Uwyn C++ Coding Standard

Geert Bevin
Uwyn

Avenue de Scailmont 34,
7170 Manage,
Belgium,
gbevin@uwyn.com

Uwyn C++ Coding Standard
by Geert Bevin

Published 2002

Table of Contents

List of Tables

Introduction

This document explains the C++ coding style we've adopted and lists a number of best practices that should be followed when contributing to our C++ projects.

I. Make Names Fit

Names are the heart of programming. In the past people believed knowing someone's true name gave them magical power over that person. If you can think up the true name for something, you give yourself and the people coming after power over the code. Don't laugh!

Chapter 1. Selecting names

A name is the result of a long deep thought process about the ecology it lives in. Only a programmer who understands the system as a whole can create a name what "fits" with the system. If the name is appropriate everything fits together naturally, relationships are clear, meaning is derivable, and reasoning from common human expectations works as expected.

If you find all your names could be `Thing` and `DoIt` then you should probably revisit your design.

Class Names

- Name the class after what it is. If you can't think of what it is that is a clue you have not thought through the design well enough.
- Compound names of over three words are a clue your design may be confusing various entities in your system. Revisit your design. Try a CRC card session to see if your objects have more responsibilities than they should.
- Avoid the temptation of bringing the name of the class a class derives from into the derived class's name. A class should stand on its own. It doesn't matter what it derives from.
- Suffixes are sometimes helpful. For example, if your system uses agents then naming something DownloadAgent conveys real information.

Method and Function Names

- Usually every method and function performs an action, so the name should make clear what it does: `CheckForErrors()` instead of `ErrorCheck()`, `DumpDataToFile()` instead of `DataFile()`. This will also make functions and data objects more distinguishable.

  Classes are often nouns. By making function names verbs and following other naming conventions programs can be read more naturally.
- Suffixes are sometimes useful:
  - `Max` - to mean the maximum value something can have.
  - `Cnt` - the current count of a running count variable.
  - `Key` - key value.

  For example: `RetryMax` to mean the maximum number of retries, `RetryCnt` to mean the current retry count.
- Prefixes are sometimes useful:
  - `Is` - to ask a question about something. Whenever someone sees Is they will know it's a question.
  - `Get` - get a value.
  - `Set` - set a value.

  For example: `IsHitRetryLimit`.

Variable Names

Make every variable name descriptive, limit the use of abbreviations or letter-words. It's worth writing words completely since it makes the code much more readable. Beware however that when trying to find a good name, you don't

end up with with something like 'the_variable_for_the_loop', use a proper English word for it like 'counter' or 'iterator'. English is a rich language and trying to find a correctly fitting word is important for code brevety, cleanness and variation. Whenever in doubt, just use an thesaurus like Merriam-Webster (http://www.m-w.com) or a rhyming dictionary like Rhyme (http://rhyme.sourceforge.net/).

Exceptions

Some standard variables are used for often recurring tasks. Below is a list of those that are accepted :

- `i` : integer counter
- `it` : STL-like iterator
- `<type>_it` : STL-like iterator of a certain type for differentiation amongst types
- `tmp_<type>` : eg. tmp_qstring, tmp_int, tmp_float for variables that are solely used for the storage of temporary intermediate values

No All Upper Case Abbreviations

When confronted with a situation where you could use an all upper case abbreviation instead use an initial upper case letter followed by all lower case letters. No matter what.

Justification

People seem to have very different intuitions when making names containing abbreviations. It's best to settle on one strategy so the names are absolutely predictable.

Take for example NetworkABCKey. Notice how the C from ABC and K from key are confused. Some people don't mind this and others just hate it so you'll find different policies in different code so you never know what to call something.

Chapter 2. Naming scheme

A standard naming scheme is important to ensure that all code looks similar and that every developer can understand new code immediately without have to grasp a new naming scheme first.

One of the main aspects of this naming scheme is that all names should contain key information about the type of language construct is refers to. Additionally, certain prefixes will be used to prevent common error in the use of basic c++ concepts such as pointers, references and scope. This however doesn't involve into a full-blown and difficult to understand and maintain Hongarian notation.

Class Names

- Use upper case letters as word separators, lower case for the rest of a word
- First character in a name is upper case
- No underbars ('_')

Justification

- Standard naming scheme in very clean OO languages such as Java and C#.
- Stands out the best amongst the other names formatting, since a class is the basic element in c++ this is a great benefit.

Example 2-1. Class Names Example

```
class NameOneTwo
class Name
```

Class Files

- Each class definition should be in its own file where each file is named directly after the class's name.
- Source files have the .cpp extension and header files have the .h extension.
- In general each class should be implemented in one source file. A common exception to this rule are inner classes that provide class specific functionality such as thread execution. Another common exception are very closely related classes such as a collection class and its iterator.
- If the source file gets too large or you want to avoid compiling templates all the time then add additional files, where the section is lower case and seperated of the classname through an underscore.

Justification

- Using exactly the same name as the real class makes it easy to establish the relation.
- Not implementing several classes in one source file makes it very easy to find a class implementation when looking for it.

Example 2-2. Class Files Example

```
ClassName.h
ClassName.cpp
ClassName_section1.cpp
ClassName_section2.cpp
```

Method Names

- Use upper case letters as word separators, lower case for the rest of a word
- First character in a name is lower case
- No underbars ('_')

Justification

- Differentiates the first word part, which is often a verb. This makes it very clear what a method does.
- Not exactly similar to class names and thus makes Class.doSomething() much more readable as Class.DoSomething(), cleanly indicating through case which is which.

Example 2-3. Method Names Example

```
class NameOneTwo
{
public:
    int     doIt();
    void    handleError();
}
```

Class Member Names

- Member names should be prepended with the character 'm'.
- Member the 'm' use the same rules as for class names.
- 'm' always precedes other name modifiers like 'p' for pointer.

Justification

- Prepending 'm' prevents any conflict with method names. Often your methods and attribute names will be similar, especially for accessors.

Example 2-4. Class Member Names Example

```
class NameOneTwo
{
public:
    int     varAbc();
    int     errorNumber();
private:
    int     mVarAbc;
```

```
    int     mErrorNumber;
    String  *mpName;
}
```

Method Argument Names

- The first character should be lower case.
- All word beginnings after the first letter should be upper case as with class names.

Justification

- You can always tell which variables are passed in variables.
- You can use names similar to class names without conflicting with class names.

Example 2-5. Method Argument Names Example

```
class NameOneTwo
{
public:
    int startYourEngines(Engine& rSomeEngine, bool autoRestart);
}
```

Variable Names on the Stack

- Use all lower case letters
- Use '_' as the word separator.

Justification

- With this approach the scope of the variable is clear in the code.
- Now all variables look different and are identifiable in the code.

Example 2-6. Variable Names on the Stack Example

```
int NameOneTwo::handleError(int errorNumber)
{
    int             error = OsErr();
    Time            time_of_error;
    ErrorProcessor  error_processor;
    String          *p_tmpstring;
}
```

Pointer Variables

- Pointers should be prepended with 'p'.
- Place the * close to the variable name not the pointer type.

Justification

- The idea is that the difference between a pointer, object, and a reference to an object is important for understanding the code, especially in C++ where -> can be overloaded, and casting and copy semantics are important.
- The * belongs near the variable name since its not cumulative to other declarations of the same line. This convention follows closely the way c++ behaves in this matter.

Example 2-7. Placement of * during pointer declarations

```
BAD:  int* p_var;
GOOD: int *p_var;
```

Example 2-8. Pointer Variables Example

```
QString *p_name = new QString;
QString *p_name; // note, only pName is a pointer.
QString name;
QString address;
```

Reference Variables and Functions Returning References

- References should be prepended with 'r'.
- Place the & close to the variable name not the reference type when working with reference variables. With function declarations, the & should be close to the reference type.
- Functions returning non const references should be prepended with 'r' to make it clear that a modifiable is being returned.

Justification

- The difference between variable types is clarified.
- It establishes the difference between a method returning a modifiable object and the same method name returning a non-modifiable object.

Example 2-9. Reference Variables and Functions Returning References Example

```
class Test
{
public:
    void                doSomething(StatusInfo &rStatus);
    StatusInfo&         rStatus();
    const StatusInfo&   status() const;
```

```
private:
    StatusInfo &mrStatus;
}
```

Global Variables

- Global variables should be prepended with 'g'.

Justification

- It's important to know the scope of a variable.

Example 2-10. Global Variables Example

```
Logger  gLog;
Logger *gpLog;
```

Global Constants

- Global constants should be all caps with '_' separators.

Justification

- It's tradition for global constants to named this way. You must be careful to not conflict with other global #defines and enum labels.

Example 2-11. Global Constants Example

```
const int A_GLOBAL_CONSTANT = 5;
```

Static Variables

- Static variables should be prepended with 's'.

Justification

- It's important to know the scope of a variable.

Example 2-12. Static Variables Example

```
class Test
{
public:
private:
    static StatusInfo   msStatus;
}
```

Type Names

- When possible for types based on native types make a typedef.
- Typedef names should use the same naming policy as for a class with the word Type appended.

Justification

- Types are things so should use upper case letters. Type is appended to make it clear this is not a class.

Example 2-13. Type Names Example

```
typedef uint16  ModuleType;
typedef uint32  SystemType;
```

Enum Names

- Labels All Upper Case with '_' Word Separators

Justification

- This is the standard rule for enum labels.

Example 2-14. Enum Names Example

```
enum PinStateType
{
    PIN_OFF,
    PIN_ON,
};
```

Note

It's often useful to be able to say an enum is not in any of its valid states. Make a label for an uninitialized or error state. Make it the first label if possible.

Example 2-15. Invalid Enum States Example

```
enum { STATE_NONE, STATE_OPEN, STATE_RUNNING, STATE_DYING};
```

C Function Names

- In a C++ project there should be very few C functions.
- For C functions use the GNU convention of all lower case letters with '_' as the word delimiter.

Justification

- It makes C functions very different from any C++ related names.

Example 2-16. C Function Names Example

```
int some_c_function()
{
}
```

#define and Macro Names

- Put #defines and macros in all upper using '_' separators.

Justification

- This makes it very clear that the value is not alterable and in the case of macros, makes it clear that you are using a construct that requires care.
- Some subtle errors can occur when macro names and enum labels use the same name.

Example 2-17. #define and Macro Names Example

```
#define MAX(a,b) blah
#define IS_ERR(err) blah
```

II. Code Formatting

This uniform code formatting style encreases readability, maintainability and the comprehension of the code's logic.

Chapter 3. Braces and paranthesis

Braces Policy

Place braces under and inline with keywords, like this :

Example 3-1. Braces Policy Example

```
if (condition)          while (condition)
{                       {
    ...                     ...
}                       }
```

Justification

- If you use an editor (such as vi) that supports brace matching, this is a much better style than the default unix style where braces aren't vertically aligned. Why? Let's say you have a large block of code and want to know where the block ends. You move to the first brace hit a key and the editor finds the matching brace.

  Example 3-2. Braces Policy Justification

  ```
  if (very_long_condition && second_very_long_condition)
  {
      ...
  }
  else if (...)
  {
      ...
  }
  ```

  To move from block to block you just need to use cursor down and your brace matching key. No need to move to the end of the line to match a brace then jerk back and forth.

Braces Usage

All if, while and do statements must either have braces or be on a single line.

Always Uses Braces Form, even if there is only a single statement within the braces.

Justification

- Easier to read, you just have to scan for one form.
- Uniform idiom for scop blocks since they are all encolsed in braces.
- It provides a more consistent look.
- This doesn't affect execution speed and it's easy to apply.
- It ensures that when someone adds a line of code later there are already braces and they don't forget.

Example 3-3. Brace Usage Example

```
if (1 == somevalue)
{
   somevalue = 2;
}
```

Parenthesis Policy

- Do not put parens next to keywords. Put a space between.
- Do put parens next to function names.
- Do not use parens in return statements when it's not necessary.

Justification

- Keywords are not functions. By putting parens next to keywords, keywords and function names adopt a similar look while they have completely different semantic meanings.

Example 3-4. Parenthesis Policy Example

```
if (condition)
{
}

while (condition)
{
}

strcpy(s, s1);

return 1;
```

Chapter 4. Class Design

Required Class Methods

To be good citizens almost all classes should implement the following methods. If you don’t have to define and implement any of the "required" methods they should still be represented in your class definition as comments. If you just let the compiler generate them without indicating through comments that you know that this is the intended behaviour, people might wonder about the possibility of an ommision or oversight.

Details

Default Constructor

If your class needs a constructor, make sure to provide one. You need one if during the operation of the class it creates something or does something that needs to be undone when the object dies. This includes creating memory, opening file descriptors, opening transactions etc.

If the default constructor is sufficient add a comment indicating that the compiler-generated version will be used.

If your default constructor has one or more optional arguments, add a comment indicating that it still functions as the default constructor.

Virtual Destructor

If your class is intended to be derived from by other classes then make the destructor virtual. You should always make a destructor virtual for the sake of future extensibility. Only make it non virtual if you’ve got a real good reason to do so.

Copy Constructor

If your class is copyable, either define a copy constructor and assignment operator or add a comment indicating that the compiler-generated versions will be used.

If your class objects should not be copied, make the copy constructor and assignment operator private and don’t define bodies for them. If you don’t know whether the class objects should be copyable, then assume not until the copy operations are needed.

Assignment Operator

If your class is assignable, either define a assignment operator or add a comment indicating that the compiler-generated versions will be used.

If your objects should not be assigned, make the assignment operator private and don’t define bodies for them. If you don’t know whether the class objects should be assignable, then assume not.

Justification

- Virtual destructors ensure objects will be completely destructed regardless of inheritance depth. You don't have to use a virtual destructor when:
  - You don't expect a class to have descendants.
  - The overhead of virtualness would be too much.
  - An object must have a certain data layout and size.
- A default constructor allows an object to be used in an array.
- The copy constructor and assignment operator ensure an object is always properly constructed. Making them private, prevents copies from objects being made without you knowing about it and thus possibly inducing an unnecessary overhead or cause for inconsistency.

Example 4-1. Required Class Methods Example

```
class Planet
{
public:
    // Planet();
    Planet(int radius= 5);
    ~Planet();

private:
    Planet(const Planet&);
    Planet& operator=(const Planet&);
};
```

Class Layout

A common class layout is critical from a code comprehension point of view and for automatically generating documentation. C++ programmers, through a new set of tools, can enjoy the same level generated documentation Java programmers take for granted.

Following is the template that should be used to organize each class declaration.

Example 4-2. Class Layout Template

```
/**
 * A detailed description of the class.
 *
 * @short A short description of the class
 * @author Name of the author
 * @version version
 * @since version
 * @see something
 */

#ifndef XX_h
#define XX_h

// SYSTEM INCLUDES
//

// PROJECT INCLUDES
//
```

```
// LOCAL INCLUDES
//

// FORWARD REFERENCES
//


class XX
{
public:
// LIFECYCLE

    /**
     * Default constructor.
     */
    XX(void);

    /**
     * Copy constructor.
     *
     * @param from The value to copy to this object.
     */
    XX(const XX& from);

    /**
     * Destructor.
     */
    ~XX(void);

// OPERATORS

    /**
     * Assignment operator.
     *
     * @param from the value to assign to this object.
     *
     * @return A reference to this object.
     */
    XX& operator=(XX& from);

// OPERATIONS
// ACCESS
// INQUIRY

protected:
private:

// MEMBER VARIABLES

};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif  // XX_h
```

Initialize all Variables

You shall always initialize variables. Always. Every time.

Justification

- More problems than you can imagine are eventually traced back to a pointer or variable that was left uninitialized. C++ tends to encourage this behaviour by propagating the initialization to the constructors of the parent classes.

Accessor Styles

Accessor methods provide access to the attibutes of an object. Accessing an object's attributes directly, as is commonly done in C structures, is greatly discouraged in C++. It exposes implementation details of the object and degrades encapsulation.

There are two accepted ways to implement accessors, the preferable way is the following :

Attributes as objects

Example 4-3. Attributes as object accessor style

```
class X
{
public:
    int             age() const     { return mAge; }
    int&            rAge()          { return mAge; }

    const String&   name() const    { return mName; }
    String&         rName()         { return mName; }
private:
    int              mAge;
    String           mName;
}
```

The main weakness of this approach is that when returning a reference to basic types (as in the rAge() method), it's impossible to perform checks on the provided value. To solve this, you can either create a wrapper class or resort to the second accessor implementation style. The preferred method is of course, in this example, to create an Age class which contains all consistancy checks as class methods or statements in its constructor.

The advantage of this approach is that it's more consistent with OOP : the object should do it. An object's assignment (=) operator can do all the checks for assignments. This centralizes the consistancy checks in one place, in the object, where it belongs.

When possible, use this approach instead of the One method name accessor style.

One method name

Example 4-4. One method name accessor style

```
class X
{
public:
    int    age() const     { return mAge; }
    void   age(int age)    { mAge = age; }
private:
    int mAge;
```

```
}
```

Using this approach, it's possible to include some checks about the value provided to the age() method. However these checks make only sense when handling basic types since objects should perform their checks internally. Failing to do so would mean that every accessor has to reimplement these consistancy rules over and over again

The hude drawback here is that objects aren't treated in their own right and that encapsulation somewhat fails. It is better to rely on the object's assignment operator.

Class Documentation

For the creation of developer API documentation we're using kdoc. Since we've selected a stripped down version of Qt as our class library it's a great asset that a documentation tool understands the concept of Qt's meta object system, slots and signals. Although this is only available in a more feature-packaged distribution of Qt, it's very probable that in the future, parts of portage will require these additional features and having support for them in the documentation tool is thus a great asset.

Kdoc is very much likely to javadoc, but then for c++.

A documentation comment is a C comment that immediately precedes a class, method, constant or property declaration. It takes the following form:

Example 4-5. Documentation Comment Example

```
/**
* Documentation goes here
*/
class MyClass
{
    ...
```

The double asterisk at the start of the comment differentiates a documentation comment from a normal comment. To make the documentation comment blocks clearly stand out, each line can be preceded by asterisks which will be ignored when the output is generated.

The documentation is a mixture of:

Normal text

Paragraphs must be separated by at least one blank line.

Code fragments

Inline code fragments have to take the following form :

Example 4-6. Documentation inlined code fragment example

```
<pre>
.....code fragments....
</pre>
```

Various kdoc tags

The tags that kdoc understand are all in the following form and should be entered on one line (@ref is an exception):

Example 4-7. The form of kdoc tags

```
@tagname [tag parameters]
```

List of KDoc valid tags

The valid KDoc tags for each type of source code entity are:

Table 4-1. List of valid tags

Classes	
@short [one_sentence]	A short description of the class.
@author [one_sentence]	The class's author.
@version [once_sentence]	The class's version. This can for example be set to the RCS/CVS tag $Id.
@see [references_to_classes_or_methods]	References to other related documentation.
Methods	
@see	as above
@return [one_sentence]	A sentence describing the return value.
@exception [exceptions]	List the exceptions that could be thrown by this method.
@param [param_name] [param_description]	Describe a parameter. The param description can span multiple lines and will be terminated by a blank line, the end of the comment, or another param entry. For this reason, param entries should normally be the last part of the doc comment.
Constants, Enums, Properties	
@see	as above
ANYWHERE	
@ref	As a departure from the javadoc format, the metatag @ref has the same format as @see, but can appear anywhere in the documentation (all other tags must appear on a line by themselves).

III. Language Directives and Best Practices

Chapter 5. Take benefit from C++

Be Const Correct

C++ provides the const keyword. This makes it possible to indicate that a method doesn't modify the objects that it receives as parameters. Using const in all the right places is called "const correctness." It's hard at first, but using const really tightens up your coding style. Const correctness grows on you.

If you're in the darkness about what Const Correctness exactly is, read the relevant section in the C++ FAQ

Use Streams

Programmers that move from C to C++ find stream IO strange and prefer the familiarity of good old stdio. Printf and its derivates seem to be more convenient since they are well understood. However when you use these old idioms, you throw away one of the most powerful features of C++.

Justification

Type Safety

Stdio is not type safe, which is one of the reasons you are using C++, right? Stream IO is type safe.

Standard Interface

When you want to dump an object to a stream there is a standard way of doing it: the $<<$ operator. This is not true of objects and stdio.

Interchangeablity of Streams

One of the more advanced reasons for using streams is that once an object can dump itself to a stream it can dump itself to any stream. One stream may go to the screen, but another stream may be a serial port or network connection. Good stuff.

Uses Of The Constructor

Only initialize member variables

The constructor should only initialize the member variables and preferably not through assignment. Also, explicitely initialize all member variables even if you're just calling their default constructor. It's better to be clear from the beginning than in doubt later.

Delegate all logic

No real action should be done in the constructor, delegate everything to a seperate initialize() method. This will allow multiple constructors to use the shared code logic in the initialize method by passing the appropriate arguments.

Call the parent's constructor explicitely

If the class has been derived from another class, the constructor has to call the appropriate parent's constructor explicitely. Don't rely on the implicite calling of the default constructor since it makes code unclear. It's much better to clearly state that the use of the parent's default constructor has been examined by you and that you think it's the best course of action.

Example 5-1. Uses Of The Constructor Example

```
class Child : public Parent
{
public:
    Child() :
        Parent(),
        mAge(),
        mName("unknown")
    {
        initialize();
    }

    void initialize()
    {
        /* do the real work */
    }

private:
    int       mAge;
    QString   mName;
}
```

Prefer C++ Casts

The traditional way of casting in C is still possible in C++, but alternative options are available which provide much better diagnoses of usage errors and are much easier to indentify and maintain.

These new casts are :

const_cast<T>(e)

Casts away the const-ness of objects, variables or pointers. It gives an error when the types differ more than in const and volatile modifiers.

Don't use this to cast away const-ness of objects that were originally defined as being const and on which non-const operations are being executed. Doing this, results in undefined behaviour.

You typically use this when you need to access an api that incorrectly defines a function signature. When you are 100% sure that the function you want to call doesn't perform non-const operations on the argument, you can safely cast away the const-ness of the argument that was initially defined as being const.

Example 5-2. const_cast Example

```
int countChars(char *pString, char character);

const char *p_somestring = "Let's make things better";
int result = countChars(const_cast<char*>(p_somestring), 'e');
```

dynamic_cast<T>(e)

Makes it possible to safely downcast pointers and references to base classes. It thus returns the appropriate sub-object in the hierarchy chain. It returns 0 if this cast wasn't possible on pointers and throws the bad_cast exception if the cast wasn't possible on references. This effectively says "convert this Object into a Penguin or give me 0 if its not an Penguin,". This provides dynamic typing, you don't know what will happen until run-time.

Example 5-3. dynamic_cast Example

```
class B { /* at least one virtual function */ };
class D : public B { /* ... */ };

B* p_b1 = new B;
B* p_b2 = new D;

     D* p_d1 = dynamic_cast<D*>(p_b1); /* will be 0 */
     D* p_d2 = dynamic_cast<D*>(p_b2); /* will be an object of type D* */
```

reinterpret_cast<T>(e)

This type of casts treats pointers and references as incomplete types. Using the reinterprete_cast yields values that are typically not guaranteed to be usable without casting back to their orginal types. It's difficult to say more about this kind of casts since their applicability is very implementation dependent.

Example 5-4. reinterpret_cast Example

```
void someFunction(char *p_string) { *p_string = 'x'; }
typedef void (*FPType)(const char*);
FPType p_functionpointer = reinterpret_cast<FPType>(&someFunction);
/* calling someFunction through p_functionpointer is not guaranteed to work */
```

static_cast<T>(e)

This is very similar to the old C-style casts. Only use it when none of the above seem to fit the bill. It will only succeed if there's an implicit conversion possible either from T to the type of e, or from the type of e to T.

Example 5-5. static_cast Example

```
Fraction fraction(1,2);
double d = static_cast<double>(fraction);
```

Chapter 6. Best practices

Commenting Out Large Code Blocks

Sometimes large blocks of code need to be commented out for testing. You can't use /**/ style comments because these can't be nested. Surely a large block of your code will contain at least one comment, won't it?

Use #if 0

The easiest way to do this is with an #if 0 block. Don't use #ifdef as someone can unknowingly trigger ifdefs from the compiler command line.

Example 6-1. Commenting Out Large Code Blocks Example

```
void example()
{
    great looking code

    #if 0
    lots of code
    /* a comment */
    some more code
    #endif

    more code
}
```

Use Descriptive Macro Names Instead of 0

The problem with #if 0 is that a while later neither you nor someone else has any idea why this code has been commented out. Is it because a feature has been dropped? Is it because it was buggy? Didn't it compile? Can it be reinstated? It's a mystery.

Therefor you can also chose to use descriptive macro names instead of #if 0.

Example 6-2. Commenting With Descriptive Macro Names

```
void example()
{
    great looking code

    #if NOT_YET_IMPLEMENTED
    travel_through_air();
    #endif

    a bit of code

    #if OBSOLETE
    /* a comment */
    travel_by_foot();
    #endif

    #if TEMP_DISABLED_OUT_OF_GAS
    travel_by_car();
    #endif
}
```

Prefer positive boolean comparisons

It's much easier to think in a positive way about a situation than to be presented with the negative alternative and having to transform it in your mind by yourself to positive. People tend to have a 'logical' or 'the default behaviour' feeling about `true`, which makes it easy to think about. On the contrary, `false` is mostly regarded is the 'exception', 'the error situation' or the 'alternative way out'. Therefor we prefer constructs like this:

Example 6-3. Positive boolean comparison, the right way

```
setup();
if (true == something)
{
    dowork();
}
cleanup();
return;
```

above the following negative counterpart:

Example 6-4. Positive boolean comparison, the wrong way

```
setup();
if (false == something)
{
    cleanup();
    return;
}
dowork();
cleanup();
return;
```

Handle cleanup situations with boolean indicators

Often you're presented with the problem that your code logic contains a series of initializations that can all potentially fail. Typically you want to interrupt any further execution, cleanup and return an error message. Such situations have been known to be resolved through the use of exceptions, gotos, large if-then-else constructs and boolean indicators. From these options, it's the last one we prefer.

Below is an example of such a typical code cleanup situation:

Example 6-5. Cleanup with boolean indicators

```
void some_function()
{
    bool file_setup = false;
    bool dir_setup = false;

    /* try to create a new file object and open it for reading */
    QFile *p_file = new QFile("/path/to/file");
    if (0 != p_file &&
        true == p_file->open(IO_ReadOnly))
    {
        file_setup = true;
    }

    QString dir_path("/path/to/default/dir");
    if (true == file_setup)
    {
        /* if the file was setup, read its contents and use it for */
```

```
        /* further processing */
        QTextStream textstream(p_file);
        QString dir_path = textstream.readLine();
        dir_path = textstream.readLine();
    }

    /* try to create a new dir object and open it for reading */
    QDir *p_dir = new QDir(dir_path);
    /* some vars that are needed by the dir logic */
    if (0 != p_dir &&
        true == p_dir->exists())
    {
        /* do stuff with the dir */
        dir_setup = true;
    }
    else
    {
        cout << dir_path.ascii() <<
            " couldn't be processed" << endl;
    }

    /* cleanup the dir setup if needed*/
    if (true == dir_setup)
    {
        /* cleanup what was done in the dir logic part */
    }
    /* cleanup the file setup if needed*/
    if (true == file_setup)
    {
        p_file->close();
    }

    delete p_dir;
    delete p_file;
}
```

Justification

- You prevent unnecessary consecutive indentations as is the case with large if-then-else constructs.
- You don't have the maintaince hassle of local gotos which could point anywhere and present a number of difficult to solve C++ issues such as accessing object whose initializations has been jumped over.
- It's very easy and clear to follow the logical flow, no jumps are executed as with gotos and exceptions.
- You can perform context-sensitive cleanups that combine the states of several boolean indicators.

Learn about enum classes

In a lot of cases regular enumerations suffice to define a group of valid integer constants as a distinct type. A typical example follows:

Example 6-6. Classic enumeration example

```
typedef enum
{
    ALPHA   = -4, /* this maps to the "alpha" string in the code */
    BETA    = -3, /* this maps to the "beta" string in the code */
    PRE     = -2, /* this maps to the "pre" string in the code */
    RC      = -1, /* this maps to the "rc" string in the code */
    NONE    = 0,  /* this maps to no string (or all the invalid strings) */
    P       = 1   /* this maps to the "p" string in the code */
} SuffixType;
```

This approach however presents a number of drawbacks:

- The values are limited to integers.
- It not possible to evolve the constants in time to provide for example custom output methods, alternative synonimous values (strings for example), conversion from and to other types, and so on ...
- Casting between integers and enums is very error prone, you could cast a value that's not available in the enum.

Therefore it's often justified to write enum classes. These classes contain private constructors and initialize constant instances of themselves as static class variables.

An advanced implementation example follows:

Example 6-7. Enumeration Classes Example

```
/*
* This creates a collection of the possible suffix type instances and
* neatly maintains the textual and numerical representation together.
* No additional instance can be created through the public interface.
* The instances are registered in an internal dictionary which makes it
* very easy to retrieve the exact object that corresponds to a textual
* representation.
* The type of the enum values is a pair, composed out of a string and a
* matching integer.
*/
class SuffixType : public QPair<QString, int>
{
public:
    /**
    * The enumeration's values
    */
    static const SuffixType ALPHA;
    static const SuffixType BETA;
    static const SuffixType PRE;
    static const SuffixType RC;
    static const SuffixType NONE;
    static const SuffixType P;

    /**
     * Retrieve a SuffixType object instance according to its textual
     * representation.
     *
     * @param string The string to look up.
     * @return A SuffixType that correponds to the provided string, or
     *         SuffixType::NONE if no match was found.
     */
    static const SuffixType& get(QString string);

    /**
     * Outputs the textual representation to an output stream.
     */
    friend std::ostream& ::operator<<(std::ostream &rStream, const SuffixType &rSuf
```

```
private:
    /**
    * Private constructor that will only be called during the initialization
    * of the static class variables.
    */
    SuffixType(QString string, int value);

    /**
    * Class wide collection that maps string representations of class
    * instances to the instances themselves.
    */
    static QDict<SuffixType> *mpMap;
};

/* Define and initialize the enumeration values. */
const SuffixType SuffixType::ALPHA("alpha", -4);
const SuffixType SuffixType::BETA("beta", -3);
const SuffixType SuffixType::PRE("pre", -2);
const SuffixType SuffixType::RC("rc", -1);
const SuffixType SuffixType::NONE("none", 0);
const SuffixType SuffixType::P("p", 1);

/* Define the static class-wide dictionary */
QDict<SuffixType> *SuffixType::mpMap(0);

SuffixType::SuffixType(QString string, int value) :
    QPair<QString, int>(string, value)
{
    /* Initialize the class-wide dictionary if this hasn't been done yet. */
    if (0 == mpMap)
    {
        mpMap = new QDict<SuffixType>(7);
    }

    /* Register this class instance in the class-wide dictionary. */
    SuffixType::mpMap->insert(string, this);
}

const SuffixType& SuffixType::get(QString string)
{
    const SuffixType *p_matchingsuffix = 0;

    p_matchingsuffix = SuffixType::mpMap->find(string);

    /* If no match was found, return the null suffix. */
    if (0 == p_matchingsuffix)
    {
        p_matchingsuffix = &SuffixType::NONE;
    }
    return *p_matchingsuffix;
}

ostream& operator<<(ostream &rStream, const SuffixType &rSuffixType)
{
    rStream << rSuffixType.first.ascii();
    return rStream;
}
```

Although this presents quite some more code initially, it pays of immensely afterwards since you can easily access both the integer as the string value of an enumeration instance by using for example : `SuffixType::ALPHA.first` and `SuffixType::ALPHA.second`. This will provide you with `alpha` and `-4`. Also, to retrieve which enumeration instance corresponds to a given text you can simply use : `SuffixType::get("alpha")` and get a reference to the `SuffixType::ALPHA` instance in return.

This approach thus neatly centralizes all enumeration logic instead of scattering it all over the code with a large number of unintuitive conditional statement blocks.

Appendix A. Todo

- exceptions <> return values
- when to return a pointer, a value or a reference from a method
- indenting of variable declarations
- unit testing
- no magic numbers
- tabs/spaces policy
- how to break lines
- don't mix pointers and non-pointers on a single declaration line
- use 0 instead of NULL
- #define <> const <> template functions
- match new ~ delete and new[] ~ delete[], don't use malloc ~ free (initialization, destruction, not overloadable)
- base classes virtual destructor
- assignment to self in operator=
- cont correctness and pointers, uses of mutable
- postponed variable declaration at beginning of relevant code block
- public inheritance : isa, layering/composition : hasa is-implemented-in-terms-of, pure virtual function : interface inheritance, simple virtual function : interface inheritance with default implementation (dangerous : sever connection with pure virtual function and explicit default implementation), non virtual function : interface inheritance + mandatory implementation, private inheritance : is-implemented-in-terms-of + implementation inheritance not interface, default parameters and virtual function.
- say what you mean, mean what you say
- inheritance (type of objects changes behaviour) <> templates (type of objects doesn't change behaviour of methods)

Bibliography

Books

Scott Meyers, 0-201-92488-9, Addison-Wesley Publishing Company, *Effective C++*.

Margaret A. Ellis and Bjarne Stroustrup, 0-201-51459-1, Addison-Wesley Publishing Company, *The Annoted C++ Reference Manual*.

Websites

Todd's C++ Coding Standard, Todd Hoff, http://www.possibility.com/Cpp/CppCodingStandard.html.

Kdoc homepage, Sirtaj S. Kang, http://www.ph.unimelb.edu.au/~ssk/kde/kdoc/.

How to Write Doc Comments for the Javadoc Tool, Sun Microsystems, http://java.sun.com/j2se/javadoc/writingdoccomments/index.html.

C++ Faq Lite, Marshall Cline, http://www.parashift.com/c++-faq-lite.