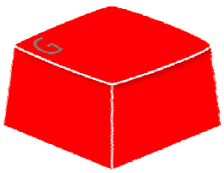
Updater

Code Conventions



Document ID	Updater Code Conventions
Date	2006-02-20
Version	1.0
Author	Geert van Horrik
Website	http://www.gvhsoftware.org

Disclaimer

Table of contents

Definitions, acronyms and abbreviations

Word	Explanation
DLL	Dynamic link library, an extension for an application in a Windows operating system.
Executable	An application that can be launched in a Windows operating system.
Project file	Visual Studio works with projects. A project contains multiple source files which can be compiled to one single executable or DLL.
Solution file	Visual Studio works with solutions. A solution is a file that can hold several different projects together.

1. Introduction

This document describes the code conventions that are and should be used in when developing Updater. All software developers are expected to develop the software using the conventions described in this document.

The coding conventions will make the code better readable, better to understand and only one style of developing will be used.

2. Files

2.1. Solution and project files

There is always a solution that can hold one or more projects. This solution should have a clear name. A solution is always in a folder with the same name as the solution.

The projects that are added to the solution should get a clear name. The project should be placed in a subfolder of the solution. The name of the folder should be the same as the project name.

2.2. Naming

There are two different types of files used for the source code. The name of the file should be exactly the same as the name of the class. When a class starts with a C (for example CString), the filename will not contain the first C. A header file will have the extension .h, a source file will have the extension .cpp. An example follows:

Class name	Header file	Source file
CString	String.h	String.cpp
CComboBox	ComboBox.h	ComboBox.cpp

When 3rd party files are used in a project, the files should be located in a subfolder of the project. This subfolder should have a clear name.

Every file can only include one class.

3. Structure

3.1. Introduction

Every file that is created should have a specific format. The different parts that should be implemented in every file are noted.

- Copyright block;
- Redefinition protected (.h only);
- Includes;
- Class declaration;
- Public functions declaration / implementation;
- Protected functions declaration / implementation;
- Protected variables declaration.

3.2. Elements

3.2.1. Copyright block

Every file (.h and .cpp) must have a copyright block at the top of the file. The block should look like this:

```
/*****************************************************************|
|                                                                 |
|       [FILENAME]                                                |
|       [DEVELOPER(S) OF FILE]                                    |
|       Date: [YYYY-MM-DD]                                        |
|                                                                 |
|       Copyright (C) [YEAR] [DEVELOPER(S)]                       |
|       All rights reserved.                                      |
|                                                                 |
|*****************************************************************/
```

The copyright block must be unchanged and may not be deleted. The filename, developers and date should be inserted in the block. The date is the latest date that the file is changed.

3.2.2. Redefinition protection

The redefinition protection must be added to every .h file. The start of the protection starts immediately after the copyright block and looks like this:

```
// Redefinition
#ifndef [FILENAME]_H
#define [FILENAME]_H
```

The end of the protection must be located at the bottom of every .h file and must look like this:

```
// End of redefinition protection
#endif // [FILENAME] H
```

3.2.3. Includes

The file includes must be located at the top of the file, immediately after the copyright block.

```
//*********************************************************************
// INCLUDES
//*********************************************************************
```

After this header, all the files should be included. After each include should be a comment which explains why the file should be included.

3.2.4. Class declaration

The class declaration in .h file should look like this:

```
//*****************************************************************
// [CLASSNAME]
//*****************************************************************

class [CLASSNAME] : public [BASECLASS]
{
public:
      // Constructor & destructor
      [CLASSNAME]();
      ~[CLASSNAME]();

      // Functions
      void        Function1();
      CString     Function2();

      // Dialog data
      enum { IDD = IDD_DIALOGID };

protected:
      // Functions
      void        Function3();
      void        Function4();

      // Variables
      int         m_iVariable1;
      CString     m_sVariable2;

      // Message map
      DECLARE_MESSAGE_MAP();
};
```

3.2.5. Function implementation

An implementation of the class in a .cpp file always starts with the constructor and destructor. The constructor and destructor part should be started with the following text:

```
//*****************************************************************
// CONSTRUCTOR & DESTRUCTOR
//*****************************************************************
```

A special function splitter should separate each function:

```
//=================================================================
```

After the constructor and destructor, the public functions must be implemented. This part also starts with a specific part of text, which will follow:

```
//*****************************************************************
// PUBLIC FUNCTIONS
//*****************************************************************
```

After the public functions part, the last part should be implemented in the .cpp file. The protected / private functions part should start with this text:

```
//*************************************************************
// PROTECTED / PRIVATE FUNCTIONS
//*************************************************************
```

All variables used in a function must be declared at the beginning of each function.

4. Naming

4.1. Classes

The name of the class is always clear. The class name must be exactly the same as the filename, and always starts with a capital. Each class should be started with a C, which should not be entered in the filename of the class.

For example:

Class name	Header file	Source file
CString	String.h	String.cpp
CComboBox	ComboBox.h	ComboBox.cpp

4.2. Variables

4.2.1. General

A variable always starts with a lowercase character. The so-called “Hungarian notation” will be used, which means each first part of a variable name explains the type of the variable. A list with prefixes follows:

Type	Prefix	Example
string as char array	sz	szValue
string as std::string or CString	s	sValue
boolean	b	bValue
integer	i	iValue
double	d	dValue
long	l	lValue
short integer	n	nValue
array	arr	arrData
pointer to object	p	pValue
object	o	oValue

Each object that is not listed (so classes and special types) should start with o which stands for a standard object.

All variables must be initialized to prevent unreliable behavior of the application.

4.2.2. Member variables

A member variable (of a class) should always have the prefix m_. An example of a member variable is:

```
int m_iValue;
```

4.2.3. Local variables

Local variables must be declared (and initialized) at the beginning of each function. The local variables must use the Hungarian notation, but will not contain the prefix m_.

The only exception that a local variable can be declared in the middle of a function is a loop variable in a for-loop.

4.3. Controls

The controls must also use the Hungarian notation. A list of possible controls follows:

Type	Prefix	Example
Dialog	dlg	dlgValue
Button	btn	btnValue
Label	lbl	lblValue
Edit	txt	txtValue
Combobox	cbo	cboValue
Listbox	lst	lstValue
Image/Picture	pic	picValue
Timer	tmr	tmrData
Checkbox	chk	chkValue
Progressbar	prg	prgValue

Each control that is not listed should start with a self-defined prefix which should be clear.

4.4. Functions

A function always starts with a capital. The function name should be clear and explain shortly what action the function will perform. Each word in the function starts with a capital. An example follows:

```
int GetThisValue();
```

4.5. Constants

Constants are always defined in uppercase (CAPITAL). An example:

```
int CONSTANTVALUE = 1;
```

5. Empty spaces

5.1. Introduction

Empty spaces in source code make the source better readable. The following chapters explain how to use the empty spaces.

5.2. White lines

White lines make code better readable by grouping parts of code. Only in these cases, an empty line must be used:

- Between all elements described in chapter 3.2 Elements;
- Between function declarations;
- At locations where the readability can be improved by adding white lines. The developer should decide the use of the white lines.

5.3. Spaces

In the following locations, *no* spaces will be used:

- Between the name of a function and the parenthesis;
- Before and after a dot (“.”);
- Between the minus (“-“) and a number when it is not a calculation (for example “-5” instead of “- 5”);
- Between a typecast and the object;
- Between a opening parenthesis (“(”) and a closing parenthesis (“)”);
- Before a semicolon (“;”)
- Before a colon (“:”);
- Before a comma (“,”).

In the following locations, spaces will be used:

- Between if, while, for and switch and the opening parenthesis (“(”);
- Before and after a binary operation (“=”, “++”, “+=”, “-=”, “--”, etc);
- After a colon (“:”);

- After a comma (“,”);
- Where the compiler needs a space (between identifiers, etc).

5.4. Indent

A normal indent is one tab. Don’t use spaces to indent. After every curly bracket (“{”), there should be one tab indent. Also, when a function header is separated over two or more lines, the following lines should contain one tab indent.

6. Comments

6.1. Introduction

C/C++ supports two types of comments, a single line and a block of comments.

All comments must be written in English

In the next cases, comments can be useful:

- At the beginning of a function, to explain the use of the function;
- Before a class definition, to explain the goal of a class;
- At each block of code, to explain the goal of that block of code.

6.2. Comment blocks

Only use comments with /* and */ when the comments are longer than one single line.

6.3. Comment lines

A comment line is prefixed with two slashes ("//") followed by the text. Between the slashes and the text should be a space. A comment line is located one line before the code it belongs to, not after the code it belongs to. The comment line should be indented the same as the code it belongs to.

It is possible to use more comment lines to create a block of comments.

7. Statements

7.1. If

There are two types of if statements. When the if is only used to set a value for a variable, use the short if statement:

```
sValue = bValue ? “true” : “false”;
```

However, when a longer if statement is used, the next formatting must be used. The curly brackets (“{” and “}”) are mandatory, even when only one line of code is used inside if and else statement.

```
if (condition)
{
    // Code that should be executed when condition is met
}
else
{
    // Code that should be executed when condition is not met
}
```

When no else is needed, the else part can be removed from the formatting.

7.2. For

See example. The curly brackets (“{” and “}”) are mandatory, even when only one line of code is used inside the loop.

```
for (int i = 0; i < condition; i++)
{
    // Code that should be executed
}
```

7.3. While

See example. The curly brackets (“{” and “}”) are mandatory, even when only one line of code is used inside the loop.

```
while (condition)
{
    // Code that should be executed
}
```

7.4. Switch

A switch statement must be formatted like the following example:”

```
switch (iValue)
{
case 0:
    // Code that should be executed
    break;

case 1:
    // Code that should be executed
    break;

case 2:
    // Code that should be executed
    break;

default:
    // Code that should be executed
    break;
}
```

7.5. Try Catch

A try catch statement must be formatted as the example that follows. The curly brackets (“{” and “}”) are mandatory, even when only one line of code is used inside the try and catch blocks.

```
try
{
    // Code that can throw exception
}
catch (ExceptionType exceptionVariable)
{
    // Code that should be executed when exception is raised
}
```