Coding Guidelines in OSSP Lab

- To understand why coding guidelines are required
- To understand Coding guidelines and apply in OSSP Project.
- To learn modifying the existing code as per coding Guidelines
- To learn to do the code review

Taj Alam

Facts of life…

- Your code will change
- No matter how clear and simple you think your code is, it is complete nonsense to almost everyone else
- If you haven't tested your code then it's probably wrong
- *I think it is inevitable that people program poorly. Training will not substantially help matters. We have to learn to live with it. (Alan Perlis)*
- *A great lathe operator commands several times the wage of an average lathe operator, but a great writer of software code is worth 10,000 times the price of an average software writer. (Bill Gates)*
- *Computer science education cannot make anybody an expert programmer any more than studying brushes and pigment can make somebody an expert painter. (Eric Raymond)*

Multi File Program

- Typically the code is organized into multiple files
  - Source Files (Main and Functions)
  - Include Files (Header Files)
  - Object Files
  - Bin (Executable)
  - File (Extra doc or text files)
  - Make (Makefile)

  e.g. Please find the example folder in help.

Multi File Program

What do we learn?

- How to organize code multiple files.
- Use of file headers and function headers.
- Use of comments.
- Use of meaningful variable and function names .

Why Coding Conventions

- Improve readability
- Improve understandability
- Improve maintainability

Coding Standards

- Covers
  - Presentation style
  - Naming conventions
  - Language usage

C Source File Structure

- Source file header (example in subsequent slides)
- Compiler options (e.g. `#pragma`)
- Preprocessor statements (that relate to include files) e.g. (`#if _WINDOWS_ #include “win.h” #endif`)
- Include files
- Other preprocessor statements
- External declarations – (should generally be in a header file)
- Initialization of global data
- Local functions or procedures definitions
- Main function

Source File Header

- Every source file starts with comment header
- Comment header includes
  - The name of the file
  - A detailed description of the file contents including what is accomplished
  - Revision history (optional). This information may already appear in your CM system.

File Header

```
/*****************************************************************
    *****************
 **
 **    FILENAME       :    ss_esl_refer_ext.c
 **
 **    DESCRIPTION    :    This file defines the functions which send
 **                        external messages for Refer.
 **
 **    Revision History    :
 **    DATE       NAME                    REFERENCE          REASON
 **    ------------------------------------------------------
 **    11 Sept 2002  Mayank Rastogi   SPR 1204New code for RY feature
**    22 Oct 2002   Narendra Shukla    SPR 1211Bug fix for loop check in
**                                                     as_sff_foo()
**
 **    Copyright © 2006   Inc  All Rights Reserved
 **

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

Include File Structure

- Include file header
- Embedded include file references
- Defines, constants, literals, macros
- Type definitions
- External variable declaration
- External function declaration

Example (Header File)

Technique for preventing multiple inclusion of include file

```
#ifndef __SS_ESL_REFER_MACRO_H
#define __SS_ESL_REFER_MACRO_H
/******************** STANDARD INCLUDE FILES *************************/

/******************** USER INCLUDE FILES *****************************/
#include "ss_esl_traces.h"
#include "ss_esl_macros.h"
/******************** MACROS***************************************/
#define ESL_REFER_INVALID_ARG 0

#ifdef ESL_TRACE_ENABLED
#define ESL_REFER_TRACE(trc_id, no_int_args, arg1, arg2, arg3, arg4) {
/*Write definition here*/}
#else
#define ESL_REFER_TRACE(trc_id, no_int_args, arg1, arg2, arg3, arg4)
#endif /* end of ifdef ESL_TRACE_ENABLED */

#endif /* End of ifndef __SS_ESL_REFER_MACRO_H */
```

Function Header

- A comment header should appear preceding every function definition
- Function comment header includes
  - Function name
  - General description of the contents and purpose of the function
  - Optional notes describing special considerations, warnings, unusual techniques, etc
  - Description of the returns from this function
  - Comments relating to the parameters are associated with the actual parameter definition

Function Header

```
/*****************************************************
**
**FUNCTION NAME   :    ss_esl_refer_arm
**DESCRIPTION     :    In this function sends a call event
**                    request to SF for arming the
**                    specified event on the specified
**                    leg.
**
**RETURNS         :   ESL_SUCCESS, and in case of any
**                    error ESL_FAILURE with the
**                    corresponding ecode.
**
*******************************************************/
```

<Function definition here>

<Parameter definitions here including comments>

Comment Style

- Block comments
    - Precede statement or groups of statements referred
    - Should begin at same indentation level as code
- Single line comments
    - Placed to the right of the statement, if they fit on the same line
- Do NOT write redundant comments

Example (Comments)

```
.
.
.
/*
 * Special case: Check if name is the name of the caller.
 */
if (strcmp(runningTask->name, name) == 0)
{
   return (int) runningTask;
}

for (i = 0; i < 256; i++)
{
   /*
    * Check ready queue.
    */
   for (tcb = (WIND_TCB *) lstFirst(&readyQ[i]);
        tcb != NULL;
        tcb = (WIND_TCB *) lstNext((NODE *) tcb))
   {
      if (strcmp(tcb->name, name) == 0)
      {
             return (int) tcb;
      }
   }
                 /*
    * Check pending queue.
    */                for (tcb = (WIND_TCB *) lstFirst(&pendQ[i]);
        tcb != NULL;
        tcb = (WIND_TCB *) lstNext((NODE *) tcb))
   {
      if (strcmp(tcb->name, name) == 0)
      {
             return (int) tcb;
      }
   }
}
.
.
.
.
```

Figure . Comment Style Example (C Language format)

Indentation

- Global variable declarations begin with a storage class designation in the first column
- Local variable declarations must be indented to the same level as the code or block in which they are declared
- Function or procedure parameters should be presented one per line, each indented at least one indentation level
- The block or body of a function or procedure must be indented one indentation level
- Indentation levels must be consistently used to indicate the depth of a statement within constructs such as conditions, loops, switches, and functions

Example (Indentation)

Global & local variable indentation

```
extern int   employeeage;
static char middleinitial;
char * abc(int data)
{
     int iloop;

     for (iloop = 0; iloop < 10; iloop++)
     {
          …
     }}
```

Example (Indentation)

```
int taskvarget(       int tid,              /* task identifier */
       int *pvar)     /* pointer to variable */
{
   register TASK_VAR *tvar;           /* local variable definitions
   */
       if (ERROR == taskidverify(tid))
   {
       return ERROR;  /* no such task */
   }
   for (tvar = ((TCB *) tid)->ptask; tvar != NULL; tvar = tvar-
   >next)
   {
       if (tvar->address == pvar)
       {
              return tvar->value;
       }
   }
   return ERROR;             /* no such variable */
}
```

Horizontal Spacing

Use horizontal spaces for

- Before and after binary operator.

```
xyz + abc
```

- After a keyword

```
for (..) or while (..) or if (..)
```

- After a comma or semicolon.

```
for (i = 0; i < 10; i++)
```

Horizontal Spacing

- **DO NOT** use spaces
  - Between a function name and opening parenthesis : `getId(int x)`
  - After an opening parenthesis : `if (xremote == 0)`
    ```
    return (1);
    ```
  - Before a closing parenthesis: `if (xremote == 0)`
  - Before or after an opening bracket. `int array[10]`
  - Between unary operator and it’s operand. `i++`
  - Before or after a structure reference operator. `abc.xyz = 1`

Vertical Spacing

- A logical paragraph of code should be preceded by block comments, describing the purpose of block, with a blank line before the comment block and NO vertical spacing between lines of code within the paragraph

Example (Vertical Spacing)

Flow control structure with 1 statement

```
// Inline - no!
   while ( /* something */ ) {i++;}

/* Write Block comments here to describe
   the
* following piece of code
* Block - better! */
   while ( /* something */ )
   {
       i++;
   }
```

Ex (Function Definition)

Declaring formal arguments for a function (BAD)

```
int mycomplicatedfunction(
 unsigned unsignedvalue, int
 intvalue, char*
 charpointervalue, int*
 intpointervalue, myclass*
 myclasspointervalue, unsigned*
 unsignedpointervalue );
```

Ex (Function Definition)

Declaring formal arguments for a function (BETTER)

```
int myComplicatedFunction( unsigned unsignedvalue,
                              int intvalue,
                              char* charpointervalue,
                              int* intpointervalue,
                       myclass* myclasspointervalue,
                         unsigned* unsignedpointervalue);
```

Example (Layout – Both Horizontal and Vertical Spacing)

Left parentheses directly after function name

```
 void foo ();   // no!!
 void foo();    // better
```

Declaring many variables in same statement

```
 // Not recommended
   char* i,j;   // i is declared pointer to
   char                 // while j is char
//Better Way
   char* i;
   char* j;
```

Sizing

- NO specific limit for size of function
- Avoid long and complex functions
- Ideal size 60 to 120 lines

Advantage Of Small Function

- Reduces complexity
- Improves readability and hence testability
- Improves understandability
- Improves maintainability
- If error discovered at end of a long function, it is difficult for the function to clean up & "undo" as much as possible before reporting error to calling function

Naming Conventions

- Module prefix should be between two to four characters in length.
- Use names that indicate the content of the variables.
- Enumeration values should be consistently, either in upper case or lower case.
- Use underscore or mixed case characters naming, in a consistent manner. `sz_user_name` OR `szUserName`
- Use all upper case for macros and `#define` constants.
- Declare local variable for each distinct purpose, instead of using one over and over again. e.g. Do not use variable “i" for all the loops.

```
while(i < MAX_ARRAY_SIZE){array specific code,
i++;}
while(i < MAX_STR_SIZE) {string specific code,
i++;}
```

Problem: the variable i was not re-initialized before second while

Naming Conventions

- Do NOT use names that differ only by the case of characters. e.g. `SzName` and `szName`
- Do NOT start names with “_” or “__”.
- Do NOT rename operations using macros. Example

```
#define EQ ==
```

(avoid)

The reason why people like to define EQ is to avoid coding mistakes like **`if(A=B)`**. But there are better ways to detect this problem. Compile the code at highest warning level, the compiler gives a warning if we write `if(A=B)` instead of `if(A == B)`

Example (Naming)

Choice of names

```
int groupid;                  // instead of
 grpid
int namelength;                   // instead of
 namln
printerstatus resetprinter;  // instead of
 rstprt
```

Ambiguous names

```
void termprocess();      // terminate
 process or
                     // Terminal process?
```

Example (Naming)

Names with numeric characters can cause errors which are difficult to locate

```
int I0 = 13;          // names with
 digits can be
int IO = I0;          // difficult
 to read
```

Ex- Intuitive Naming (BAD)

Example, a boolean variable '`noZbuffer`'

```
noZbuffer == TRUE     //Z buffer not available
noZbuffer == FALSE    //Z buffer is available
```

To execute code following fragment....

```
if ( ! noZbuffer )
        { ...Z buffer code here... }
```

If there's NOT the absence of a `Zbuffer`, then there IS a `Zbuffer`, so this code runs when a `Zbuffer` is available

Ex- Intuitive Naming (BETTER)

- Instead, to name the boolean `'zBuffPresent'`,

```
if ( zBuffPresent )
        { ...Z buffer code here... }
```

Or

```
if ( ! zBuffPresent )
        { ...Non-z code here... }
```

- This naming mechanism holds for preprocessor symbols and configuration parameters

Language Usage

- Declarations and constants
- Expressions and statements
- Functions and files

Declarations & Constants

- Global data should NOT be used to pass parameters
- All functions must be explicitly declared to return some type which may be void

Example (C Function)

- Functions which return no value should be specified as having the return type void

```
void Strangefunction(char* before,
 char* after )
  {
 /*Do something here*/
  }
```

Declarations & Constants

- `NULL` must only be used with pointers, `0` with integers, and `'\0'` with ASCII characters
- Names should NOT be redefined in inner blocks
- Pointers to un-typed objects must be of type `void *`
- If registers are used, declare registers in order of importance to insure compiler assigns the most important ones if it runs out of **`register`**`s` to use.
- When defining macros, each of the parameters in the replacement text must be surrounded by parentheses, as well as the entire replacement text.

Example On Macros

- `#define square(x) (x * x)` (BAD)

- `int b= SQUARE(2+3);`
- `//b = (2+3*2+3) = 11`

- `#define square(x) ((x) * (x))` (GOOD)

Declarations & Constants

- External variables should only be defined once.
- Constants are to be defined using `const` or `enum`. Do NOT use `#define`
- All variables which are read before written must be initialized
- In general, minimize the use of global data
- Use unions only if you cannot avoid them

Declarations & Constants

- Array subscripts are usually considered "magic" numbers and should be defined as constants
- Enumerators should be defined with `typedef` statement
- Any user defined type should be defined with `typedef`
- Pointer to pointers should be avoided wherever possible

Expressions & Statements

- Check every system and library call for error returns unless you wish to ignore errors
- Parentheses should **NOT** be used to force evaluation order. When evaluation order is important, one can introduce an extra temporary variable instead

```
/*
 * instead of the following:
 */
anum * (bnum / cnum);

/*
 * do the following
 */
temp  = bnum / cnum;
eval = anum * temp;
```

Expressions & Statements

- The use of **`goto`** is not explicitly forbidden. Its use must be looked upon as an act of desperation to avoid major function/procedure redesign
- **`goto`** must NOT be used to branch:
    1. into an **`if`** statement or within **`if`** statement's **`then`** or **`else`** segments,
    2. into an iteration statement, **`switch`** **or** **`case`** statement body,
    3. into a compound statement block.

Example: use of **`goto`**

```
for (...)
        {
                    ...
                    if  (disaster)
                        goto error;
         }
    ...
error:
       /* clean up the mess you've gotten yourself
into!! */
```

Expressions & Statements

- If a sub-expression changes the value of a variable, then that variable may not appear anywhere else within the expression, except where explicit evaluation order is guaranteed

EXAMPLE:

```
/*
 *below may be evaluated as either alist[1] = 1  or
alist[2] = 1
 */
int index = 1;
alist[index] = index++;
```

- Sub-expressions which have side-effects (++ and --) may NOT be used in logical expressions.

Expressions & Statements

- Sizeof rules:
  - Use `sizeof` instead of a constant to represent the size of an object
  - `sizeof` should be used for array and structure size definition
  - `sizeof` is a operator and not function so there is no overhead in using it
- Do NOT use the `<<` and `>>` operators to perform multiplication and division only use them for bit wise operation

Expressions & Statements

- Use of nested conditional expressions, makes programs harder to read and should be avoided. e.g. `? : ? :`
- Use bit fields only when necessary to achieve correct bit alignments

Functions and Files

- All external functions must have prototypes. The prototype is declared in a header file accessible to other functions wishing to call that function
- Don't use absolute pathnames

  (ex: `#include </project/include/abcd.h>`) for include files
- Included files may include other files. To avoid "double loading" of include files, put protections in front to detect an already loaded include file

```
#ifndef  SAMPLE
#define SAMPLE
/*
 * put some include file stuff in here that may
be included more once
*/
...
#endif
```

Functions and Files

- If your C function takes no arguments, use the word void as the argument
- The names of formal arguments to functions are to be specified and are to be the same both in the function declaration and in the function definition
- Do NOT use the preprocessor directive `#define` to define a macro to obtain more efficient code; instead use inline functions

Functions and Files

- Structures should not be passed to functions because the whole structure is pushed onto stack. Use pointers or references instead
- Place machine-dependent code in a special file so that it may be easily located when porting code from one machine to another
- Use constant references (`const *`) instead of call-by-value for user defined types

Best Practices

- Do not hardcode values

  Values should be defined as constants.

  ```
  const int SZ_USER_NAME_LEN = 21;
  ```

  Also use

  ```
  const int INVALID_INDEX = -1; // invalid
    array index
  ```

  ```
  int i_index = INVALID_INDEX;
  ```

  Instead of

  ```
  int i_index = -1; // initialized to
    invalid value
  ```

Best Practices

- For nested loops or conditional statements, mark end of loop for easy identification

```
while (a < b)
{ ...
   while (c > d)
  {
  } // End while (c > d)
} // End while (a < b)
```

Thank You