Qt创建者编码规则

注意： 本文档正在进行中。

编码规则旨在指导Qt Creator开发人员，帮助他们编写可理解和可维护的代码，并最大限度地减少混乱和意外。

像往常一样，规则不是一成不变的。如果你有一个很好的理由打破一个，那就这样做。但首先要确保至少有一些其他开发人员同意你的观点。

要为Qt主创建者来源做出贡献，您应该遵守以下规则：

- 最重要的规则是：KISS（保持简短和简单）。始终选择更简单的实现选项，而不是更复杂的实现选项。这使得维护变得更加容易。
- 写出好C++代码。也就是说，可读性强，必要时可以很好地注释，并且是面向对象的。
- 利用Qt的优势。不要重新发明轮子。想想你的代码的哪些部分足够通用，以至于它们可能会被合并到Qt而不是Qt创建器中。
- 使代码适应Qt创建者中的现有结构。如果您有改进的想法，请在编写代码之前与其他开发人员讨论。
- 遵循代码构造、格式设置以及模式和实践中的准则。
- 文档界面。现在我们使用qdoc，但正在考虑更改为多氧。

提交代码

要向Qt Creator提交代码，您必须了解工具和机制以及Qt开发背后的理念。有关如何设置用于处理Qt Creator的开发环境以及如何提交代码和文档以进行包含的更多信息，请参阅Qt贡献指南。

二进制和源兼容性

以下列表描述了如何对版本进行编号，并定义了*版本之间的二进制兼容性*和*源代码兼容性*：

- Qt Creator 3.0.0 是*主要版本*，Qt Creator 3.1.0 是*次要版本*，Qt Creator 3.1.3 是*补丁版本*。
- *向后二进制兼容性*意味着链接到早期版本的库的代码仍然有效。
- *前向二进制兼容性*意味着链接到较新版本库的代码可与较旧的库配合使用。
- *源代码兼容性*意味着代码无需修改即可编译。

我们目前不保证主要版本和次要版本之间的二进制或源代码兼容性。

但是，我们尝试为同一次要版本中的修补程序版本保留向后二进制兼容性和向后源代码兼容性：

- 软 API 冻结：从次要版本的 Beta 发布后不久开始，我们开始在该次要版本中保持向后源代码兼容性，包括其补丁版本。这意味着从那时起，使用Qt Creator API的代码将针对此次要版本的所有即将推出的版本（包括其补丁版本）的API进行编译。这条规则可能偶尔也会有例外，应适当传达。

分。

- 硬 ABI 冻结：从次要版本的最终版本开始，我们保留此版本及其所有补丁版本的向后源代码和二进制兼容性。

为了保持向后兼容性：

- 不要添加或删除任何公共 API（例如全局函数、公共/受保护/私有成员函数）。
- 不要重新实现函数（甚至不要内联，也不要执行受保护或私有函数）。
- 检查二进制兼容性解决方法，了解保持二进制兼容性的方法。

有关二进制兼容性的详细信息，请参阅C++的二进制兼容性问题。

从插件元数据的角度来看，这意味着：

- 补丁版本中的Qt Creator插件将次要版本设置为 。例如，版本3.1.2中的插件将具有.compatVersioncompatVersion="3.1.0"
- 次要版本的预发布版本（包括候选版本）仍将以自身作为 ，这意味着针对最终版本编译的插件将不会在预发布版本中加载。compatVersion
- Qt Creator插件开发人员可以通过在声明对其他插件的依赖关系时设置相应的补丁来决定他们的插件是否需要其他Qt Creator插件的某个补丁版本（或更高版本），或者他们是否可以使用这个次要版本的所有补丁版本。Qt项目提供的Qt创建者插件的默认值是需要最新的补丁版本。version

例如，Qt Creator 3.1 beta（内部版本号3.0.82）中的插件将具有Find

```
<plugin name="Find" version="3.0.82" compatVersion="3.0.82">
  <dependencyList>
    <dependency name="Core" version="3.0.82"/>
    ....
```

Qt Creator 3.1.0最终版中的插件将具有Find

```
<plugin name="Find" version="3.1.0" compatVersion="3.1.0">
  <dependencyList>
    <dependency name="Core" version="3.1.0"/>
    ....
```

The plugin in Qt Creator 3.1.1 patch release will have version 3.1.1, will be backward binary compatible with plugin version 3.1.0 (), and will require a plugin that is binary backward compatible with plugin version 3.1.1:FindFindcompatVersion="3.1.0"CoreCore

```
<plugin name="Find" version="3.1.1" compatVersion="3.1.0">
  <dependencyList>
    <dependency name="Core" version="3.1.1"/>
    ....
```

Code Constructs

Follow the guidelines for code constructs to make the code faster and clearer. In addition, the guidelines allow you to take advantage of the strong type checking in C++.

```
++T;
--U;

-NOT-

T++;
U--;
```

- Try to minimize evaluation of the same code over and over. This is aimed especially at loops:

```
Container::iterator end = large.end();
for (Container::iterator it = large.begin(); it != end; ++it) {
        ...;
}

-NOT-

for (Container::iterator it = large.begin();
     it != large.end(); ++it) {
        ...;
}
```

Formatting

Capitalizing Identifiers

Use camel case in identifiers.

Capitalize the first word in an identifier as follows:

- Class names begin with a capital letter.
- Function names begin with a lower case letter.
- Variable names begin with a lower case letter.
- Enum names and values begin with a capital letter. Unscoped Enum values contain some part of the name of the enum type.

Whitespace

- Use four spaces for indentation, no tabs.
- Use blank lines to group statements together where suited.
- Always use only one blank line.

Pointers and References

For pointers or references, always use a single space before an asterisk (*) or an ampersand (&), but never after. Avoid C-style casts when possible:

```
char *blockOfMemory = (char *)malloc(data.size());
```

```
char* blockOfMemory = (char* ) malloc(data.size());
```

Of course, in this particulare case, using might be an even better option.new

Operator Names and Parentheses

Do not use spaces between operator names and parentheses. The equation marks (==) are a part of the operator name, and therefore, spaces make the declaration look like an expression:

```
operator==(type)

-NOT-

operator == (type)
```

Function Names and Parentheses

Do not use spaces between function names and parentheses:

```
void mangle()

-NOT-

void mangle ()
```

Keywords

Always use a single space after a keyword, and before a curly brace:

```
if (foo) {
}

-NOT-

if(foo){
}
```

Comments

In general, put one space after "//". To align text in multiline comments, you can insert multiple spaces.

Braces

As a base rule, place the left curly brace on the same line as the start of the statement:

```
if (codec) {
}
```

```
if (codec)
{
}
```

Exception: Function implementations and class declarations always have the left brace in the beginning of a line:

```
static void foo(int g)
{
    qDebug("foo: %i", g);
}

class Moo
{
};
```

Use curly braces when the body of a conditional statement contains more than one line, and also if a single line statement is somewhat complex. Otherwise, omit them:

```
if (address.isEmpty())
    return false;

for (int i = 0; i < 10; ++i)
    qDebug("%i", i);

-NOT-

if (address.isEmpty()) {
    return false;
}

for (int i = 0; i < 10; ++i) {
    qDebug("%i", i);
}
```

Exception 1: Use braces also if the parent statement covers several lines or if it wraps:

```
if (address.isEmpty()
        || !isValid()
        || !codec) {
    return false;
}
```

Note: This could be re-written as:

```
if (address.isEmpty())
    return false;
```

```
if (!codec)
    return false;
```

Exception 2: Use braces also in if-then-else blocks where either the if-code or the else-code covers several lines:

```
if (address.isEmpty()) {
    --it;
} else {
    qDebug("%s", qPrintable(address));
    ++it;
}

-NOT-

if (address.isEmpty())
    --it;
else {
    qDebug("%s", qPrintable(address));
    ++it;
}
```

```
if (a) {
    if (b)
        ...
    else
        ...
}

-NOT-

if (a)
    if (b)
        ...
    else
        ...
```

Use curly braces when the body of a conditional statement is empty:

```
while (a) {}

-NOT-

while (a);
```

Parentheses

Use parentheses to group expressions:

```
-NOT-

if (a && b || c)
```

```
(a + b) & c

-NOT-

a + b & c
```

Line Breaks

- Keep lines shorter than 100 characters.
- Insert line breaks if necessary.
- Commas go at the end of a broken line.
- Operators start at the beginning of the new line.

```
if (longExpression
    || otherLongExpression
    || otherOtherLongExpression) {
}

-NOT-

if (longExpression ||
    otherLongExpression ||
    otherOtherLongExpression) {
}
```

Declarations

- Use this order for the access sections of your class: public, protected, private. The public section is interesting for every user of the class. The private section is only of interest for the implementors of the class (you).
- Avoid declaring global objects in the declaration file of the class. If the same variable is used for all objects, use a static member.
- Use instead of . Some compilers mangle that difference into the symbol names and spit out warnings if a struct declaration is followed by a class definition. To avoid ongoing changes from one to the other we declare the preferred way.`classstructclass`

Declaring Variables

- Avoid global variables of class type to rule out initialization order problems. Consider using if they cannot be avoided.`Q_GLOBAL_STATIC`
- Declare global string literals as

```
const char aString[] = "Hello";
```

counters and temporaries, where the purpose of the variable is obvious.

- Declare each variable on a separate line:

```
QString a = "Joe";
QString b = "Foo";

-NOT-

QString a = "Joe", b = "Foo";
```

Note: formally calls a copy constructor on a temporary that is constructed from a string literal. Therefore, it is potentially more expensive than direct construction by . However, the compiler is allowed to elide the copy (even if this has side effects), and modern compilers typically do so. Given these equal costs, Qt Creator code favours the '=' idiom as it is in line with the traditional C-style initialization, it cannot be mistaken as function declaration, and it reduces the level of nested parantheses in more initializations.`QString a = "Joe"QString a("Joe")`

- Avoid abbreviations:

```
int height;
int width;
char *nameOfThis;
char *nameOfThat;

-NOT-

int a, b;
char *c, *d;
```

- Wait with declaring a variable until it is needed. This is especially important when initialization is done at the same time.

Namespaces

- Put the left curly brace on the same line as the keyword.`namespace`
- Do not indent declarations or definitions inside.
- Optional, but recommended if the namespaces spans more than a few lines: Add a comment after the right curly brace repeating the namespace.

```
namespace MyPlugin {

void someFunction() { ... }

    }  // namespace MyPlugin
```

- As an exception, if there is only a single class declaration inside the namespace, all can go on a single line:

```
namespace MyPlugin { class MyClass; }
```

region.

- Do not rely on using-directives when accessing global functions.
- In other cases, you are encouraged to use using-directives, as they help you avoid cluttering the code. Prefer putting all using-directives near the top of the file, after all includes.

```
[in foo.cpp]
...
#include "foos.h"
...
#include <utils/filename.h>
...
using namespace Utils;

namespace Foo {
namespace Internal {

void SomeThing::bar()
{
    FilePath f;              // or Utils::FilePath f
    ...
}
...
} // namespace Internal      // or only // Internal
} // namespace Foo           // or only // Foo

-NOT-

[in foo.h]
...
using namespace Utils;       // Wrong: no using-directives in headers

class SomeThing
{
    ...
};

-NOT-

[in foo.cpp]
...
using namespace Utils;

#include "bar.h"             // Wrong: #include after using-directive

-NOT-

[in foo.cpp]
...
using namepace Foo;

void SomeThing::bar()        // Wrong if Something is in namespace Foo
{
    ...
}
```

Namespacing

Read Qt In Namespace and keep in mind that all of Qt Creator is *namespace aware* code.

The namespacing policy within Qt Creator is as follows:

- Classes/Symbols of a library or plugin that are exported for use of other libraries or plugins are in a namespace specific to that library/plugin, e.g. .`MyPlugin`
- Classes/Symbols of a library or plugin that are not exported are in an additional namespace, e.g. .`InternalMyPlugin::Internal`

Passing File Names

Qt Creator API expects file names in portable format, that is, with slashes (/) instead of backslashes (\) even on Windows. To pass a file name from the user to the API, convert it with QDir::fromNativeSeparators first. To present a file name to the user, convert it back to native format with QDir::toNativeSeparators. Consider using Utils::FilePath::fromUserInput(QString) and Utils::FilePath::toUserOutput() for these tasks.

Use Utils::FilePath when comparing file names, because that takes case sensitivity into account. Also make sure that you compare clean paths (QDir::cleanPath()).

Classes to Use and Classes Not to Use

A significant portion of Qt Creator code handles data on devices that are not the same as the development machine. These may differ in aspects like path separator, line endings, process launching details and so on.

However, some basic Qt classes assume that a Qt application is only concerned with machines that are similar to the development machine.

These classes are therefore not appropriate to use in the part of Qt Creator that is concerned with non-local code. Instead, Qt Creator's Utils library provides substitutes, leading to the following rules:

- Use Utils::FilePath for any QString that semantically is a file or directory, see also Passing File Names.
- Prefer using Utils::FilePath over any use of QDir and QFileInfo.
- Prefer using Utils::QtcProcess over QProcess.
- If Utils::FilePath or Utils::QtcProcess functionality is not sufficient for your purpose, prefer enhancing them over falling back to QString or QProcess.
- Avoid platform #ifdefs unless they are absolutely needed for locally executed code, and even then prefer Utils::HostInfo over #ifdefs.

Plugin Extension Points

A plugin extension point is an interface that is provided by one plugin to be implemented by others. The plugin then retrieves all implementations of the interface and uses them. That is, they *extend* the functionality of the plugin. Typically, the implementations of the interface are put into the global object pool during plugin initialization, and the plugin retrieves them from the object pool at the end of plugin initialization.

For example, the Find plugin provides the FindFilter interface for other plugins to implement. With the FindFilter interface, additional search scopes can be added, that appear in the **Advanced Search** dialog. The Find plugin retrieves all FindFilter implementations from the global object pool and presents them in the dialog. The plugin forwards the actual search request to the correct FindFilter implementation, which then performs the search.

Using the Global Object Pool

You can add objects to the global object pool via ExtensionSystem::PluginManager::addObject(), and retrieve objects of a specific type again via ExtensionSystem::PluginManager::getObject(). This should mostly be used for implementations of Plugin Extension Points.

C++ Features

- Prefer over header guards.`#pragma once`
- Do not use exceptions, unless you know what you do.
- Do not use RTTI (Run-Time Type Information; that is, the typeinfo struct, the dynamic_cast or the typeid operators, including throwing exceptions), unless you know what you do.
- Do not use virtual inheritance, unless you know what you do.
- Use templates wisely, not just because you can.

  Hint: Use the compile autotest to see whether a C++ feature is supported by all compilers in the test farm.
- All code is ASCII only (7-bit characters only, run if unsure)`man ascii`
  - Rationale: We have too many locales inhouse and an unhealthy mix of UTF-8 and Latin1 systems. Usually, characters > 127 can be broken without you even knowing by clicking Save in your favourite editor.
  - For strings: Use \nnn (where nnn is the octal representation of whatever locale you want your string in) or \xnn (where nn is hexadecimal). For example: QString s = QString::fromUtf8("\213\005");
  - For umlauts in documentation, or other non-ASCII characters, either use the qdoc command or use the relevant macro. For example: for ü.`\unicode\uuml`
- Use static keywords instead of anonymous namespaces whenever possible. A name localized to the compilation unit with static is guaranteed to have internal linkage. For names declared in anonymous namespaces, the C++ standard unfortunately mandates external linkage (ISO/IEC 14882, 7.1.1/6, or see various discussions about this on the gcc mailing lists).

Null Pointers

Use nullptr for null pointer constants.

```
void *p = nullptr;

-NOT-

void *p = NULL;

-NOT-

void *p = '\0';

-NOT-

void *p = 42 - 7 * 6;
```

Note: As an exception, imported third party code as well as code interfacing the native APIs (src/support/os_*) can use NULL or 0.

C++11 and C++14 Features

Code should compile with Microsoft Visual Studio 2013, g++ 4.7, and Clang 3.1.

Lambdas

note that this is a C++14 feature and you might need to enable C++14 support in your compiler.

```
[]() {
    Foo *foo = activeFoo();
    return foo ? foo->displayName() : QString();
});
```

- If you use static functions from the class that the lambda is located in, you have to explicitly capture . Otherwise it does not compile with g++ 4.7 and earlier.`this`

```
void Foo::something()
{
    ...
    [this]() { Foo::someStaticFunction(); }
    ...
}

-NOT-

void Foo::something()
{
    ...
    []() { Foo::someStaticFunction(); }
    ...
}
```

Format the lambda according to the following rules:

- Place the capture-list, parameter list, return type, and opening brace on the first line, the body indented on the following lines, and the closing brace on a new line.

```
[]() -> bool {
    something();
    return isSomethingElse();
}

-NOT-

[]() -> bool { something();
somethingElse(); }
```

- Place a closing parenthesis and semicolon of an enclosing function call on the same line as the closing brace of the lambda.

```
foo([]() {
    something();
});
```

- If you are using a lambda in an 'if' statement, start the lambda on a new line, to avoid confusion between the opening

```
if (anyOf(fooList,
        [](Foo foo) {
            return foo.isGreat();
        }) {
    return;
}

-NOT-

if (anyOf(fooList, [](Foo foo) {
            return foo.isGreat();
        }) {
    return;
}
```

- Optionally, place the lambda completely on one line if it fits.

```
foo([] { return true; });

if (foo([] { return true; })) {
    ...
}
```

auto Keyword

Optionally, you can use the keyword in the following cases. If in doubt, for example if using could make the code less readable, do not use . Keep in mind that code is read much more often than written.autoautoauto

- When it avoids repetition of a type in the same statement.

```
auto something = new MyCustomType;
auto keyEvent = static_cast<QKeyEvent *>(event);
auto myList = QStringList({ "FooThing",  "BarThing" });
```

- When assigning iterator types.

```
auto it = myList.const_iterator();
```

Scoped enums

You can use scoped enums in places where the implicit conversion to int of unscoped enums is undesired or the additional scope is useful.

Delegating constructors

Use delegating constructors if multiple constructors use essentially the same code.

Initializer list

Use initializer lists to initialize containers, for example:

Initialization with Curly Brackets

If you use initialization with curly brackets, follow the same rules as with round brackets. For example:

```
class Values // the following code is quite useful for test fixtures
{
    float floatValue = 4; // prefer that for simple types
    QVector<int> values = {1, 2, 3, 4, integerValue}; // prefer that syntax for initializer lists
    SomeValues someValues{"One", 2, 3.4}; // not an initializer_list
    SomeValues &someValuesReference = someValues;
    ComplexType complexType{values, otherValues} // constructor call
}

object.setEntry({"SectionA", value, doubleValue}); // calls a constructor
object.setEntry({}); // calls default constructor
```

But be careful not to overuse it, to not obfuscate your code.

Non-Static Data Member Initialization

Use non-static data member initialization for trivial initializations, except in public exported classes.

Defaulted and Deleted Functions

Consider using and to control the special functions.`=default=delete`

Override

It is recommended to use the keyword when overriding virtual functions. Do not use virtual on overridden functions.`override`

Make sure that a class uses consistently, either for all overridden functions or for none.`override`

Nullptr

All compilers support , but there is no consensus on using it. If in doubt, ask the maintainer of the module whether they prefer using .`nullptrnullptr`

Range-Based for-Loop

You may use range-based for-loops, but beware of the spurious detachment problem. If the for-loop only reads the container and it is not obvious whether the container is const or unshared, use to ensure that the container is not unnecessarily detached.`std::cref()`

Using QObject

- Remember to add the Q_OBJECT macro to QObject subclasses that rely on the meta object system. Meta object system related features are the definition of signals and slots, the use of , and others. See also Casting.`qobject_cast<>`
- Prefer Qt5-style calls over Qt4-style.`connect()`
- When using Qt4-style calls, normalize the arguments for signals and slots inside connect statements to safely make signal and slot lookup a few cycles faster. You can use $QTDIR/util/normalize to normalize existing code. For more information, see QMetaObject::normalizedSignature.`connect()`

Qt DOCUMENTATION

Creator.

Including Headers

- Use the following format to include Qt headers: . Do not include the module as it might have changed between Qt4 and Qt5.`#include <QWhatEver>`
- Arrange includes in an order that goes from specific to generic to ensure that the headers are self-contained. For example:
  - `#include "myclass.h"`
  - `#include "otherclassinplugin.h"`
  - `#include <otherplugin/someclass.h>`
  - `#include <QtClass>`
  - `#include <stdthing>`
  - `#include <system.h>`
- Enclose headers from other plugins in angle brackets (<>) rather than quotation marks ("") to make it easier to spot external dependencies in the sources.
- Add empty lines between long blocks of *peer* headers and try to arrange the headers in alphabetic order within a block.

Casting

- Avoid C casts, prefer C++ casts (, ,) Both and C-style casts are dangerous, but at least will not remove the const modifier.`static_castconst_castreinterpret_castreinterpret_castreinterpret_cast`
- Do not use , use for QObjects, or refactor your design, for example by introducing a function (see QListWidgetItem), unless you know what you do.`dynamic_castqobject_casttype()`

Compiler and Platform-specific Issues

- Be extremely careful when using the question mark operator. If the returned types are not identical, some compilers generate code that crashes at runtime (you will not even get a compiler warning):

```
QString s;
// crash at runtime - QString vs. const char *
return condition ? s : "nothing";
```

- Be extremely careful about alignment.

  Whenever a pointer is cast such that the required alignment of the target is increased, the resulting code might crash at runtime on some architectures. For example, if a is cast to a , it will crash on machines where integers have to be aligned at two-byte or four-byte boundaries.`const char *const int *`

  Use a union to force the compiler to align variables correctly. In the example below, you can be sure that all instances of AlignHelper are aligned at integer-boundaries:

```
union AlignHelper
{
    char c;
    int i;
};
```

- Anything that has a constructor or needs to run code to be initialized cannot be used as global object in library code, since it is undefined when that constructor or code will be run (on first usage, on library load, before or not at all).`main()`

  Even if the execution time of the initializer is defined for shared libraries, you will get into trouble when moving that code in a plugin or if the library is compiled statically:

```
// global scope

-NOT-

// Default constructor needs to be run to initialize x:
static const QString x;

-NOT-

// Constructor that takes a const char * has to be run:
static const QString y = "Hello";

-NOT-

QString z;

-NOT-

// Call time of foo() undefined, might not be called at all:
static const int i = foo();
```

  Things you can do:

```
// global scope
// No constructor must be run, x set at compile time:
static const char x[] = "someText";

// y will be set at compile time:
static int y = 7;

// Will be initialized statically, no code being run.
static MyStruct s = {1, 2, 3};

// Pointers to objects are OK, no code needed to be run to
// initialize ptr:
static QString *ptr = 0;

// Use Q_GLOBAL_STATIC to create static global objects instead:

Q_GLOBAL_STATIC(QString, s)

void foo()
{
    s()->append("moo");
}
```

  Note: Static objects in function scope are no problem. The constructor will be run the first time the function is

- A is signed or unsigned dependent on the architecture. Use signed or if you explicitly want a signed or unsigned char. The following code will break on PowerPC, for example:`charcharuchar`

```
// Condition is always true on platforms where the
// default is unsigned:
if (c >= 0) {
    ...
}
```

- Avoid 64-bit enum values. The AAPCS (Procedure Call Standard for the ARM Architecture) embedded ABI hard codes all enum values to a 32-bit integer.
- Do not mix const and non-const iterators. This will silently crash on broken compilers.

```
for (Container::const_iterator it = c.constBegin(); it != c.constEnd(); ++it)

-NOT-

for (Container::const_iterator it = c.begin(); it != c.end(); ++it)
```

- Do not inline virtual destructors in exported classes. This leads to duplicated vtables in dependent plugins and this can also break RTTI. See QTBUG-45582.

Esthetics

- Prefer unscoped enums to define const over static const int or defines. Enumeration values will be replaced by the compiler at compile time, resulting in faster code. Defines are not namespace safe.
- Prefer verbose argument names in headers. Qt Creator will show the argument names in their completion box. It will look better in the documentation.

Inheriting from Template or Tool Classes

Inheriting from template or tool classes has the following potential pitfalls:

- The destructors are not virtual, which can lead to memory leaks.
- The symbols are not exported (and mostly inline), which can lead to symbol clashes.

For example, library A has class and library B has class . Suddenly, QList symbols are exported from two libraries which results in a clash.`Q_EXPORT X: public QList<QVariant> {};Q_EXPORT Y: public QList<QVariant> {};`

Inheritance Versus Aggregation

- Use inheritance if there is a clear *is-a* relation.
- Use aggregation for re-use of orthogonal building blocks.
- Prefer aggregation over inheritance if there is a choice.

Conventions for Public Header Files

Our public header files have to survive the strict settings of some of our users. All installed headers have to follow these rules:

- No C style casts (). Use , or , for basic types, use the constructor form: instead of . For more information, see Casting.`-Wold-style-caststatic_castconst_castreinterpret_castint(a)(int)a` equalqFuzzyCompareqIsNull

- Do not hide virtual functions in subclasses ({-Woverloaded-virtual}). If the baseclass A has a virtual and subclass B an overload with the same name, , the A function is hidden. Use the keyword to make it visible again, and add the following silly workaround for broken compilers:`int val()int val(int x)valusing`

```
class B: public A
{
#ifdef Q_NO_USING_KEYWORD
inline int val() { return A::val(); }
#else
using A::val;
#endif
};
```

- Do not shadow variables ().`-Wshadow`
- Avoid things like if possible.`this->x = x;`
- Do not give variables the same name as functions declared in your class.
- To improve code readability, always check whether a preprocessor variable is defined before probing its value ().`-Wundef`

```
#if defined(Foo) && Foo == 0

  -NOT-

#if Foo == 0

-NOT-

#if Foo - 0 == 0
```

- When checking for a preprocessor define using the operator, always include the variable name in parentheses.`defined`

```
#if defined(Foo)

  -NOT-

  #if defined Foo
```

Class Member Names

We use the "m_" prefix convention, except for public struct members (typically in *Private classes and the very rare cases of really public structures). The and pointers are exempt from the "m_" rule.dq

The pointers ("Pimpls") are named "d", not "m_d". The type of the pointer in is , where is declared in the same namespace as , or if is exported, in the corresponding {Internal} namespace.`ddclass FooFooPrivate *FooPrivateFooFoo`

If needed (for example when the private object needs to emit signals of the proper class), can be a friend of .`FooPrivateFoo`

If the private class needs a backreference to the real class, the pointer is named , and its type is . (Same convention as in Qt: "q" looks like an inverted "d".)`qFoo *`

```
############### bar.h

#include <QScopedPointer>
//#include <memory>

struct BarPrivate;

struct Bar
{
    Bar();
    ~Bar();
    int value() const;

    QScopedPointer<BarPrivate> d;
    //std::unique_ptr<BarPrivate> d;
};

############### bar.cpp

#include "bar.h"

struct BarPrivate { BarPrivate() : i(23) {} int i; };

Bar::Bar() : d(new BarPrivate) {}

Bar::~Bar() {}

int Bar::value() const { return d->i; }

############### baruser.cpp

#include "bar.h"

int barUser() { Bar b; return b.value(); }

############### baz.h

struct BazPrivate;

struct Baz
{
    Baz();
    ~Baz();
    int value() const;

    BazPrivate *d;
};

############### baz.cpp

#include "baz.h"

struct BazPrivate { BazPrivate() : i(23) {} int i; };

Baz::Baz() : d(new BazPrivate) {}

Baz::~Baz() { delete d; }
```

```
#include "baz.h"

int bazUser() { Baz b; return b.value(); }

############### main.cpp

int barUser();
int bazUser();

int main() { return barUser() + bazUser(); }
```

Results:

```
Object file size:

 14428 bar.o
  4744 baz.o

  8508 baruser.o
  2952 bazuser.o

Symbols in bar.o:

    00000000 W _ZN3Foo10BarPrivateC1Ev
    00000036 T _ZN3Foo3BarC1Ev
    00000000 T _ZN3Foo3BarC2Ev
    00000080 T _ZN3Foo3BarD1Ev
    0000006c T _ZN3Foo3BarD2Ev
    00000000 W _ZN14QScopedPointerIN3Foo10BarPrivateENS_21QScopedPointerDeleterIS2_EEEC1EPS2_
    00000000 W _ZN14QScopedPointerIN3Foo10BarPrivateENS_21QScopedPointerDeleterIS2_EEED1Ev
    00000000 W _ZN21QScopedPointerDeleterIN3Foo10BarPrivateEE7cleanupEPS2_
    00000000 W _ZN7qt_noopEv
             U _ZN9qt_assertEPKcS1_i
    00000094 T _ZNK3Foo3Bar5valueEv
    00000000 W _ZNK14QScopedPointerIN3Foo10BarPrivateENS_21QScopedPointerDeleterIS2_EEEptEv
             U _ZdlPv
             U _Znwj
             U __gxx_personality_v0

Symbols in baz.o:

    00000000 W _ZN3Foo10BazPrivateC1Ev
    0000002c T _ZN3Foo3BazC1Ev
    00000000 T _ZN3Foo3BazC2Ev
    0000006e T _ZN3Foo3BazD1Ev
    00000058 T _ZN3Foo3BazD2Ev
    00000084 T _ZNK3Foo3Baz5valueEv
             U _ZdlPv
             U _Znwj
             U __gxx_personality_v0
```

Documentation

In the .cpp files, you can document the implementation if the implementation is not obvious.

< User Interface Text Guidelines Qt Creator API Reference >

Qt DOCUMENTATION