Creating Project Files

Project files contain all the information required by qmake to build your application, library, or plugin. Generally, you use a series of declarations to specify the resources in the project, but support for simple programming constructs enables you to describe different build processes for different platforms and environments.

Project File Elements

The project file format used by qmake can be used to support both simple and fairly complex build systems. Simple project files use a straightforward declarative style, defining standard variables to indicate the source and header files that are used in the project. Complex projects may use control flow structures to fine-tune the build process.

The following sections describe the different types of elements used in project files.

Variables

In a project file, variables are used to hold lists of strings. In the simplest projects, these variables inform qmake about the configuration options to use, or supply filenames and paths to use in the build process.

qmake looks for certain variables in each project file, and it uses the contents of these to determine what it should write to a Makefile. For example, the lists of values in the HEADERS and SOURCES variables are used to tell qmake about header and source files in the same directory as the project file.

Variables can also be used internally to store temporary lists of values, and existing lists of values can be overwritten or extended with new values.

The following snippet illustrates how lists of values are assigned to variables:

```
HEADERS = mainwindow.h paintwidget.h
```

The list of values in a variable is extended in the following way:

```
SOURCES = main.cpp mainwindow.cpp \
          paintwidget.cpp
CONFIG += console
```

Note: The first assignment only includes values that are specified on the same line as the HEADERS variable.

The CONFIG variable is another special variable that qmake uses when generating a Makefile. It is discussed in General Configuration. In the snippet above, `console` is added to the list of existing values contained in `CONFIG`.

The following table lists some frequently used variables and describes their contents. For a full list of variables and their descriptions, see Variables.

Variable	Contents
CONFIG	General project configuration options.
DESTDIR	The directory in which the executable or binary file will be placed.
FORMS	A list of UI files to be processed by the user interface compiler (uic).
HEADERS	A list of filenames of header (.h) files used when building the project.
QT	A list of Qt modules used in the project.
RESOURCES	A list of resource (.qrc) files to be included in the final project. See the The Qt Resource System for more information about these files.
SOURCES	A list of source code files to be used when building the project.
TEMPLATE	The template to use for the project. This determines whether the output of the build process will be an application, a library, or a plugin.

The contents of a variable can be read by prepending the variable name with `$$`. This can be used to assign the contents of one variable to another:

```
TEMP_SOURCES = $$SOURCES
```

The `$$` operator is used extensively with built-in functions that operate on strings and lists of values. For more information, see qmake Language.

Whitespace

Usually, whitespace separates values in variable assignments. To specify values that contain spaces, you must enclose the values in double quotes:

```
DEST = "Program Files"
```

The quoted text is treated as a single item in the list of values held by the variable. A similar approach is used to deal with paths that contain spaces, particularly when defining the INCLUDEPATH and LIBS variables for the Windows platform:

```
win32:INCLUDEPATH += "C:/mylibs/extra headers"
unix:INCLUDEPATH += "/home/user/extra headers"
```

Comments

```
# Comments usually start at the beginning of a line, but they
# can also follow other content on the same line.
```

To include the # character in variable assignments, it is necessary to use the contents of the built-in LITERAL_HASH variable.

Built-in Functions and Control Flow

qmake provides a number of built-in functions to enable the contents of variables to be processed. The most commonly used function in simple project files is the include() function which takes a filename as an argument. The contents of the given file are included in the project file at the place where the `include` function is used. The `include` function is most commonly used to include other project files:

```
include(other.pro)
```

Support for conditional structures is made available via scopes that behave like `if` statements in programming languages:

```
win32 {
    SOURCES += paintwidget_win.cpp
}
```

The assignments inside the braces are only made if the condition is true. In this case, the `win32` CONFIG option must be set. This happens automatically on Windows. The opening brace must stand on the same line as the condition.

More complex operations on variables that would usually require loops are provided by built-in functions such as find(), unique(), and count(). These functions, and many others are provided to manipulate strings and paths, support user input, and call external tools. For more information about using the functions, see qmake Language. For lists of all functions and their descriptions, see Replace Functions and Test Functions.

Project Templates

The TEMPLATE variable is used to define the type of project that will be built. If this is not declared in the project file, qmake assumes that an application should be built, and will generate an appropriate Makefile (or equivalent file) for the purpose.

The following table summarizes the types of projects available and describes the files that qmake will generate for each of them:

Template	qmake Output
app (default)	Makefile to build an application.
Template	qmake Output

	because your project is written in an interpreted language. **Note:** This template type is only available for Makefile-based generators. In particular, it will not work with the vcxproj and Xcode generators.
subdirs	Makefile containing rules for the subdirectories specified using the SUBDIRS variable. Each subdirectory must contain its own project file.
vcapp	Visual Studio Project file to build an application.
vclib	Visual Studio Project file to build a library.
vcsubdirs	Visual Studio Solution file to build projects in sub-directories.

See Building Common Project Types for advice on writing project files for projects that use the `app` and `lib` templates.

When the `subdirs` template is used, qmake generates a Makefile to examine each specified subdirectory, process any project file it finds there, and run the platform's `make` tool on the newly-created Makefile. The `SUBDIRS` variable is used to contain a list of all the subdirectories to be processed.

General Configuration

The CONFIG variable specifies the options and features that the project should be configured with.

The project can be built in *release* mode or *debug* mode, or both. If debug and release are both specified, the last one takes effect. If you specify the `debug_and_release` option to build both the debug and release versions of a project, the Makefile that qmake generates includes a rule that builds both versions. This can be invoked in the following way:

```
make all
```

Adding the `build_all` option to the `CONFIG` variable makes this rule the default when building the project.

Note: Each of the options specified in the `CONFIG` variable can also be used as a scope condition. You can test for the presence of certain configuration options by using the built-in CONFIG() function. For example, the following lines show the function as the condition in a scope to test whether only the `opengl` option is in use:

```
CONFIG(opengl) {
    message(Building with OpenGL support.)
} else {
    message(OpenGL support is not available.)
}
```

This enables different configurations to be defined for `release` and `debug` builds. For more information, see

Note: Some of these options only take effect when used on the relevant platform.

Option	Description
qt	The project is a Qt application and should link against the Qt library. You can use the QT variable to control any additional Qt modules that are required by your application. This value is added by default, but you can remove it to use qmake for a non-Qt project.
x11	The project is an X11 application or library. This value is not needed if the target uses Qt.

The application and library project templates provide you with more specialized configuration options to fine tune the build process. The options are explained in detail in Building Common Project Types.

For example, if your application uses the Qt library and you want to build it in debug mode, your project file will contain the following line:

```
CONFIG += qt debug
```

Note: You must use "+=", not "=", or qmake will not be able to use Qt's configuration to determine the settings needed for your project.

Declaring Qt Libraries

If the CONFIG variable contains the `qt` value, qmake's support for Qt applications is enabled. This makes it possible to fine-tune which of the Qt modules are used by your application. This is achieved with the QT variable which can be used to declare the required extension modules. For example, we can enable the XML and network modules in the following way:

```
QT += network xml
```

Note: QT includes the `core` and `gui` modules by default, so the above declaration *adds* the network and XML modules to this default list. The following assignment *omits* the default modules, and will lead to errors when the application's source code is being compiled:

```
QT = network xml # This will omit the core and gui modules.
```

If you want to build a project *without* the `gui` module, you need to exclude it with the "-=" operator. By default, QT contains both `core` and `gui`, so the following line will result in a minimal Qt project being built:

For a list of Qt modules that you can add to the QT variable, see QT.

Configuration Features

qmake can be set up with extra configuration features that are specified in feature (.prf) files. These extra features often provide support for custom tools that are used during the build process. To add a feature to the build process, append the feature name (the stem of the feature filename) to the CONFIG variable.

For example, qmake can configure the build process to take advantage of external libraries that are supported by pkg-config, such as the D-Bus and ogg libraries, with the following lines:

```
CONFIG += link_pkgconfig
PKGCONFIG += ogg dbus-1
```

For more information about adding features, see Adding New Configuration Features.

Declaring Other Libraries

If you are using other libraries in your project in addition to those supplied with Qt, you need to specify them in your project file.

The paths that qmake searches for libraries and the specific libraries to link against can be added to the list of values in the LIBS variable. You can specify the paths to the libraries or use the Unix-style notation for specifying libraries and paths.

For example, the following lines show how a library can be specified:

```
LIBS += -L/usr/local/lib -lmath
```

The paths containing header files can also be specified in a similar way using the INCLUDEPATH variable.

For example, to add several paths to be searched for header files:

```
INCLUDEPATH = c:/msdev/include d:/stl/include
```

‹ Getting Started with qmake

Building Common Project Types ›

Qt DOCUMENTATION

The Qt Company

Contact Us

Company

About Us
Investors
Newsroom
Careers
Office Locations

Licensing

Terms & Conditions
Open Source
FAQ

Support

Support Services
Professional Services
Partners
Training

For Customers

Support Center
Downloads
Qt Login
Contact Us
Customer Success

Community

Contribute to Qt
Forum
Wiki
Downloads
Marketplace

Feedback Sign In