C++ programming guidelines

1. Language independent rules

1.1. Unary operators

Correct:	`v++`
	`--v`
	`-10`

Wrong:	`-- v`
	`- 10`

1.2. Binary operator

Use always exactly one separating blank on the left and on the right side of a binary operator.

Correct:	`a + b`
	`a - b`
	`a * b`
	`a / b`
	`a & b`
	`a | b`
	`a ^ b`
	`a << b`
	`a >> b`
	`a <= b`
	`a >= b`
	`a != b`
	`a == b`
	`a && b`
	`a || b`
	`a = b * c`
	`a *= b`

Wrong:	`a   + b`
	`a- b`
	`a ==b`
	`a*b`
	`a = b*c`
	`a *=b`

1.3. Brackets

There is no separating blank allowed between the outer brackets and the inner expression.

Correct:	`(a + b)`
	`((a - b) * (a + b))`

Wrong:	`( a - b)`

	`( (a - b) * (a + b))`

1.4. Statements and statement delimiters

- Only one statement per line is allowed.
- There is no blank between the statement and its statement delimiter.

Correct:	`a++;`

Wrong:	`a++ ;`
	`a++; b++;`

1.5. Indentation

- Tabulator characters are not allowed for indentation.
- Two space characters represent one indentation level.
- The indentation level is initialised with zero.
- Entering a block (all statements between the '{' and '}' brackets) increases this indentation level by one.

1.6. Font

It is recommended to use a font with a fixed width like *Courier*. The C++ guidelines are optimised for such fonts.

1.7. Line length

- 100 characters is the maximum length of a line.
- Each single statement that is longer than 100 characters must be splitted into several lines. The indentation level of the first line is increased by two for all other lines.

2. Control structures

Do not use the following statements:

- Continue
- Break (only allowed for the termination of a switch case)
- Goto

The correct formatting of the control structures is described by EBNF syntax in the following sections. Additionally two important breaking rules must be defined.

- Statements line must be broken at the start of a new parameter, if the line is too long. If this rule fails the breaking rule of expressions must be applied. When the expression rule also has failed the break should be made at a place, which looks good for the programmer. The indentation level of each new line is increased by two. This means four additional indentation characters at the position of the first statement character.
- Expressions that are too long for the line must be broken at the first character of an operand expression. Higher order operand terms are preferred. The indentation position is the same position as the first character of the broken expression.

EBNF syntax of an expression:

```
EXPRESSION =
     '(' EXPRESSION ') ' OPERATOR ' (' EXPRESSION  ')'
```

```
    { OPERATOR ‘ (’ EXPRESSION  ‘)’ } |
    TERM { ‘ ’ OPERATOR ‘ ’ TERM }.
```

Example that has to be broken:	(a && b) || (c && d)
Correct:	preferred break: (a && b) || (c && d)
	or: (a && b) || (c && d)
Wrong:	(a && b) || (c && d)
	(a && b) || (c && d)
	(a && b) || (c && d)
	(a && b) || (c && d)

2.1. if

EBNF syntax :

```
IF_STATEMENT =
    ‘if (’ EXPRESSION ‘) {’ NEWLINE {
      ADD_IDENT STATEMENT_SEQUENZE NEWLINE DEC_IDENT
    ‘} else if (’ EXPRESSION ‘) {’ NEWLINE
      ADD_IDENT STATEMENT_SEQUENZE NEWLINE DEC_IDENT } [
    ‘} else {’ NEWLINE
      ADD_IDENT STATEMENT_SEQUENZE NEWLINE DEC_IDENT ]
    ‘}’ NEWLINE .
```

If the first line is too long it must be broken by the expression rule.

Correct:	if (expression) {
	  statement;
	}
	
	if (expression1 &&
	    expression2) {
	  statement;
	}
	
	if (expression) {
	  statement1;
	} else {
	  statement2;
	}
	
	if (expression1) {

	`  statement1;`
	`} else if (expression2) {`
	`  statement2;`
	`}`
	
	`if (expression1) {`
	`  statement1;`
	`} else if (expression2) {`
	`  statement2;`
	`} else {`
	`  statement3;`
	`}`

Wrong:	`if (expression)`
	`{`
	`  statement;`
	`}`
	
	`if (expression){`
	`  statement;`
	`}`
	
	`if(expression) {`
	`  statement;`
	`}`
	
	`if (expression) {`
	`  statement1;`
	`}else {`
	`  statement2;`
	`}`
	
	`if (expression) {`
	`  statement1;`
	`} else{`
	`  statement2;`
	`}`
	`if (expression) statement;`

2.2. while

EBNF syntax :

```
WHILE_STATEMENT =
    'while (' EXPRESSION ') {' NEWLINE
      ADD_IDENT STATEMENT_SEQUENZE NEWLINE DEC_IDENT
    '}' NEWLINE .
```

If the first line is too long it must be broken by the expression rule.

Correct:	`while (expression) {`
	`  statement;`
	`}`

	`while (expression1 &&`
	`       expression2) {`
	`  statement;`
	`}`

Wrong:	`while (expression)`
	`{`
	`  statement;`
	`}`
	
	`while (expression)`
	`statement;`

2.3. for

EBNF syntax :

```
FOR_STATEMENT =
    'for (' [ STATEMENT ] ';' [ ' ' EXPRESSION ] ';'
            [ ' ' STATEMENT ] ') {' NEWLINE
      ADD_IDENT STATEMENT_SEQUENZE NEWLINE DEC_IDENT
    '}' NEWLINE .
```

If the first line is too long you must break the line after each `';'`. If one of the new lines is still too long you must break the affected line by applying the breaking rules of statements and expressions.

Correct:	`for (int i = 0; i < 10; i++)`
	`{`
	`  statement;`
	`}`
	
	`for (int realLongCounter =`
	`0;`
	`     realLongCounter < 10;`
	`     realLongCounter ++) {`
	`  statement;`
	`}`
	`for (; i < 10; i++) {`
	`  statement;`
	`}`
	
	`for (; i < 10;) {`
	`  statement;`
	`}`
	
	`for (;;) {`
	`  if (expression) {`
	`   return 10;`
	`  }`
	`  statement;`
	`}`

Wrong:	`for (int i = 0; i < 10; i++)`
	`{`
	`  statement;`

	`}`
	
	`for (int i = 0;i < 10;i++) {`
	`  statement;`
	`}`
	
	`for (int i = 0;i < 10;i++)`
	`  statement;`

2.4. do while

EBNF-Syntax :

```
DO_WHILE_STATEMENT =
    'do {' NEWLINE
      INC_IDENT STATEMENT_SEQUENZE NEWLINE DEC_IDENT
    '} while (' EXPRESSION ');' NEWLINE .
```

If the last line is too long it must be broken by the expression rule.

Correct:	`do {`
	`  statement;`
	`} while (expression);`
	
	`do {`
	`  statement;`
	`} while (expression1 &&`
	`         expression2);`

Wrong:	`do`
	`{`
	`  statement;`
	`} while (expression);`
	
	`do statement;`
	`while (expression);`

2.5. switch

EBNF-Syntax :

```
SWITCH_STATEMENT =
    'switch (' EXPRESSION ') {' NEWLINE {
      ADD_IDENT 'case ' CONST ':' NEWLINE
        ADD_IDENT STATEMENT_SEQUENZE NEWLINE
        [ 'break;' NEWLINE ] DEC_IDENT DEC_IDENT } [
      ADD_IDENT 'default ' CONST ':' NEWLINE
        ADD_IDENT STATEMENT_SEQUENZE NEWLINE
        [ 'break;' NEWLINE ] DEC_IDENT DEC_IDENT ]
     '}' NEWLINE .
```

If the first line is too long it must be broken by the expression rule.

Correct:	`switch (x) {`
	`  case 1:`
	`    statement1;`
	`    break;`
	`  default:`
	`    statement2;`

	`    break;`
	`}`
	`switch (expression1 +`
	`        expression2) {`
	`  case 1:`
	`    statement1;`
	`    break;`
	`}`
Wrong:	`switch (x) {`
	`case 10:`
	`  statement1;`
	`  break;`
	`default:`
	`  statement2;`
	`  break;`
	`}`
	`switch (x) {`
	`  case 10: statement1;`
	`break;`
	`  default: statement2;`
	`break;`
	`}`

3. Names

3.1. Language

All names must be in English.

3.2. Methods, namespaces, enumerations, variables and attributes

A name must be build of lowercase words making sense and describing the variable or attribute. To separate two words in a name you must write the first letter of the second uppercase. Summarizing all characters of the name must be lowercase or a number, except the first one and all characters separating two words.
Don't use acronyms with a small number of characters like 'dba' which could mean 'data base architecture' or 'data base access' or 'data base administrator' or 'double bold arrow' and so on.
Don't use characters in front of the name that describes meta information like type or visibility.

3.3. Classes and types

The name is build like in the previous section. The difference it that the first character is uppercase.

3.4. Defined constant values and macros

All letters of defines (*#define EXAMPLE_NAME*...) have to be uppercase and the words are separated by a `'_'` character.

EBNF syntax:
UPPERCASE_IDENTIFIER = all characters are uppercase and numbers are also allowed.
DEFINE =
    UPPERCASE_IDENTIFIER { '_' UPPERCASE_ IDENTIFIER } .

4. Classes

- All fields of a class and methods are sorted by static modifier, kind (attributes, constructors, destructors) and access (public, protected and private).
- All static fields are at the top of the class.
- Inside of this two blocks the fields are sorted by the kind: attributes comes first, constructors and destructors comes next and methods comes at last.
- Constructors come before destructors.
- Virtual methods (dynamic binding at runtime) come before non-virtual methods (static binding at compile time).
- All kind of fields are sorted by their access: 'public' comes first, 'protected' next and 'private' at last.
- 'public', 'protected' and 'private' are on the same indentation level as 'class'
- Inner classes are indented by one indentation level (two additional space characters).
- A method names starts with a lowercase character.
- Get- and set-methods should be used to access attributes (get-methods for Boolean value can be named with 'is…' instead of 'get…').
- Virtual methods should only be used for methods that really need dynamic binding at runtime.
- Abstract virtual methods should not be implemented (use instead an assignment of zero to the method table) to ensure (by the compiler) that the abstract class and method are not directly used.
- The use of inline methods is allowed to optimise the performance or to create template classes.
- Class and type names must begin with an uppercase character.

EBNF syntax:
```
CLASS =
    'class ' CLASS_NAME ' : ' BASE_CLASSES ' {'
    'public:' NEWLINE STATIC_ATTRIBUTES SEPERATINGLINE
    'protected:' NEWLINE STATIC_ATTRIBUTES SEPERATINGLINE
    'private:' NEWLINE STATIC_ATTRIBUTES SEPERATINGLINE

    'public:' NEWLINE STATIC_METHODS SEPERATINGLINE
    'protected:' NEWLINE STATIC_METHODS SEPERATINGLINE
    'private:' NEWLINE STATIC_METHODS SEPERATINGLINE

    'public:' NEWLINE ATTRIBUTES SEPERATINGLINE
    'protected:' NEWLINE ATTRIBUTES SEPERATINGLINE
    'private:' NEWLINE ATTRIBUTES SEPERATINGLINE

    'public:' NEWLINE CONSTRUCTORS SEPERATINGLINE
    'protected:' NEWLINE CONSTRUCTORS SEPERATINGLINE
    'private:' NEWLINE CONSTRUCTORS SEPERATINGLINE

    'public:' NEWLINE METHODS SEPERATINGLINE
    'protected:' NEWLINE METHODS SEPERATINGLINE
    'private:' NEWLINE METHODS SEPERATINGLINE

    'public:' NEWLINE INNER_CLASSES SEPERATINGLINE
    'protected:' NEWLINE INNER_CLASSES SEPERATINGLINE
    'private:' NEWLINE INNER_CLASSES
    '};' NEWLINE .

SEPERATINGLINE = if more fields in class then NEWLINE
    otherwise nothing .

ATTRIBUTES =
    INC_INDENT
    ATTRIBUTES NEWLINE { ATTRIBUTE NEWLINE }
    DEC_INDENT .

CONSTRUCTORS =
    INC_INDENT
    CONSTRUCTOR NEWLINE { CONSTRUCTOR NEWLINE }
    [ DESTRUCTOR NEWLINE { DESTRUCTOR NEWLINE } ]
    DEC_INDENT .

METHODS =
    INC_INDENT
   'virtual ' METHOD NEWLINE { 'virtual ' METHOD NEWLINE
    }
   SEPERATINGLINE
    METHOD NEWLINE { METHOD NEWLINE }
    DEC_INDENT .

INNER_CLASSES =
    INC_INDENT
    INNER_CLASS SEPERATINGLINE
    { INNER_CLASS SEPERATINGLINE }
    DEC_INDENT .
```

Example:

```
class MyClassInherited : public MyClass {
  public:
    static int x;

  protected:
    static int y;
  private:
    static int z;

  public:
    static int getX();

  protected:
    static int getY();

  private:
    static int getZ();

  public:
    int a;
    int b;

  protected:
    int c;
    int d;

  private:
    int e;
    int f;

  public:
    MyClassInherited();

  protected:
    MyClassInherited(int y);

  private:
    MyClassInherited(bool z);

  public:
    virtual ~MyClassInherited();
    virtual void h(int x = 1) = 0;

    void g();

  protected:
    virtual void i() = 0;

    void j();

  private:
    void k();
    void l();
};
```

5. Namespaces

Use namespaces to ensure unique class and function names. The namespace doesn't affect defined macros of constants. So the name of a define is very important to prevent several defines with the same name. Instead of defines you must prefer enumerations (maybe in combination with a type definition).

Example: Instead of

```
#define FRUIT_APPLE 0
#define FRUIT_PEACH 1
#define FRUIT_PEAR 2

void drawFruit(int fruit);
```

use this

```
namespace fruits {
  typedef enum {
    apple = 0,
    peach,
    pear
  } Fruit;

  void drawFruit(Fruit fruit);
}
```

The name of namespaces must reflect the purpose of this package. It must begin with a lowercase letter. Also the project name should be included in the namespace. The best way to do this is to create a namespace with the project name containing all other namespaces (you can also add the company name to the namespace, when the project name is not unique).
In source files the use of `'use namespace NAMESPACE_NAME;'` is allowed to reduce the indentation level and to use the namespace content without the complete qualifier.

EBNF syntax:

NAMESPACE = 'namespace ' *NAMESPACE_NAME* ' {' *NEWLINE* *INC_INDENT* *NAMESPACE_CONTENT* *DEC_INDENT* *NEWLINE* '}' *NEWLINE* .

NAMESPACE_NAME = name describing the functionality provided by the namespace content (first word is completely lowercase the next words are lowercase except each first character is uppercase) .

NAMESPACE_CONTENT = contains other namespaces, classes, enumerations, types, functions, methods, and so on .

6. Header and Source files

To ensure that a header file can be included more than once you must provide information to the pre-processor that this file is already included. Additionally in header file only includes with `'<'` and `'>'` brackets are allowed to ensure that the compiler is able to find the header files. In source file you must use includes with `'"'` characters for project internal headers to speedup the time of compilation. For including header files of other projects or libraries you must use the includes with `'<'` and `'<'` brackets.

6.1. Prevent re-including

The prevention of re-including is based on the pre-processor. It uses defines and unique names for the header files. So the first problem is to get a unique name of the header file. Therefore use only one namespaces path in a header file. Based on this convention you can use the namespace and header file name to create a unique define name for the header file.

EBNF syntax:
```
UNIQUE_HEADER_FILE_NAME =
     '_' { UPPERCASE_NAMESPACE_NAME '_' }
     UPPERCASE_HEADER_FILE_NAME .
```

At the top of the header file you must ask the pre-processor if the unique header file name is not defined (*`#ifndef`*). The pre-processor will only consider the next lines when it isn't defined. So we have to define the unique header file name. After the definition you can put the content of the header file and at the end you must end the pre-processor if-command (*`#ifndef`*) with *`#endif`*.

Example:
```
#ifndef _NAMESPACE_FILENAME_H
#define _NAMESPACE_FILENAME_H

// content ...

#endif
```

6.2. Head information

The header and source file must contain a head with the information about:
- Author
- Creation date
- Modification date
- Small description
- Copyright
- Licence
- …

Example of a head information:
```
/* Description:   This is a sample demonstrating the head
 *                information of a header or source file.
 *
 * Copyright by:  FH Hagenberg
 *                Studiengang Medientechnik und -design
 *                Hauptstrasse 117
 *                A-4232 Hagenberg
 *                mailto://mtd@fh-hagenberg.at
 *                http://mtd.fh-hagenberg.at
 *
 * Licence:       See Licence.txt
 *
 * Author:        Juergen Zauner
 *                mailto://juergen.zauner@fh-hagenberg.at
 *
 * Co-authors:    Werner Hartmann
 *                mailto://whartmann@faw.uni-linz.ac.at
 *
 * Verision:      0.0.1
 *
 * Created:       2002-06-18
 * Last modified: 2002-06-21
 */
```

7. Documentation

Documentation is always a trade-off between too much information and too less information. In the source code for example only a required minimum should be documented. When the documentation for one specific problem gets too long it often points to a problem concern the source code quality. Try to think over the affected code and maybe you find another solution that is more satisfying and requires less documentation. The best code is the code that explains itself and doesn't need any comment.
The documentation of header files is more important than the documentation of source files. Source files are only visible and maintained by a small set of developers. So for the maintenance process it's more important to get a clean and high quality source code understandable for each developer. Header files are used by developers who want to use the functionality of the library. Therefore the functionality and behaviour of the header file content (classes, methods, attributes, variables, types, macros, and so on) must be explained by the documentation. The behaviour also includes the behaviour at a failure situation and not only the default behaviour.
An important problem of documentation is its validity. Documenting virtual method in each derived class would create a huge set of redundant documentation, which has to be maintained when the default behaviour of the base class changes. The solution for this problem is to document only the virtual methods of the base classes. Additional behaviour that differs from the behaviour of the base class method must be documented at the method of the derived class.
The use of documentation tools that create HTML documents out of the source and header files should be used. For example:

- doxygen (http://www.stack.nl/~dimitri/doxygen/index.html),
- ccdoc (http://www.joelinoff.com/ccdoc/) or
- doc++ (http://www.zib.de/Visual/software/doc++/index.html).

When you have decided to use one tool it should be used for the whole project to ensure an unique look and feel. They often support to copy the documentation of base class methods to undocumented methods of the derived classes. You must add a kind of reference for the undocumented method of a derived class pointing to the base documentation if the tool does not provide the copy functionality.