GENERAL

Use real tabs that equal 4 spaces.

Use typically trailing braces everywhere (if, else, functions, structures, typedefs, class definitions, etc.)

```
if ( x ) {
}
```

The else statement starts on the same line as the last closing brace.

```
if ( x ) {
} else {
}
```

Pad parenthesized expressions with spaces

```
if ( x ) {
}
```

Instead of

```
if (x) {
}
```

And

```
x = ( y * 0.5f );
```

Instead of

```
x = (y * 0.5f);
```

Use precision specification for floating point values unless there is an explicit need for a double.

```
float f = 0.5f;
```

Instead of

```
float f = 0.5;
```

And

```
float f = 1.0f;
```

Instead of

```
float f = 1.f;
```

Function names start with an upper case:

```
void Function( void );
```

In multi-word function names each word starts with an upper case:

```
void ThisFunctionDoesSomething( void );
```

The standard header for functions is:

```
/*
====================
FunctionName

  Description
====================
*/
```

Variable names start with a lower case character.

```
float x;
```

In multi-word variable names the first word starts with a lower case character and each successive word starts with an upper case.

```
float maxDistanceFromPlane;
```

Typedef names use the same naming convention as variables, however they always end with "_t".

```
typedef int fileHandle_t;
```

Struct names use the same naming convention as variables, however they always end with "_t".

```
struct renderEntity_t;
```

Enum names use the same naming convention as variables, however they always end with "_t". The enum constants use all upper case characters. Multiple words are separated with an underscore.

```
enum contact_t {
     CONTACT_NONE,
     CONTACT_EDGE,
     CONTACT_MODELVERTEX,
     CONTACT_TRMVERTEX
};
```

Names of recursive functions end with "_r"

```
void WalkBSP_r( int node );
```

Defined names use all upper case characters. Multiple words are separated with an underscore.

```
#define SIDE_FRONT       0
```

Use 'const' as much as possible.

Use:

```
const int *p;            // pointer to const int
int * const p;           // const pointer to int
const int * const p;     // const pointer to const int
```

Don't use:

```
int const *p;
```

CLASSES

The standard header for a class is:

```
/*
====================================================================================

	Description

====================================================================================
*/
```

Class names start with "id" and each successive word starts with an upper case.

```
class idVec3;
```

Class variables have the same naming convention as variables.

```
class idVec3 {
	float		x;
	float		y;
	float		z;
}
```

Class methods have the same naming convention as functions.

```
class idVec3 {
	float		Length( void ) const;
}
```

Indent the names of class variables and class methods to make nice columns. The variable type or method return type is in the first column and the variable name or method name is in the second column.

```
class idVec3 {
	float		x;
	float		y;
	float		z;
	float		Length( void ) const;
	const float *	ToFloatPtr( void ) const;
}
```

The * of the pointer is in the first column because it improves readability when considered part of the type.

Ording of class variables and methods should be as follows:

1. list of friend classes
2. public variables
3. public methods
4. protected variables
5. protected methods
6. private variables
7. private methods

This allows the public interface to be easily found at the beginning of the class.

Always make class methods 'const' when they do not modify any class variables.

Avoid use of 'const_cast'. When object is needed to be modified, but only const versions are accessible, create a function that clearly gives an editable version of the object. This keeps the control of the 'const-ness' in the hands of the object and not the user.

Return 'const' objects unless the general usage of the object is to change its state. For example, media objects like idDecls should be const to a majority of the code, while idEntity objects tend to have their state modified by a variety of systems, and so are ok to leave non-const.

Function overloading should be avoided in most cases. For example, instead of:

```
    const idAnim * GetAnim( int index ) const;
    const idAnim * GetAnim( const char *name ) const;
    const idAnim * GetAnim( float randomDiversity ) const;
```

Use:

```
    const idAnim * GetAnimByIndex( int index ) const;
    const idAnim * GetAnimByName( const char *name ) const;
    const idAnim * GetRandomAnim( float randomDiversity ) const;
```

Explicitly named functions tend to be less prone to programmer error and inadvertent calls to functions due to wrong data types being passed in as arguments. Example:

```
Anim = GetAnim( 0 );
```

This could be meant as a call to get a random animation, but the

compiler would interpret it as a call to get one by index.

Overloading functions for the sake of adding 'const' accessible function is allowable:

```
class idAnimatedEntity : public idEntity {
     idAnimator *        GetAnimator( void );
     const idAnimator *  GetAnimator( void ) const;
};
```

In this case, a const version of GetAnimator was provided in order to allow GetAnimator to be called from const functions. Since idAnimatedEntity is normally a non-const object, this is allowable. For a media type, which is normally const, operator overloading should be avoided:

```
class idDeclMD5 : public idDecl {
     const idMD5Anim *   GetAnim( animHandle_t handle ) const;
     idMD5Anim *         GetEditableAnim( animHandle_t handle );
};
```

id Studio Names

```
id<name>Dlg      // dialog class
id<name>Ctrl     // dialog control class
id<name>Frm      // frame window
id<name>View     // view window
id<name>         // any other class
```

FILE NAMES

Each class should be in a seperate source file unless it makes sense to group several smaller classes.
The file name should be the same as the name of the class without the "id" prefix. (Upper/lower case is preserved.)

```
class idWinding;

files:

Winding.cpp
Winding.h
```

When a class spans across multiple files these files have a name that starts with the name of the class without "id", followed by an underscore and a subsection name.

```
class idRenderWorld;


files:

RenderWorld_load.cpp
RenderWorld_demo.cpp
RenderWorld_portals.cpp
```

When a class is a public virtual interface to a subsystem the public interface is implemented in a header file with the name of the class without "id". The definition of the class that implements the subsystem is placed in a header file with the name of the class without "id" and ends with "_local.h". The implementation of the subsystem is placed in a cpp file with the name of the class without "id".

```
class idRenderWorld;

RenderWorld.h            // public virtual idRenderWorld interface
RenderWorld_local.h      // definition of class idRenderWorldLocal
RenderWorld.cpp          // implementation of idRenderWorldLocal
```