doesn't currently seem to be another way).

7.2 Attaching the debugger to a running process



Sometimes, one wants to debug a program that's already running. One scenario for this is debugging parallel programs using MPI, or for debugging a long running background process. To do this, go to the menu entry **Run** → **Attach to Process**, which will open a window like the one

above. You will want to select the program that matches your currently open project in KDevelop - in my case that would be the step-32 program.

This list of programs can be confusing because it is often long as in the case shown here. You can make your life a bit easier by going to the dropdown box at the top right of the window. The default value is **User processes**, i.e. all programs that are run by any of the users currently logged into this machine (if this is your desktop or laptop, you're probably the only such user, apart from root and various service accounts); the list doesn't include processes run by the root user, however. You can limit the list by either choosing **Own processes**, removing all the programs run by other users. Or better still: Select **Programs only**, which removes a lot of processes that are formally running under your name but that you don't usually interact with, such as the window manager, background tasks and so on that are unlikely candidates for debugging.

Once you have selected a process, attaching to it will get you into KDevelop's debug mode, open all the usual debugger tool views and stop the program at the position where it happened to be when you attached to it. You may then want to set breakpoints, viewpoints, or whatever else is necessary and continue program execution by going to the menu item **Run** → **Continue**.

7.3 Some useful keyboard shortcuts

Debugging	
F10	Step over (gdb's 'next')
F11	Step into (gdb's 'step')
F12	Step out of (gdb's 'finish')

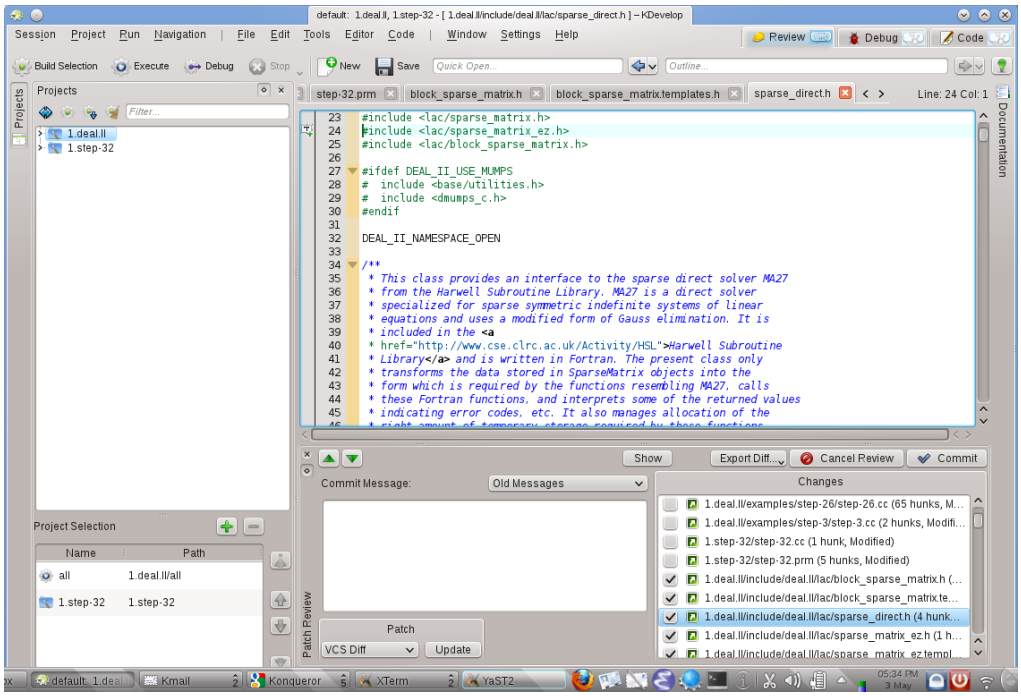
Chapter 8

Working with version control systems

If you are working with larger projects, chances are that the source code is managed by a version control system such as subversion or git. The following description is written with **subversion** in view but will be equally true if you use **git** or any other supported version control system.

First not that if the directory in which a project is located is under version control, KDevelop will automatically notice. In other words: It is not necessary that you tell KDevelop to check out a copy itself when setting up your project; it is ok to point KDevelop at a directory into which you have previously checked out a copy from the repository. If you have such a directory under version control, open the **Projects** tool view. There are then a number of things you can do:

- If your directory has become outdated, you can update it from the repository: Click on the project name with the right mouse button, go to the menu **Subversion** and select **Update**. This will bring all files that belong to this project up to date with respect to the repository.
- If you want to restrict this action to individual subdirectories or files, then expand the tree view of this project to the level you want and right click on a subdirectory or file name, then do the same as above.



- If you've edited one or more files, expand the view of this project to the directory in which these files are located and right click on the directory. This gives you a menu item **Subversion** that offers you different choices. Choose **Compare to base** to see the differences between the version you have edited and the version in the repository you had previously checked out (the revision 'base'). The resulting view will show the 'diffs' for all files in this directory.

- If you only edited a single file, you can also get the **Subversion** menu for this file by simply right clicking on the corresponding file name in the project view. Even simpler, just right clicking into the **Editor** view in which you have opened this file will also give you this menu option.

- If you want to check in one or more edited files, right click either on an individual file, subdirectory, or whole project and select **Subversion** → **Commit**. This will get you into **Review** mode, the third mode besides **Code** and **Debug** as you can see in the top right corner of the KDevelop main window. The picture on the right shows you how this looks. In **Review** mode, the top part shows you diffs for the entire subdirectory/project and each individual changed file with changes highlighted (see the various tabs on this part of the window). By default, all changed files are in the changeset you are about to commit, but you can unselect some of the files if their modifications are unrelated to what you want to commit. For example, in the example on the right I have unselected `step-32.cc` and `step-32.prm` because the changes in these files have nothing to do with the other ones I made to this project and I don't yet want to check them in yet (I may later want to do so in a separate commit). After reviewing the changes you can enter a commit message into the text box and hit **Commit** on the right to send things off.

- As with seeing differences, if you want to check in a single file you can also just right click into the editor window to get the **Subversion** → **Commit** menu item.

Chapter 9

Customizing KDevelop

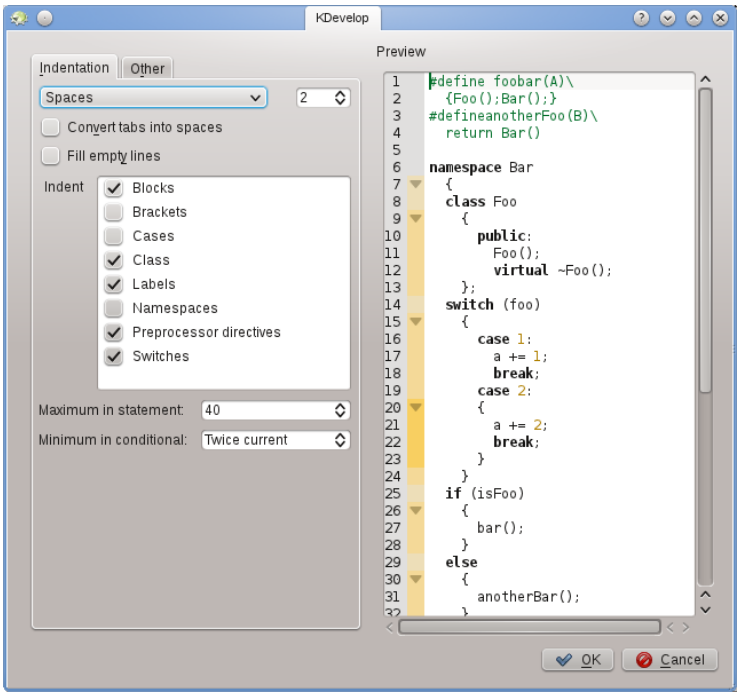
There are times when you want to change the default appearance or behavior of KDevelop, for example because you are used to different keyboard shortcuts or because your project requires a different indenting style for source code. In the following sections, we briefly discuss the various ways how KDevelop can be customized for these needs.

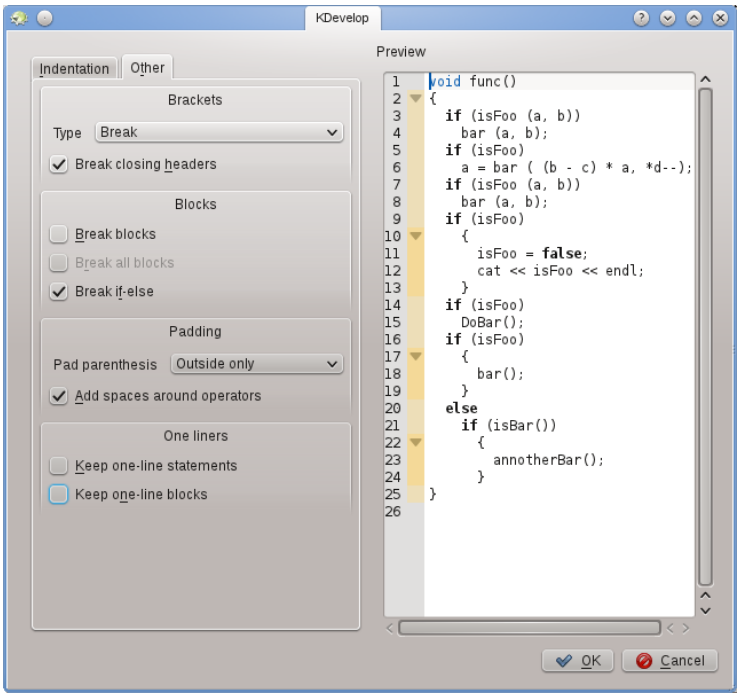
9.1 Customizing the editor

There are a number of useful things one can configure in and around KDevelop's built-in editor. Of more universal use is to switch on line numbering using the menu entry **Editor** → **View** → **Show line numbers**, making it easier to match compiler error messages or debug messages with locations in the code. In the same submenu you may also want to switch on the *Icon border* - a column to the left of your code in which KDevelop will show icons such as whether there is a breakpoint on the current line.

9.2 Customizing code indentation

Many of us like code formatted in a particular way. Many projects also enforce a particular indentation style. Neither may match KDevelop's default indentation style. However, this can be customized: Go to the **Settings** → **Customize KDevelop** menu item, then click on **Source Formatter** on the left. You can choose one of the predefined indentation styles that are widely in use, or define your own one by adding a new style and then editing it. There may not be a way to exactly re-create the style in which your project's sources have been indented in the past, but you can come close by using the settings for a new style; an example is shown in the two pictures below.


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

NOTE
With **KDevelop 4.2.2**, you can create a new style for a particular mimetype (e.g. for C++ header files) but this style does not show up among the list of possible styles for other mimetypes (e.g. for C++ source files) although it would of course be useful to use the same style for both kinds of files. You will therefore have to define the style twice, once for header and once for source files. This has been reported as KDevelop bug 272335.

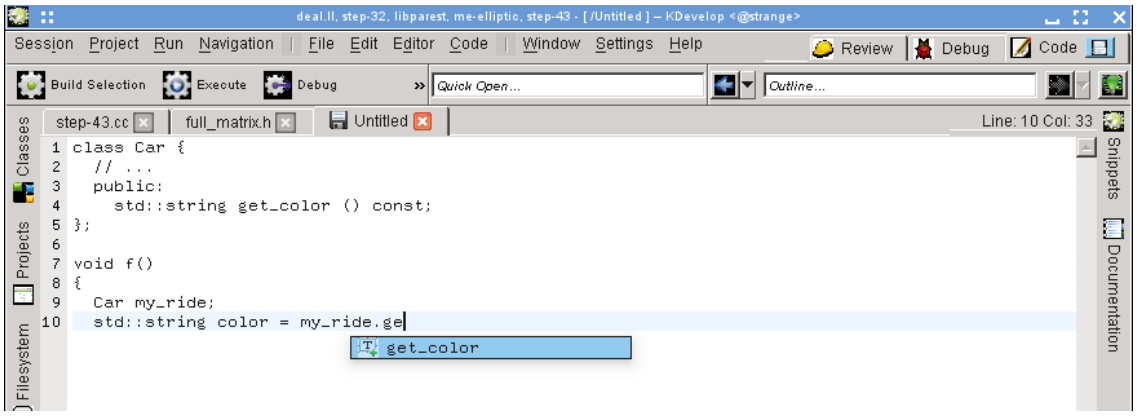
9.3 Customizing keyboard shortcuts

KDevelop has an almost boundless list of keyboard shortcuts (some of them are listed in the 'Useful keyboard shortcuts sections' of several chapters in this manual) that can be changed to your taste through the menu **Settings** → **Configure Shortcuts**. At the top of the dialog you can enter a searchword and it only shows those commands that match; you can then edit which key combination is bound to this command.

Two that have been found to be very useful to change are to set **Align** to the **Tab** key (many people don't usually enter tabs by hand and rather prefer if the editor chooses the layout of code; with the changed shortcut, hitting **Tab** makes KDevelop indent/outdent/align the code). The second one is putting **Toggle Breakpoint** on **Ctrl-B** since this is quite a frequent operation.

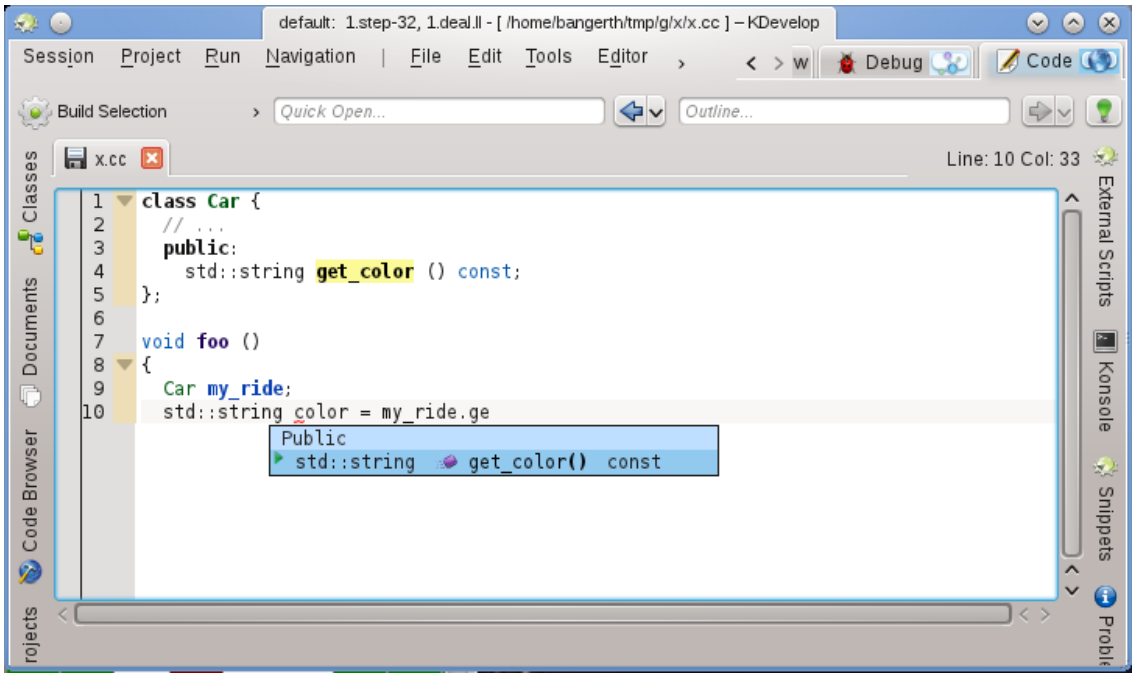
9.4 Customizing code auto-completion

Code completion is discussed in this manual's section on writing source code. In KDevelop, it comes from two sources: the editor, and the parse engine. The editor (Kate) is a component of the larger KDE environment and offers auto-completion based on words it has already seen in other parts of the same document. Such auto-completions can be identified in the tooltip by the icon that precedes it:



The editor's code completion can be customized via **Settings** → **Configure Editor** → **Editing** → **Auto Completion**. In particular, you can select how many characters you need to type in a word before auto-completion kicks in.

On the other hand, KDevelop's own auto-completion is much more powerful as it takes into account semantic information about the context. For example, it knows which member functions to offer when you type `object.`, etc., as shown here:



This context information comes from various language support plugins, which can be used after a given file has been saved (so it can check the filetype and use the correct language support).

KDevelop's completion is set to appear as you type, right away, pretty much everywhere that it could possibly complete something. This is configurable in **Settings** → **Configure KDevelop** → **Language Support**. If it isn't already set (as it should, by default), make sure **Enable Automatic Invocation** is set.

KDevelop has two ways to show a completion: **Minimal Automatic Completion** shows just the basic information in completion tooltips (i.e. the namespace, class, function, or variable name). This will look similar to Kate completion (except for the icons).

On the other hand, **Full completion** will additionally show the type for each entry, and in the case of functions, also the arguments they take. Also, if you are currently filling in the arguments to a function, full completion will have an additional info-box above the cursor that will show you the current argument you are working on.

KDevelop's code completion should also bring-to-top and highlight in green any completion items that match the currently expected type in both minimal and full completion, known as 'best-matches'.

The three possible choices for the completion level in the configuration dialog are:

- **Always minimal completion**: Never show 'Full Completion'
- **Minimal automatic completion**: Only show 'Full Completion' when auto-completion has been triggered manually (i.e., whenever you hit **Ctrl-Space**)
- **Always full completion**: Always show 'Full Completion'

Chapter 10

Credits and License