*STANDARD*

# C/C++ Coding Convention

| | |
|---|---|
| Code | 09ae-HD/PM/HDCV/FSOFT |
| Version | 1/3 |
| Effective date | 10-May-2013 |

## RECORD OF CHANGE

*A - Added M - Modified D - Deleted

| Effective Date | Changed Items | A* M, D | Change Description | New Version |
|---|---|---|---|---|
| 10/12/2005 | | | | 1/2 |
| 26/04/2013 | | A | Add item 2.1.1 | 1/3 |
| | 3.1.1. Source file name | D | Remove "File names may only have 8 characters, with at most a three-character extension, so they can be accessed by DOS based-machines". | |
| | 3.1.3.1.Cconvention | A | change "W: integer (word)" to "W: unsigned integer (word)" | |
| | 3.3.2.Spages | D | Remove "The return statement of a function should be<br>written as if it were a function call. Example:return(lRetValue);" | |
| | 3.3.Format of Statements | A | Add new : "3.3.6. Use of return"; "3.3.7. Use of goto"; "3.3.8. Comparison with constant". | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

## TABLE OF CONTENTS

# 1. INTRODUCTION

## 1.1. Purpose

This document presents a compilation of the C/C++ coding standards. The aim is to provide the programmer with program templates that highlight the layout and structure of C and C++ source programs.

An attempt has been made to outline a style that is easily readable, amenable to documentation and acceptable to most programmers. These standards were adapted where necessary to meet the project reality.

Most of the details given here apply equally to C and C++ programs. Where details only apply to C++, this will be stated explicitly.

It is assumed that the reader has knowledge of both the C and C++ programming languages. Whatever programming style you use, be consistent, and use an ANSI C or C++ compiler.

## 1.2. Application scope

Fsoft projects

## 1.3. Related documents

| No. | Code | Name of documents |
|---|---|---|
| 1 | 04e-QT/PM/HDCV/FSOFT | Process_Coding |

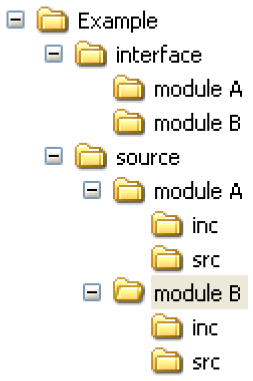
## 2. PROGRAM STRUCTURE

### 2.1. C and C++ Source Files

#### *2.1.1. Management file*

Source code should be organized as below structure in order to be readable, understandable and easy to maintain

Example:



- Interface/module A: contains header file of which are used by other modules
- Source/module A/inc : contains header file of which are only used in module A

#### *2.1.2. Source File Header*

The first part of a C/C++ program contains the source file header. The source file header details the creation/modification date, author, program name, and a description of the program. The header also contains a AMS copyright notice.

Use AMS as the author for all programs which are submitted to the customer, and use your personal name as the author only during development. This prevents clients that have our source code from contacting you directly for problems with code.

An example of a file header for a source code file is:

```
/*******************************************************************
********
 * ++
 * Author:          American Management Systems
 * Module Name   :   xxxxxxxx.cpp
 *
 * Description   :
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 *
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 *
 * Mod. History :  DD.MMM.YY - Full Name
 *                             File first created
 *                              CR# PROJECT
 *                              Description
 *
 * --

*****************************************************************
*****/
```

The extract marks ('* ++' & '* --') must be present in every header so that an utility can extract the headers for documentation purposes. Be sure to use the correct date format (DD.MM.YY) and the developers full name in all file headers.

### *2.1.3. File Organization*

This section describes how the header and source file should be organized. The first section outlines the header file (.h) layout. The second section outlines the souce file (.cpp) layout.

#### *a) Header File Organization*

The header files should be organized in the following manner:

#ifndef <file name descriptor>

#define <file name descriptor>

File header

Include files

Design categories (#defines, typedefs)

Global variables

Class definition 1

....

Class definition n

#endif

The #ifndef at the top of the file prevents multiple definitions. The <file name descriptor> should be the name of the file with an underscore substituted for the dot ('.'). For example, if the file is named PAGE_AP7.H, the file descriptor should be call PAGE_AP7_H.

#### *b) Source File Organization*

The source files should be organized in the following manner:

File header

Include files

Message map

Design categories (#defines, typedefs)

Global variables (these should be avoided as far as possible)

Method 1

....

Method n

Note that the use of global variables is discouraged. They should only be used where they appear in external libraries (e.g. errno).

Each file should begin with an include file list. System include files should come first, followed by include files from other packages, followed by developer-defined include files. Developer-defined include files should be enclosed by quotes, all other include files enclosed by < and >. Relative path names should be employed for the include file names, using compiler options (e.g. -I) to provide include file search paths where necessary.

After the include files, global definitions and variables (if any) should be placed in the following order:

Macros, i.e. #define statements local to the file

typedef statements local to the file

global variables which are exported (hopefully there will be none of these)

static global variables (hopefully there will be none of these)

static function prototypes

Preceding each method there should be a header. This header should be filled out at the time the method is written. The method header describes the name of the method, the function class, the method purpose, each parameter passed to the method, the date of the method, the author, the project the method is written for, and a brief description of the purpose. If a parameter is passed by reference, note whether the method modifies the parameter. If a value is returned, possible return values are documented. Any global variables accessed and/or modified should be noted. In general, these should be kept to a minimum. There is also a section for logging modifications, with the date of modification, corresponding author, the modification number, the project the modification is being made for, and a description of the modification.

### 2.1.4. File Modifications

When making modifications within the body of a function, it is important that the changes be documented within the code. This is done with the use of modification flagging. Before changes within the code, there is a beginning modification flag that contains the type of modification (PN/CR/SR) and the number associated with the modification.

example.

```
//++CR 10132
```

At the end of the changes, there is an ending modification flag that contains the same information as the beginning modification flag with "--" replacing "++".

example.

```
//--CR 10132
```

All functions have prototypes. Functions that have no return status should declare a void return type in the function definition and prototype declaration.

For example,

```
/*******************************************************************
************
 * ++
 * Method name      : Class::Method()
 * Description            :
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 *                     xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 *                     xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
 *
 * Parameters       :
 *
 * Global Variables
 *          Accessed    : none
 *          Modified    : none
 *
 * Return values    :
 *
 * Modifications    : DD.MMM.YY - Full Name
 *                                Function first created
 *                                CR/PN# - Project
 *                                Description
 *
 * --

*******************************************************************
**********/
```

Note the extract marks ('* ++') at the beginning and end of the header. Don't use the extract marks anywhere else. Be sure to use the correct date format (DD.MMM.YY) and full name in the modification log.

## 2.2. C/C++ Include Files

The first page of the include file is the include file header, which has the same format as the source file header. In general there are two types of information that may be listed in an include file: public information and private information. All public information must go into an include file, where it is accessible to the user. Information is considered public if it is required by the user for effective use of the functions with the C source code file. Typical public information is represented by return codes, exported function types, public structure and other type definitions etc.

Private information is represented by any structures, functions or constants that the user does not require to use the functions in the C source file. Whereas public information defines the interface, private information defines the implementation. The developer should feel free to alter the private implementation as long as it does not affect the public interface. Thus, private information should be kept in a separate include file or incorporated in the source file to which it refers.

System include files come first, followed by include files from other software packages, followed by user-defined include files. Nesting of include files is permitted.

Include files are organized similarly to source files. If any of the sections contained in the source files are contained in an include file they will follow the same layout as the C/C++ source file:

Include files

Design categories (#defines and typedefs)

Function prototypes

C++ class definitions

An include file should not include global variables or C/C++ function definitions (except for C++ inline functions). These should be placed in source modules.

Where C functions have been compiled with a compatible C compiler, and the include files have not been explicitly prepared for C++, this must be made known to the C++ compiler. C++ uses different names inside the compiler for its identifiers than C does. To tell the compiler that a C library is being used with C names, an alternate-linkage specification is required. Thus to use the memory functions (memcp() etc.), one must specify:

extern "C" { #include <memory.h> }  as the appropriate include file.

# 3. PROGRAMMING STYLE

This next series of sections will cover various programming topics. Most of the topics are focused on taking many different application programming styles and creating a standard that is to be used for the PROXIGATE GUI application.

## 3.1. Naming Conventions

This section will cover the naming schemes for variables, classes, and macros to create naming conventions standards for the application.

### *3.1.1. Source File Names*

Source and include file names should be lower case, with the use of an underline ("_") as a separator character. C/C++ source code files should end with one of the strings: .c, .C, .CPP, depending on the operating system and the compiler used. Include files have the extension ".h". If an include file is used principally or only with one source file, it should have the same base name.

### *3.1.2. Function Names*

These should be mixed upper and lower case, and begin with a capital letter. Individual words within the name begin with a capital letter, for example DoSomethingLikeThis(). Embedded acronyms should be left as capitals, e.g. DoThisASAP(). Underscores in these names should be used very sparingly, if at all.

Ensure that all C functions which are only local to a source module are made static, so as to reduce global name space pollution.

Each function has to do one task, so naming function should obey the following rules (as Microsoft often use):

-Function do a task should have names as: Do[Something] or Make[Something] (the first is a verb and then a noun).

-Function process an event: On[EventName], OnReceiveData, OnSendMailHeader,…

-Call-back function: [Event]Proc, ThreadProc, WinProc,…

### *3.1.3. Variable Naming Conventions*

The majority of Windows applications are written using the popular "Hungarian" variable-naming convention. This method of variable-naming reduces coding errors due to mismatched types, and will eliminate the need to repeatedly check the data type of a variable when analyzing source code.

#### 3.1.3.1. C Convention

Variable names are composed of two important parts:

- a type
- a qualifier (that belongs to a small set of standard qualifiers)

example:

`szPassword` the type is `sz` and the qualifier is `Password`

Types should be abbreviations or acronyms of a type description, preferably 2 to 3 characters long. The base types used in C are:

| **Base Types** | |
|---|---|
| `sz` | null-terminated string |
| `ch` | character |
| `w` | unsigned integer (word) |
| `l` | long |
| `f` | boolean (flag) |
| `x` | structure |
| `c` | C++ class |

Examples:

| | |
|---|---|
| `szFilename` | Filename null-terminated string; |
| `chYesLit` | Character representing a Yes; |
| `wNumUsers` | Number of users; |
| `lOffset` | Long offset; |
| `xRect` | Structure defining a rectangle; |
| `pxRect` | Pointer to a rectangle structure; |
| `hWindow` | Handle to a window. |
| `cPerson` | Class person... |

All variables and constants are simply named after their type, followed by a qualifier to make the name unique in its context. Variables may also have type prefixes which help identify how the variable is used.

**Type Prefixes**

| | |
|---|---|
| G | A global variable |
| R g T 2 | An Array of elements of type T2, indexed by numbers |
| I | An index into an array of elements |
| C | Counter of instances of a given type |
| P | A pointer |
| H | A handle (point to pointer) |
| M | A member variable (of an object) |

Examples:

| | |
|---|---|
| `wAge` | Numeric variable (word) representing somebody's age… |
| `gwAge` | Global numeric variable (word) representing somebody's age… |
| `rgwAge` | Range (array) of words representing ages… |
| `iAge` | Index into rgwAge… |
| `mwAge` | wAge is a member of an object... |

Qualifiers distinguish values of identical type or document important properties. Contrary to types, full words can be used as qualifiers. The syntactic rules for qualifiers are described below:

- Contrary to types, full words can be used as qualifiers.
- Multiple qualifiers can be combined if necessary.
- Individual words are capitalized.
- A qualifier can be a number or might not even exist.

Uniform criteria must be defined for choosing qualifiers.

For boolean variables (f), the qualifier is the condition under which the variable is true.

Example:

`fOpen` Boolean variable relating the state of a file

**Standard Qualifiers**

| | |
|---|---|
| `Temp (T)` | A temporary. Typically used in loop variables or other temporary functions. |
| `Sav` | A temporary, which will have its value stored. |
| `Prev` | A value worth keeping, just behind a current value for an iteration. |
| `Cur` | Current value in an iteration. |
| `Next` | Next value in an iteration. |
| `Dest, Src` | Destination and origin in a consumer/producer relationship |
| `Nil` | A special and illegal value, that is distinguishable from all legal values |
| `1, 2` | Numbers can distinguish values; like in arguments to a function |
| `Min` | Smallest legal index. Typically defined to be 0 |
| `Mac` | Current maximum. The top of a stack, current upper limit for legal indexes. Also the number of elements in an array when `Min` is 0. |
| `Max` | Limit allocated to a stack. Max is always greater or equal to `Mac`. |
| `First` | First element in a closed interval `[First, Last]` |
| `Last` | Last element in a closed interval `[First, Last]` |
| `Lim` | Strict upper limit in an open interval `[First, Lim[` |

### 3.1.3.2. C++ Convention

The principles are as C convention. Each variables name has 3 parts:

**[The type of variable]_[The data type prefix][Name]**

-The first(the type of variable) is prefix to indicate it is a global variable, static variable, class member variable or temporary variable.

| | |
|---|---|
| `g_` | A global variable |
| `s_` | A static variable |

| | |
|---|---|
| m_ | A class member variable |
| | None is temporary variable. Eg: TCHAR lpsz[MAX_PATH] = _T(“”); |

-The second indicates the data type. It can obeyed the recommendation in the following table:

| **Base Types** | |
|---|---|
| lpsz | null-terminated string |
| wsz | UNICODE string |
| w | WORD |
| l | long |
| b | BOOL/bool |
| p | pointer |
| c | char/TCHAR |
| dw | DWORD |
| n | integer/short/long |
| u | unsigned int/short/long/char |
| arr | Array |
| map | Map |
| lst | List |
| h | HANDLE |
| rc | RECT/CRect |

And here is an example of some variable:

```
int s_nSessionIndex;        //Static variable

class X

{

  private:

           TCHAR m_lpszName[MAX_PATH];

}
```

#### *3.1.4. Class and Instance Names (only C++)*

Class names follow the same rules as function names, whereas instance names follow those of variable names. It is recommended that instances which occur only once in a program should be prefixed with the string "the", e.g. "theApplication".

#### *3.1.5. Preprocessor Macros, typedefs and C++ constants*

Preprocessor macros should be named using uppercase letters, using an underline character to separate individual words within the name. Typedefs and C++ constants should follow the same rules as for function names. In C++, use **const** to define constants values as far as possible, rather than **#define**.

Constants in a group should be group into an enum.

Should:

```
enum OBJECTTYPE

{

  OTCAR = 1,

  OTPLANE = 2

};
```

Should not:

```
#define OTCAR 1

#define OTPLANE 2
```

### 3.2. Format of Comments

In ANSI C programs, all comments should be marked with the delimiters /* and */. The double slash comment, //, is non-ANSI and should only be used in C++ programs.

Comment blocks should be used to separate and describe following sections of code. These comments should be indented to the level of the code and use normal punctuation rules.

Comment lines should be used on the same line as the statement or declaration to describe its use or operation. They should start with a small letter, as they are generally not complete sentences. In C++, a one line comment is preceded with "//".

example:

```
#define ACT_FILE  "ACT_FILE"  // ini file section header
```

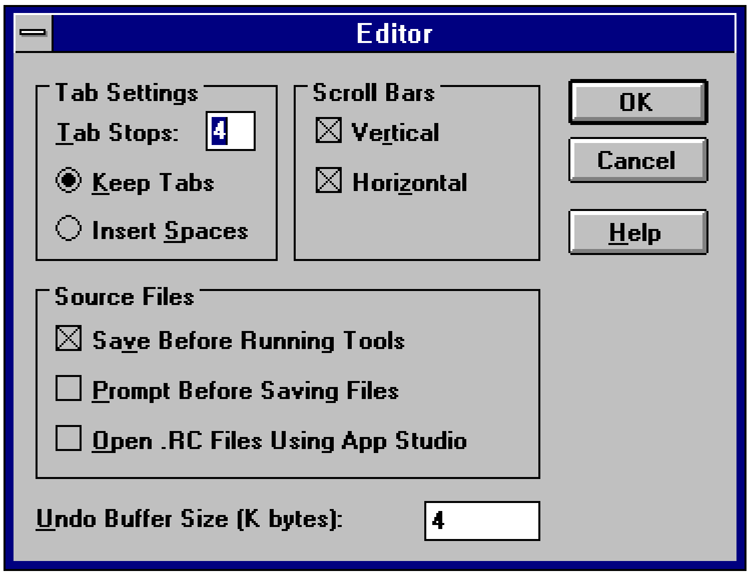
Code that is no longer in use should be removed from the source. Occasionally, the programmer may wish to retain old during testing. This old commented code must be removed prior to the migration from the developers area. Sections of old code that is removed must be marked with a modification flag as described in section 2.1.3.

## 3.3. Format of Statements

The following sections describe the formatting of statements within the PROXIGATE GUI application.

### *3.3.1. Indentation*

Indentation across the screen should use a width of 4 spaces. This can be achieved using TAB characters of 4 spaces width or 4 separate spaces. Within the Visual C++editor, the tab setting should be set and confirmed by the developer prior to code modification. Select the Editor dialog from the Options menu.


Editor

Tab Settings
Tab Stops: 4
● Keep Tabs
○ Insert Spaces

Scroll Bars
☒ Vertical
☒ Horizontal

Source Files
☒ Save Before Running Tools
☐ Prompt Before Saving Files
☐ Open .RC Files Using App Studio

Undo Buffer Size (K bytes): 4

Blank lines should be used to separate unrelated sections of code and where it will aid in clarity and make the code more readable. They should not be inserted too liberally.

The following should start at the first column in a source file or include file:

**#include** file statements

Macros

global typedefs

variables - only source file

function prototypes (unless occurring within a class definition in C++)

class name declarations;

class definitions

function definitions (unless as **inline** within a class definition in C++) - only source file

For example:

```
#include <iostream.h>

#define MY_NAME        "John"
typedef long  lMyLong;

int wGlobalWord;                // global variable
int wGlobalFunc(int w);         // global function prototype
static int LocalFunc(int w);    // local function prototype

class SomeClass;                 // class name declaration

class MyClass                   // class definition
{
private:
  int wMe, wMyself, w;
public:
  MyClass();              // constructor
  void showMyself {       // inline function
        cout << "me: " << iMe << "myself: " << iMyself << "i: " <<
        i << endl;
}
};

MyClass::MyClass ()
{
    wMe = 1;
    wMyself = 2;
    w = 3;
}
```

### 3.3.2. Spaces

Spaces should be used to separate all operators from their operands. Conditional keywords, such as **if**, **while**, **switch**, and **for** should be separated from their succeeding bracket by a space. There should *not* be a space in a function call between the function name and its parameter list, enclosed in round brackets. Spaces should be used after a semicolon and a comma, as is used in normal writing. Spaces should not in general be used to separate function arguments from their enclosing round brackets or in other expressions enclosed by round brackets, unless very complicated expressions are used, where they might be needed for clarity.

### 3.3.3. Lines Longer than 80 Characters

Lines should be split up if they are longer than 80 characters. This aids the use of editors and for printing purposes. This is best done using an indented structure.

With source code line has lots of expression, they should be divided to each expression in one line. Example:

```
if ((lNumberEntries < MAXNUM_ENTRIES) &&
    (lStatus == APPROVED) &&
    (lSituation == NO_EMERGENCY_PRESENT) &&
    lThisIsAllowed)
    GoAhead();
```

### 3.3.4. Statements and Curly Brackets

The format of statements seems to be a deeply personal issue to many programmers. In some cases one of two standard layouts should be adopted, both of which occur frequently in literature on C programming. In the case of the *if .. else* statement, the version in the below example should be used.  This *if..else* format is used for the purposes of standardization, clarity, and readability.

| **if..else** |
|---|
| if (condition)<br> {<br>    statement1;<br>    statement2<br> }<br>else<br> {<br>    statement3;<br>    statement4;<br> } |

In the next example, it should be noted that the second variant takes up considerably more space on paper. The **else** in the first variant must be started on a new line. If nested conditions are used, insert extra curly brackets to ensure clarity. With an **if** and a single line of code, both these constructions are allowed:

| **Version 1** | **Version 2** |
|---|---|
| If (condition)<br>    statement; | if (condition)<br> {<br>    statement;<br> } |

Case statements can be written in one of two ways:

| **Version 1** | **Version 2** |
|---|---|
| Switch (condition) { | switch (condition) |
| Case 1: | { |
| statement1; | case 1: |
| break; | statement1; |
| case 2: | break; |
| statement2; | case 2: |
| break; | statement2; |
| default: | break; |
| statement3; | default: |
| break; | statement3; |
| } | break; |
|  | } |

The first variant has the advantage of occupying one less indent, yet there is a slight disadvantage in the readability. Where a case condition contains a large number of statements, it is advisable to use curly brackets to contain them.

For example:

```
switch (condition)
{
      case 1:
      {
            statement1;
            ..
            statementn;
      }
      case 2:
      {
            statement1;
            ...
      }
}
```

This allows checking of brackets for indentation purposes etc. In general there should be at most one statement per line.

### 3.3.5. Function Declarations (Prototypes) and Function Definitions

Should they be placed in a source file, as module-local functions or external link declarations, function declarations (function prototypes) are always to be declared together near the top of the file.

Function definitions should be laid out as shown in the following example:

```
int ClassName::SomeFunction  (int iParam1, char *pzParam , long
lParam3 )
{
 int iFirstVar;
 char cSecondVar = 'c';

 iFirstVar = 1;
 return(iFirstVar);
}
```

Space permitting, the function definition argument list should be written on a single line. When programming in C,  there should be a blank line separating the last variable declaration from the first statement in the function. There should be a space following the function base name to help to differentiate the function definition from the function declaration (function prototype), especially when they are both in the same file.

Empty function argument lists (with variable no. of arguments)

There is a significant difference between C and C++ for functions with empty argument lists. In C, the function declaration

```
 int Func2();
```

means "a function with any number and type of argument." This prevents type-checking, so in C++ it means "a function with no arguments."

### 3.3.6. Use of return

Only 1 return statement should be used in a function. Do not return at the middle, but always make it the last statement of the function.

### *3.3.7. Use of goto*

In general, use of ***goto*** is considered to be a bad habit, but proper use of it can make the code more robust, as well as easier for error handling/maintenance. It should be used in combination with rule 3.3.6 above.

- Use only 1 ***label*** in a function, this is the place for handling errors (display message, trace logging, free allocated memory).
- In case of error, do not return immediately but ***goto*** the ***label*** instead.

Example:

```
void DoSomething(char* param)
{

   if (NULL == param)
   {
      // error handling
      goto End_of_function;
   }

   char* pPixels = malloc(1024*1024);
   if (NULL == pPixels)
   {
      // error handling
      goto End_of_function;
   }

   // other processing
   // if something WRONG occurred here, and return imemediately, pPixels would be
leaked.

End_of_function:
   if (pPixels)
   {
        free(pPixels);
    }
}
```

In the example above, whether error has occurred or not, allocated memory always freed properly.

### *3.3.8. Comparison with constant*

Should place constant on the LEFT when writing comparison with ***if*** statement. In case only 1 ‘=’ is written, the compiler generate the error immediately.

For example:

```
If (PI == x) {};                    // PI = 3.14156
```

## 4. VARIABLE ARGUMENT LISTS (WITH VARIABLE NO. OF ARGUMENTS)

The ANSI C library variant should always be used here (see include file varargs.h).

## 5. USE OF C++ FUNCTIONS IN C FUNCTION ARGUMENT LISTS

Normally, a C++ member function contains a hidden additional initial parameter, which is not seen by the developer. This parameter points to the object instance, thus allowing this instance to be accessed easily with the function itself. For example, when such a member function is used as a C callback function which has specific arguments, there will be one argument too many. This problem can be alleviated by using a static member function as the callback function.

Class Name Declarations and Class Definitions (only C++)

The following is an example of a class name declaration:

example:

```
  class CSalesman;
```

This can be placed in files similarly to function prototypes in C.

A class definition on the other hand should have the following format:

```
class CSalesman
{
private:
  char *pszName;
  char *pszDepartment;
  long lSalary;
protected:
  void SetSalary(long lSalary);
public:
  Salesman(char *pszName, char *pszDepartment);
  char chGetName();
  char chGetDept();
  long lGetSalary();
}
```

The **private** keyword should always be stated explicitly.

## 6. PORTABILITY

Attempt to avoid the use of language constructs particular to the compiler, as these can compromise the potential portability of the program. Always use the ANSI compiler switch where the use of an ANSI compiler is optional.

One tip to do this is as Microsoft tip by using pre-compile processor:

```
#ifdef _UNICODE||UNICODE
        #define STRLEN wcslen
#else
        #define STRLEN strlen
#endif
```

Or the sample below to make the code running on Linux without changing the code.

```
#ifdef _WIN32||WIN32
        #define CMyMap CMapObToString
#else
        #define CMyMap FSOFTMapObToString
#endif
```

### 6.1. Machine Word Length.

The size of the data type **int** depends on the machine word length and is therefore different for different machines. Do not use **int** unless you *must* in order to interface with a library or third party package function. Should there be any doubts as to the result of an integer operation then use the data type **long**.

It is not in general guaranteed that **int** occupies a complete machine word. **C** demands only that the value of **short** is contained in the value of **int** and that this is contained in **long**.

The word length can influence constants, e.g. bit masks:

example:

```
#define MASK 0177770    // incorrect
int iX;
iX &= MASK;
```

The 3 furthermost right bits of x are only correctly cleared if **int** is 16 bits long. If **int** has more than 16 bits then the **leftmost** bits of x will also be cleared. One can avoid problems by using the following #define statement:

example:

```
#define MASK (~07)
int iX;
iX&= MASK;
```

**typedef** declarations should be used to hide machine-dependent data types.

Check shift operations carefully.

## 6.2. Use of External References

Keep all references to **external** variables in a special source file, to improve code portability and maintainability. Ensure that all modules have access to this file (using **#include**).

## 6.3. Use Text Input and Output for Binary Information

As far as possible, do not store data such as data dictionaries in binary form, although it may be used as such in the program. Use conversion routines to store it in a human-readable form, which is easily accessible through a text editor. This simplifies program testing.

## 6.4. Errors

Every tested condition or system call should have an error trap in case of failure. Each function should set and return a status flag to its calling program so any errors can be handled correctly or the program terminated cleanly.

## 6.5. Compiling

Final versions of programs should compile without any warnings with the compiler warning level turned up high.

# 7. LOGGING/DEBUGGING CONVENTION

This section will describe some guidelines to help developers in debugging their applications.

Logging is one of simple but effective way to debug the application. Define a framework to log information about current point of execution will help developers to trace the flow and isolate problems easily. Of course, this technique also has a major drawback is it will impact the performance of the application so it should be excluded when performance test is required.

Follow these steps to enable logging to your application

## 7.1. Define functions to write log

To implement logging, simply define a global function to write logging information to file or system debugger as following sample:

Log(long lFlag, char* lpszModule, char* lpszFormat, ...);

In this function, developers are freely to format the content to be sent to debugger. Following is a sample:

```
2002/09/17 17:30:18 [DEBUG] (service) Start service
2002/09/17 17:30:18 [DEBUG] (service) Open from file [C:\FOO.TXT]
2002/09/17 17:30:18 [ERROR] (service) Read from file failed
[C:\FOO.TXT]
```

In order to filter log for developers or users, there are two simple methods:

## 7.2. Use #define and #ifdef keywords. With this method, in the main source or include file, use #define keyword to define a symbol to turn on the debugging and use #ifdef keyword to check this symbol is defined or not before writing log to file.

## 7.3. Use flag which is settable by user. Inside the Log function, developers will check the input flag against the setting flag before writing to file.

Both methods can be used simultaneously to provide full features for the application.

Note:

- For Windows platform, function OutputDebugString() can be used to send to system debugger. Consult Windows MSDN Library for more detail of this function.
- For Linux or U*X platform, function vsyslog() can be used to send to system log. Consult vsyslog man page for more detail of this function.

## 7.4. Write to logger inside each function

In order to easily debug and troubleshoot the system in runtime, log writing must ensure: provide enough but not excessive information. Following guidelines should be taken care:

- For complicate or very important functions, should write log at the beginning and the end of the functions, and all parameters should be written at the beginning of function, for example:
  - Log(DEBUG, “XMLParser”, “LoadXML(): started, szFileName = [%s]”, szFileName) at when starting
  - Log(DEBUG, “XMLParser”, “LoadXML(): finished”) at every exit of function.
- For a simple function, one log should be written with a filter flag so that for deployment, we can omit this log by using configuration.
- Before **switch** statement, should have a log to provide the first value for evaluation. Inside each case, if necessary, a log can be written.
- For important evaluation by using **if-else**, **do-while**, **for** a log should be used.
- In a complex function, there are some checkpoints should be defined and for each checkpoint, a log can be used for this notice. For example:
  - Log(DEBUG, “XMLParser”, “LoadXML(): Load content to document”)
  - Log(DEBUG, “XMLParser”, “LoadXML(): Parse successfully, iterate through all nodes”)
- Log writing can be used to monitor time for a certain task or process (in real-time application), to monitor usage memory (in heavy load system), to evaluate very important variables (in application/system which is very difficult to debug).

| Approver | Reviewer | Creator |
|---|---|---|
| | | |
| **Nguyen Quang Hoa** | **Dang Van Thanh** | **Le Hong Viet**<br>**Do Trung Hieu** |