qmake Language

Many qmake project files simply describe the sources and header files used by the project, using a list of `name = value` and `name += value` definitions. qmake also provides other operators, functions, and scopes that can be used to process the information supplied in variable declarations. These advanced features allow Makefiles to be generated for multiple platforms from a single project file.

Operators

In many project files, the assignment (=) and append (+=) operators can be used to include all the information about a project. The typical pattern of use is to assign a list of values to a variable, and append more values depending on the result of various tests. Since qmake defines certain variables using default values, it is sometimes necessary to use the removal (-=) operator to filter out values that are not required. The following sections describe how to use operators to manipulate the contents of variables.

Assigning Values

The = operator assigns a value to a variable:

```
TARGET = myapp
```

The above line sets the TARGET variable to `myapp`. This will overwrite any values previously set for `TARGET` with `myapp`.

Appending Values

The += operator appends a new value to the list of values in a variable:

```
DEFINES += USE_MY_STUFF
```

The above line appends `USE_MY_STUFF` to the list of pre-processor defines to be put in the generated Makefile.

Removing Values

The `-=` operator removes a value from the list of values in a variable:

The above line removes USE_MY_STUFF from the list of pre-processor defines to be put in the generated Makefile.

Adding Unique Values

The *= operator adds a value to the list of values in a variable, but only if it is not already present. This prevents values from being included many times in a variable. For example:

```
DEFINES *= USE_MY_STUFF
```

In the above line, USE_MY_STUFF will only be added to the list of pre-processor defines if it is not already defined. Note that the unique() function can also be used to ensure that a variable only contains one instance of each value.

Replacing Values

The ~= operator replaces any values that match a regular expression with the specified value:

```
DEFINES ~= s/QT_[DT].+/QT
```

In the above line, any values in the list that start with QT_D or QT_T are replaced with QT.

Variable Expansion

The $$ operator is used to extract the contents of a variable, and can be used to pass values between variables or supply them to functions:

```
EVERYTHING = $$SOURCES $$HEADERS
message("The project contains the following files:")
message($$EVERYTHING)
```

Variables can be used to store the contents of environment variables. These can be evaluated at the time when qmake is run, or included in the generated Makefile for evaluation when the project is built.

To obtain the contents of an environment value when qmake is run, use the $$(...) operator:

```
DESTDIR = $$(PWD)
message(The project will be installed in $$DESTDIR)
```

In the above assignment, the value of the PWD environment variable is read when the project file is processed.

To obtain the contents of an environment value at the time when the generated Makefile is processed, use the $(...) operator:

```
DESTDIR = $$(PWD)
message(The project will be installed in $$DESTDIR)

DESTDIR = $(PWD)
message(The project will be installed in the value of PWD)
message(when the Makefile is processed.)
```

In the above assignment, the value of PWD is read immediately when the project file is processed, but `$(PWD)` is assigned to DESTDIR in the generated Makefile. This makes the build process more flexible as long as the environment variable is set correctly when the Makefile is processed.

Accessing qmake Properties

The special `$$[...]` operator can be used to access qmake properties:

```
message(Qt version: $$[QT_VERSION])
message(Qt is installed in $$[QT_INSTALL_PREFIX])
message(Qt resources can be found in the following locations:)
message(Documentation: $$[QT_INSTALL_DOCS])
message(Header files: $$[QT_INSTALL_HEADERS])
message(Libraries: $$[QT_INSTALL_LIBS])
message(Binary files (executables): $$[QT_INSTALL_BINS])
message(Plugins: $$[QT_INSTALL_PLUGINS])
message(Data files: $$[QT_INSTALL_DATA])
message(Translation files: $$[QT_INSTALL_TRANSLATIONS])
message(Settings: $$[QT_INSTALL_CONFIGURATION])
message(Examples: $$[QT_INSTALL_EXAMPLES])
```

For more information, see Configuring qmake.

The properties accessible with this operator are typically used to enable third party plugins and components to be integrated in Qt. For example, a *Qt Designer* plugin can be installed alongside *Qt Designer*'s built-in plugins if the following declaration is made in its project file:

```
target.path = $$[QT_INSTALL_PLUGINS]/designer
INSTALLS += target
```

Scopes

Scopes are similar to `if` statements in procedural programming languages. If a certain condition is true, the declarations inside the scope are processed.

Scope Syntax

Scopes consist of a condition followed by an opening brace on the same line, a sequence of commands and definitions, and a closing brace on a new line:

```
    ...
}
```

The opening brace *must be written on the same line as the condition*. Scopes may be concatenated to include more than one condition, as described in the following sections.

Scopes and Conditions

A scope is written as a condition followed by a series of declarations contained within a pair of braces. For example:

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

The above code will add the `paintwidget_win.cpp` file to the sources listed in the generated Makefile when building for a Windows platform. When building for other platforms, the define will be ignored.

The conditions used in a given scope can also be negated to provide an alternative set of declarations that will be processed only if the original condition is false. For example, to process something when building for all platforms *except* Windows, negate the scope like this:

```
!win32 {
    SOURCES -= paintwidget_win.cpp
}
```

Scopes can be nested to combine more than one condition. For instance, to include a particular file for a certain platform only if debugging is enabled, write the following:

```
macx {
    CONFIG(debug, debug|release) {
        HEADERS += debugging.h
    }
}
```

To save writing many nested scopes, you can nest scopes using the : operator. The nested scopes in the above example can be rewritten in the following way:

```
macx:CONFIG(debug, debug|release) {
    HEADERS += debugging.h
}
```

```
win32:DEFINES += USE_MY_STUFF
```

The above line adds USE_MY_STUFF to the DEFINES variable only when building for the Windows platform. Generally, the : operator behaves like a logical AND operator, joining together a number of conditions, and requiring all of them to be true.

There is also the | operator to act like a logical OR operator, joining together a number of conditions, and requiring only one of them to be true.

```
win32|macx {
    HEADERS += debugging.h
}
```

If you need to mix both operators, you can use the `if` function to specify operator precedence.

```
if(win32|macos):CONFIG(debug, debug|release) {
    # Do something on Windows and macOS,
    # but only for the debug configuration.
}
win32|if(macos:CONFIG(debug, debug|release)) {
    # Do something on Windows (regardless of debug or release)
    # and on macOS (only for debug).
}
```

The condition accepts the wildcard character to match a family of CONFIG values or mkspec names.

```
win32-* {
    # Matches every mkspec starting with "win32-"
    SOURCES += win32_specific.cpp
}
```

Note: Historically, checking the mkspec name with wildcards like above was qmake's way to check for the platform. Nowadays, we recommend to use values that are defined by the mkspec in the QMAKE_PLATFORM variable.

You can also provide alternative declarations to those within a scope by using an `else` scope. Each `else` scope is processed if the conditions for the preceding scopes are false. This allows you to write complex tests when combined with other scopes (separated by the : operator as above). For example:

```
win32:xml {
    message(Building for Windows)
    SOURCES += xmlhandler_win.cpp
} else:xml {
```

```
    message( "Unknown configuration" )
}
```

Configuration and Scopes

The values stored in the CONFIG variable are treated specially by qmake. Each of the possible values can be used as the condition for a scope. For example, the list of values held by `CONFIG` can be extended with the `opengl` value:

```
CONFIG += opengl
```

As a result of this operation, any scopes that test for `opengl` will be processed. We can use this feature to give the final executable an appropriate name:

```
opengl {
    TARGET = application-gl
} else {
    TARGET = application
}
```

This feature makes it easy to change the configuration for a project without losing all the custom settings that might be needed for a specific configuration. In the above code, the declarations in the first scope are processed, and the final executable will be called `application-gl`. However, if `opengl` is not specified, the declarations in the second scope are processed instead, and the final executable will be called `application`.

Since it is possible to put your own values on the `CONFIG` line, this provides you with a convenient way to customize project files and fine-tune the generated Makefiles.

Platform Scope Values

In addition to the `win32`, `macx`, and `unix` values used in many scope conditions, various other built-in platform and compiler-specific values can be tested with scopes. These are based on platform specifications provided in Qt's `mkspecs` directory. For example, the following lines from a project file show the current specification in use and test for the `linux-g++` specification:

```
message($$QMAKESPEC)

linux-g++ {
    message(Linux)
}
```

You can test for any other platform-compiler combination as long as a specification exists for it in the `mkspecs` directory.

Variables

variables with a given name when it encounters an assignment to that name. For example:

```
MY_VARIABLE = value
```

There are no restricitions on what you do to your own variables, as qmake will ignore them unless it needs to evaluate them when processing a scope.

You can also assign the value of a current variable to another variable by prefixing $$ to the variable name. For example:

```
MY_DEFINES = $$DEFINES
```

Now the MY_DEFINES variable contains what is in the DEFINES variable at this point in the project file. This is also equivalent to:

```
MY_DEFINES = $${DEFINES}
```

The second notation allows you to append the contents of the variable to another value without separating the two with a space. For example, the following will ensure that the final executable will be given a name that includes the project template being used:

```
TARGET = myproject_$${TEMPLATE}
```

Replace Functions

qmake provides a selection of built-in functions to allow the contents of variables to be processed. These functions process the arguments supplied to them and return a value, or list of values, as a result. To assign a result to a variable, use the $$ operator with this type of function as you would to assign contents of one variable to another:

```
HEADERS = model.h
HEADERS += $$OTHER_HEADERS
HEADERS = $$unique(HEADERS)
```

This type of function should be used on the right-hand side of assignments (that is, as an operand).

You can define your own functions for processing the contents of variables as follows:

```
defineReplace(functionName){
    #function code
```

The following example function takes a variable name as its only argument, extracts a list of values from the variable with the eval() built-in function, and compiles a list of files:

```
defineReplace(headersAndSources) {
    variable = $$1
    names = $$eval($$variable)
    headers =
    sources =

    for(name, names) {
        header = $${name}.h
        exists($$header) {
            headers += $$header
        }
        source = $${name}.cpp
        exists($$source) {
            sources += $$source
        }
    }
    return($$headers $$sources)
}
```

Test Functions

qmake provides built-in functions that can be used as conditions when writing scopes. These functions do not return a value, but instead indicate *success* or *failure*:

```
count(options, 2) {
    message(Both release and debug specified.)
}
```

This type of function should be used in conditional expressions only.

It is possible to define your own functions to provide conditions for scopes. The following example tests whether each file in a list exists and returns true if they all exist, or false if not:

```
defineTest(allFiles) {
    files = $$ARGS

    for(file, files) {
        !exists($$file) {
            return(false)
        }
    }
    return(true)
}
```