NATIONAL WEATHER SERVICE
OFFICE of HYDROLOGIC DEVELOPMENT

Science Infusion Software Engineering Process Group (SISEPG)

C Programming Standards and Guidelines

Version 2.8

Revision History

Date	Version	Description
12/18/2006	2.6	Initial Version
04/23/2007	2.7	Section 2.4.14 Do Not Hardcode Values was updated and expanded to provide further clarity.
08/13/2007	2.8	Updated to allow programmers to use “camel case” when naming variables and functions. Gave Pointers and Dynamic Memory section its own section (3.5).

Table of Contents

1. INTRODUCTION

The Office of Hydrologic Development (OHD) produces software which NWS Weather Forecast Offices (WFOs) and River Forecast Centers (RFCs) use to generate hydrologic forecasts and warnings for rivers and streams across the country. OHD also develops and maintains software which runs centrally to acquire and distribute critical data to the WFOs and the RFCs. Just like many other organizations, software has become a critical component supporting the operations of these forecast offices. Because software plays such an important role, it is essential that it be well written and maintained.

The OHD Science Infusion Software Engineering Process Group (SISEPG) is developing standards and guidelines to ensure that programmers follow good, widely accepted software development practices when coding. It is believed that this will lead to well written and better structured programs.

Well-written software offers many advantages. It should contain fewer bugs and run more efficiently than poorly written programs. It also makes it easier for a programmer who was not involved in the development of the software to learn how it works.

Software has a lifecycle. A large part of its lifecycle revolves around maintenance. Software may exist for many years, even decades. Long after the original programmer has moved on, the software will require maintenance in the form of bug fixes and enhancements. The time spent doing this and hence the cost is greatly reduced when the code is developed and maintained according to software standards.

The C programming language is a popular and powerful application development language. The C programmer is given access to memory and system routines which if used improperly can result in unreliable programs which waste system resources and CPU cycles.

This document will present standards and guidelines for the C Programming Language. The standards are programming techniques which OHD programmers are expected to adhere to. Their use will be enforced through peer reviews and code walkthroughs. The programming guidelines are good programming practices which developers are encouraged to adopt.

The developer should read the OHD General Software Development Standards and Guidelines document to become familiar with the standards and guidelines deemed by the SISEPG to be applicable to all programming languages.

It is important to note that standards are not fixed, but will evolve over time. Developers are encouraged to provide feedback to the SISEPG (sisepg@gateway2.nws.noaa.gov). Also each project area may derive its own standards and guidelines if special requirements are desired.

2. STANDARDS

C programming standards are a set of C programming rules which must be applied by C developers when creating programs. These techniques are considered best practices which greatly enhance the readability and maintainability of a program. During a code walkthrough, the software reviewers will be inspecting the code to make sure that these standards are adhered to.

2.1 Readability and Maintainability

See the *OHD Software Development Standards and Guidelines* for guidance on how to make your files more readable and maintainable. These are standards which apply to programming languages in general not just to the C language.

2.1.1 Indentation

The *OHD Software Development Standards and Guidelines* document states that consistent indentation shall be used when distinguishing conditional or control blocks of code.

Example:

Bad:

```
/* This example does NOT use consistent indentation. */
int main ( )
{
  int a = 1 ;
   int b = 2 ;

 if ( a == b )
 {
   fprintf ( stdout , “A and B are equal.\n” ) ;
   }
   else
   {
   fprintf ( stdout , “A and B are not equal\n” ) ;
   }

   return 0 ;
}
```

Better:

```
/* This example uses consistent indentation of 4 spaces. */
int main ( )
{
    int a = 1 ;
    int b = 2 ;

    if ( a == b )
    {
       fprintf ( stdout , “A and B are equal.\n” ) ;
```

```
    }
    else
    {
        fprintf ( stdout , "A and B are not equal\n" ) ;
    }

    return 0 ;
}
```

2.1.2 Braces

The programmer shall be consistent in the use of braces.

Example:

Preferred Style:

```
for ( i = 0 ; i < NUM_ANGELS_ON_A_PIN_HEAD ; ++i )
{
    /* Do some work here. */
}
```

Alternate Less Preferred Style (Kernighan and Ritchie):

```
for ( i = 0 ; i < NUM_ANGELS_ON_A_PIN_HEAD ; ++i ) {

   /* Do some work here. */
}
```

2.2 File Organization

See the *OHD Software Development Standards and Guidelines* for details on file organization. These are high-level standards and guidelines which pertain to programming languages in general, not just specifically to the C programming language.

2.2.1 Include Files

2.2.1.1 Header File Include Locations

A list of header files shall follow the file documentation block outlined in the *OHD Software Development Standards and Guidelines* document.

2.2.1.2 Header File Include Syntax

For include files, the '<' and '>' symbols shall be used to include system header files in your C source code. Double quotation shall be used for the inclusion of all other header files, including the header files that you define. List the system header files first in alphabetical order. Then list the non system include files (including COTS includes) also in alphabetical order.

Example

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

#include "pdc_engine.h"
#include "time_util.h"
```

2.2.1.3 Header File Paths

Do not use absolute or relative paths to point to your header files. It is mandatory that you use the –I<dir> directive of the C compiler to instruct the compiler where your header files are located.

2.2.1.4 Guarding Against Multiple Inclusions

Header files, particularly system header files, often include other header files. When this occurs, one or more header files may be included more than once. This is called multiple "inclusion" and is considered a poor programming practice. It may also keep source code from compiling or result in compile-time warnings. To prevent multiple inclusions, the following construct shall be used:

```
#ifndef   HEADER_FILE_H
#define   HEADER_FILE_H

/* Body of header_file.h file here. */

#endif    /* HEADER_FILE_H */
```

The multiple exclusion symbol is the header file name in upper case with the suffix _H. For example, for header file com_defs.h the multiple exclusion symbol would be COMMS_DEFS_H.

2.3 Variable and Function Scope

The C programmer needs to have a good grasp of the lifetime and scope of a variable. The following items describe common issues associated with variable scope.

2.3.1 Global Variables

The use of global variables shall be avoided. It is often difficult to determine where global variables originate. It is equally difficult to ensure that the value of a global variable is not modified in unexpected and unpredictable ways. It can be challenging to debug global variable values in large programs. If there is a need to access the value of a variable across files, then the scope of the variable should be made static to the file it is located in. Access functions should then be created in that file so that the value can easily be accessed by functions outside of that file. Doing this adds an objected oriented flavor to your C programming style by promoting information hiding and restricted access.

2.4 Variable Declaration, Initialization, and Qualifiers

Naming and defining variables in a clear and consistent fashion is an essential component in good programming. Assigning the const qualifier protects data from being inadvertently modified.

2.4.1 Leading and Trailing Underscores

Names with leading and trailing underscores are reserved for system use and shall not be used by the C programmer in this form or as part of a user created name.

Example:

Bad

```
int _customer_name;
int employee_name_;
```

Better

```
int customer_name;
int employee_name;
```

2.4.2 Constants

Constants defined by `#define` shall be in all CAPITAL letters. The same is true for enumerated (`enum`) constants and constants defined with a specific data type. This distinguishes these variables from variables with mutable values.

Example:

```
#define MINUTES_IN_HOUR 60

const short int MINUTES_IN_HOUR 60 ;
const short int num_minutes = MINUTES_IN_HOUR;
enum SevenDrawfs( BASHFUL, DOC, DOPEY, GRUMPY, HAPPY, SLEEPY,
                  SNEEZY ) ;
```

2.4.3 Names Differing Only by Case

Names that differ by case only, such as bar and Bar shall not be used.

2.4.4 Conflicts with Standard Library Names

Do not use names that might conflict with standard library function names.

Example:
Do not define a routine named `strcpy` for copying strings because the standard C string library has already defined a `strcpy` function for this purpose.

2.4.5 Local Variable Names

One of two conventions shall be used when naming variables. The programmer has the choice of separating words in the names of variables either by using underscores or by using camel case. In either case, the names of the variables shall be as descriptive as necessary to convey the meaning/usage of the variable. The programmer shall be consistent and use the same naming convention for variable names and function names.

Example using underscores:

```
/* When using underscores, the words in the names of variables shall be
separated by underscores not capital letters. */
int do_something ( )
{
   int the_first_thing_to_do ;
   int the_second_thing_to_do ;
  …
   return 0 ;
}
```

Example using camel case:

```
/* When using camel case, the first word in the variable name shall not
be capitalized.  Subsequent words in the variable name shall be
capitalized.  In camel case, words in the variable name are not
separated by underscores. */

int doSomething ( )
{
   int theFirstThingToDo;
   int theSecondThingToDo;
  …
   return 0 ;
}
```

2.4.6 User-Defined Type Names

The names of user-defined types including structures and unions shall have the first letter capitalized. Subsequent words in the name should also be capitalized (camel case) and should not be separated by underscores.

Example:

```
typedef struct
{
   char * city_name ;
   char * name_of_mayor ;
   int population ;
} CityStruct ;
```

2.4.7 Initialize pointers to NULL

A pointer which is declared but not initialized is known as a dangling pointer. Un-initialized pointers are dangerous because subsequent operations on the pointer will not know if the pointer references a valid memory address. When a pointer variable is declared always initialize it either with the address of an existing object or the value NULL. This usually sets the pointer's value to zero. The operating system is very strict about memory operations in the reserved zero page address space. So initializing pointers to NULL will quickly find code problems resulting from them being used before being set to point to valid memory addresses. Any place the pointer is subsequently used the C code shall first check the pointer to see if it is NULL before trying to dereference it.

Example:
```
char * pChar = NULL ;
char * pInt = NULL ;
int * pMeaningOfLife = NULL ;
int the_meaning_of_life = 42 ;

pMeaningOfLife = & the_meaning_of_life ;
```

2.4.8 Test pointers for NULL

Always test pointers for `NULL` values before trying to dereference them.

Example:
```
int * pInteger = NULL ;

...

if ( pInteger != NULL )
{
   fprintf ( stdout, "The value is %d.\n", *pInteger );
}
```

2.4.9 Use the Const Qualifier

Use the `const` qualifier to denote variables whose value should not change during the execution of a program. This allows the compiler to find places where a variable's value is being modified when it shouldn't be. Using const takes advantage of the compile-time error checking capabilities.

Example:
```
const int num_lives = 9 ;     /* The number of lives a cat
                                 has. */
++ num_lives ;                /* Results in a compilation
                                 error.  Besides, cats
                                 can't have 10 lives. */
```

2.4.10 Static Qualifier

Apply the `static` keyword to all file scope variables and functions whose use is local to a single file. This will prevent routines outside of the file from linking to any of the static modules in the file. It will also keep the namespace from becoming too cluttered and help to prevent conflicts caused by duplicate symbols.

Example:

The following are the contents of the number_cruncher.c file. Variable global1 is visible to all functions in this file and to all functions outside of this file. The declarations of variables global2, global3, and global4 are preceded by the static qualifier. Using the static keyword in this fashion sets the scopes of these variables to be this file only. That is, they are only visible to routines in this file. Furthermore, these variables are only visible from the point of their declaration to the end of the file. So, global2 is visible to

routines crunch_integers, crunch_floats, and crunch_doubles. Global3 is visible to crunch_floats and crunch_doubles. Global4 is only visible to crunch_doubles.

```
/* FileName:    number_cruncher.c
   Purpose:    Number Crunching Routines. */

int global1;
static int global2 = 0;

int crunch_integers ( )
{
}

static int global3 = 0;

int crunch_floats ( )
{
}

static int global4 = 0;

int crunch_doubles ( )
{
}
```

2.4.11 Initialize Static Variables When Declaring Them

The lifetime of a static variable is for the duration of a program. Always assign a static variable an initial value when declaring it. This is despite the fact that static variables will be automatically initialized to 0 (zero) when declared.

Example:

```
static int first_call_flag = 1 ;
```

2.4.12 Static/Const Arrays

Always make local arrays static. If the array's contents are not to be modified, then make the array `const`. Local variables are created on the stack each time a function is called. This can waste valuable CPU cycles, especially for arrays which are initialized with a list of initializers. Declaring these arrays as static will cause them to be created and initialized only once at compile time. Declaring these arrays as const will prevent their contents from being modified.

Example:

```
const static char * dwarfs [ ] = { "Bashful", "Doc", "Dopey",
                                   "Grumpy", "Happy", "Sleepy",
                                   "Sneezy" } ;
static int student_ids [ NUM_STUDENTS ] = {0} ;
```

2.4.13 Constant Declarations

Constant declarations shall appear in a header file. This promotes the sharing of the constants and reduces the likelihood that the same constant will be declared in multiple

source files. If a constant must be declared in a source file, then it should be declared in the beginning of a file after the include files. Placing definitions before the beginning of the first routine in a file allows for easier location and identification when searching for them.

2.4.14 Do Not Hardcode Values

(except 0, 1 and sometimes 2 used as basic math concepts)

Avoid hard-coding numerical constants in a C source module. Instead, declare a constant using either a `#define` directive or a `const` directive. In both cases, the constant name should be all uppercase. In fact, const is preferred over `#define` except in the case of array dimensions.

Declaring constants promotes code readability. Also, if the value of the constant needs to be modified, it only needs to be changed at the place it is declared. The code can then be recompiled and all references to that constant will receive the modified value.

Example:

```
const int MAX_DICE_ROLLS = 3;
const int PENALTY_POINTS = 100;

...

if ( num_rolls == MAX_DICE_ROLLS )
{
   printf ( "You have no rolls left!\n" );
}
else if ( num_rolls > MAX_DICE_ROLLS )
{
   /* Deduct points from the score. */
   score = score - PENALTY_POINTS;
}
```

Possible exceptions to this rule include:

- Cases in which the meaning of a hard-coded variable is obvious

  Example:

  ```
  /* Increment the number of minutes by one hour. */
  minutes_to_launch = minutes_to_launch + 60;
  ```

- Cases in which the hard-coded variable is a part of a standard formula

  Example:

  ```
  /* Convert the temperature from Fahrenheit to Celsius. */
  temperatureF = (temperatureC * 1.8) + 32.0;
  ```

- Cases in which the values of 0, 1 and 2 are used in basic mathematical operations such as initializing a variable to 0 or incrementing the value of a variable by 1. The value 2 may be used in binary mathematical operations.

Example:

```
/* Initialize the number of moles whacked to zero. */
num_moles_whacked = 0;

/* Increment Pujol's number of base hits by 1. */
albert_pujols_base_hits = albert_pujols_base_hits + 1;
```

2.5 Pointers and Dynamic memory

2.5.1 Prevent Memory Leaks

Always free dynamically allocated memory when it is no longer needed. Always free all dynamically allocated memory when an application terminates. Dynamic memory is deallocated by calling the `free ( )` function. Always set the value of a pointer to NULL after freeing the memory that it points to. Memory tools such as IBM Rational Purify or Valgrind can be used to detect the locations of memory leaks.

2.5.2 Returning Pointers from Functions

Do not return a pointer to a stack dynamic variable in a function if the variable has not been declared as static. The variables will go out of scope and be destroyed when the function ends and the pointer will be pointing to unused memory. If returning a pointer to memory which was dynamically allocated in a function (heap dynamic), make sure the caller knows that it is his/her responsibility to free the memory.

2.5.3 Dynamic Memory Allocation

Use dynamic memory allocation (`malloc`) prudently.

2.5.4 Create Large Arrays on the Heap

Use dynamically allocated memory for large arrays, as opposed to declaring stack-based variables. Allocating arrays dynamically will create them on the heap. The heap has more space than the stack.

2.6 Functions

2.6.1 Function Names

One of two conventions shall be used when naming functions. The programmer has the choice of separating words in the function name either by using underscores or by using camel case. In either case, the name of the function shall be as descriptive as necessary to convey the meaning/usage of the function. The programmer shall be consistent and use the same naming convention for function names and variable names.

Example using underscores:

```
/* When using underscores, the words in the function name shall be
separated by underscores not capital letters. */
int do_something ( )
{
   int the_first_thing_to_do ;
   int the_second_thing_to_do ;
  …
   return 0 ;
}
```

Example using camel case:

```
/* When using camel case, the first word in the function name shall not
be capitalized.  Subsequent words in the function name shall be
capitalized.  In camel case, words in the function name are not
separated by underscores. */

int doSomething ( )
{
   int theFirstThingToDo;
   int theSecondThingToDo;
  …
   return 0 ;
}
```

2.6.2 Function Prototypes

Function prototypes inform the compiler about the existence of functions before they are defined. This allows the code to compile without having to ensure all functions are defined before they are called. It also enforces prototype checking. That is, it makes sure that the correct number and type of arguments are being passed into the function.

2.6.3 Function Prototype Variables

The arguments specified in a function prototype shall be associated with variable names. These variable names must match the variable names in the function definition. Doing this makes function prototypes more meaningful. A programmer can tell more about the arguments which need to be supplied to a function if the arguments in the prototype have meaningful names associated with them.

Example:

Bad:

```
/* A function prototype whose arguments do not have variable
   names. */

float compute_total_price ( float , float );
```

Better:

```
/* The same function prototype with variable names for
   its arguments. */
```

```
float compute_total_price ( float price_of_merchandise ,
                            float sales_tax ) ;
```

2.6.4 Function Prototype Location

For all non-static functions, prototypes shall be declared in header files. These header files must be included in the files where the functions are defined and in any source file where the function is called.

2.6.5 Static Functions

If a function is only used in a module, then precede its definition with the keyword "`static`". This will ensure that the linker only tries to link this routine with routines in that file. It keeps the variable/function namespace from getting cluttered with the names of functions which nobody will use. It will also help to prevent collisions (linker duplicate symbol errors) between routines which have the same names.

Note that when a function is declared static, it does not need a prototype. Also, the function definition must be placed at a point in the file before the first time it is called.

In the example below, the crunch_integers and crunch_floats functions are static. They cannot be called by routines outside of this file. The visibility of static functions begins at the point they are declared in a file and continues until the end of the file. Prototypes are not needed for static functions and should not be used. Because of the limited visibility of static functions, the crunch_floats routine below can call the crunch_integers routine, but the crunch_integers routine cannot call the crunch_floats routine. The crunch_doubles routine can call both the crunch_integers and the crunch_floats routines. Also, the crunch_doubles routine is not declared as static. So it is visible to routines outside of this file.

Example

```
/* FileName: number_cruncher.c
   Purpose:    Number Crunching Routines. */
static int crunch_integers ( )
{
   /* Some crunching takes place here...*/
}

static int crunch_floats ( )
{
   /* Some crunching takes place here...*/
}

int crunch_doubles ( )
{
   crunch_integers ( );
   crunch_floats ( );
}
```

2.6.6 Explicit Function Return Types

Do not default a return type to `int`. Always explicitly state the return type of a function. If the function does not return a value, then make sure that its prototype and definition have return types of `void`. This should not be a problem if programmer is careful to define prototypes for all the functions he/she has created (except of course the static functions).

Example:

Do not Do This:

```
list_builder ( List ** pListHead ) ; /* No explicit return type.
                                        It defaults to int. */
```

Do This:

```
int list_builder ( List ** pListHead) ; /* Explicit return
                                           type. */
```

2.6.7 Do Not Reinvent the Wheel

Always use routines from the Standard C library when and where appropriate. Also, try to reuse routines developed by other programmers in your development organization.

2.7 Portability

Portability means that a source code file can be compiled and executed on different machines with the only changes being the possible inclusion of different header files and the use of different compiler flags. The header files contain `#define` and `typedef` constructs that may vary from machine to machine (a different machine may mean different hardware, a different operating system, a different compiler, or any combination of these).

2.7.1 Strive for Portability

Recognize that some things are inherently non-portable. These shall be avoided wherever possible.

2.7.2 Data Alignment

Avoid writing code which assumes that data are stored in a particularly way with respect to word boundaries in memory. For instance, it is not correct to assume that the size of a structure is the total of the sizes of its data elements. In fact, the size of a given structure can be different on different operating systems (HP vs. LINUX vs. Windows).

Data alignment is an important consideration. Various machines begin addresses at even numbers while others may do this but restrict the valid addresses to a multiple-of-four address.

3. GUIDELINES

C programming guidelines are a set of C programming practices which are considered best practices which enhance the readability and maintainability of a program. While developers are not required to use these techniques, they are encouraged to integrate them into their programming style.

3.1 Readability and Maintainability

See the *OHD Software Development Standards and Guidelines* for guidelines on how to make your files more readable and maintainable. These are guidelines which apply to programming languages in general, not specifically to C. C specific items will be discussed in the sections below.

3.2 File Organization

See the OHD *Software Development Standards and Guidelines* for guidelines on file organization. These are guidelines which pertain to programming languages in general, not specifically to the C programming language.

3.2.1 Source File Names

The names of source files which belong to a common library or an executable should have a common prefix which identifies them as being part of that library or executable. This includes header files. This helps programmers quickly determine which library a source file belongs to, especially a header file.

Example: The names of the source files used to build the libPdcEngine.a library could appear as follows:

```
pdc_engine_parser.c
pdc_engine_reader.c
pdc_engine_writer.c
```

3.2.2 Comments

See the *OHD Software Development Standards and Guidelines* document for details on writing prologue documentation in a source file.

3.2.3 C Comment Organization

Three general types of comments are encountered in C programming:

Block comments

Example

```
int main ( )
{
   /* This is an example of a block comment.
```

```
      It spans multiple lines. Be careful not
      to forget the closing comment symbol. */

   /* This as an alternative example of block comments.
    * In this example, there is a '*' character at the
    * at the beginning of each line.  This makes the
    * block stand out visually, making it more obvious
    * to the reader that this is a block comment, not code.
    * In this style, the final closing comment is on a line
    * by itself:
    */

   return 0;
}
```

One-line comments

Example

```
int main ( )
{
   /* This is an example of a one-line comment. */
   return 0 ;
}
```

In-line comments

Example

```
int main ( )
{
   int a = 1;     /* This is an example of an inline comment. */
   int b = 2;
   return 0;
}
```

Different forms of comments are appropriate for different places in source code. Place comments describing data structures, algorithms, and the like in block form. Generally one-line comments and inline comments describe a short code fragment. Try to align one-line comments with the code they describe. Place enough space between the code and the beginning of inline comments to promote readability.

Placing a blank line before and after a block or one-line C comment to separate it from the surrounding source code will make the source code easier to read.

3.3 Variable and Function Scope

The C programmer needs to have a good grasp of the lifetime and scope variables and functions.

3.4 Variable Declaration, Initialization, and Qualifiers

Naming and defining variables in a clear and consistent fashion is an essential component in good programming. Assigning the `const` qualifier protects data from being inadvertently modified.

3.4.1 Loop Index Variable Names

Loop index variable names may be short and do not need to be descriptive.

```
int i ;    /* A loop index variable. */

for ( i=0; i < NUM_ITEMS; ++ i )
{
    …
}
```

3.4.2 Pointer Names

Pointers should be named in some fashion that distinguishes them from other "ordinary" variables. One possibility is that pointer variable names start with a lower case "p" followed by the rest of the name with the first character capitalized, e.g. pValue. This makes recognition of pointer variables in source code easier.

Another possible naming convention for pointers is to append "ptr" to the end of pointer variable names. The name of the pointer should either imply the type of the data object it is referencing or the name of the pointer should describe the data it represents.

The programmer should be consistent in naming pointers. Do not mix different naming conventions.

Examples:

/* Pointer names using the "p" naming convention. */

```
CityStruct * pCityStruct = NULL ;   /* A pointer to a CityStruct
                                       structure. */
char * pChar = NULL ;    /* A pointer to a character. */
char * pComma = NULL ;   /* A pointer to the comma returned
                            by a call to the C library function
                            strchr. */
int * pStudentAges = NULL ;  /* A pointer to an array of
                                student ages. */

/*Pointer names using the "Ptr" naming convention. */
char * charPtr = NULL ;  /* A pointer to a character. */
char * commaPtr = NULL ; /* A pointer to the comma returned
                            by a call to  the C library function
                            strchr. */
```

3.4.3 User-Defined Types

3.4.3.1 Enumerated Types

You may use an `enum` to place constants together in a logical group or when actual values for the constants are unimportant. The value of the first item in an `enum` list is always 0 (zero) unless otherwise specified. As with other constants, use uppercase characters to construct the constant name. Notice that proper alignment and indentation will improve readability. If an enum is going to be used in mapping elements into an array, then it is often helpful to have the last constant in the enumeration be named something like NUM_ENUM_ELEMENTS. An array can then be dimensioned to have NUM_ENUM_ELEMENTS, and the array will then have an element for each constant in the enumeration (with the exception of the NUM_ENUM_ELEMENTS constant).

Example:

```
typedef enum Clouds { CIRRUS, ALTOCUMULUS, CUMULONIMBUS, NIMBUS,
                      STRATUS, NUM_CLOUD_TYPES } CloudTypes ;

/* This array will have space for the CIRRUS, ALTOCUMULUS,
   CUMULONIMBUS, NIMBUS, and STRATUS types. */

CloudTypes cloud_array [ NUM_CLOUD_TYPES ] ;
```

3.4.3.2 Macros

Macros should be used judiciously. Macros are useful to represent really short blocks of logic which are called only a few times. If the block of logic is used more frequently, then it is preferable to use a small static function in place of a macro.

3.4.3.2.1 Use Parentheses in Marco Definitions

When using macros, it is essential to use parentheses to ensure correct evaluation of the macro.

Example:

```
/* The following macro definition could result in an error. */
#define PI 3.14159
#define CIRCLE_AREA(x)   ( PI * x * x )

area = CIRCLE_AREA ( c + 2 ) ;

/* The preprocessor will expand this macro to: */

area = PI * c + 2 * c + 2 ;

/* Given operator precedence in C, this will be incorrectly
   evaluated as: */

area = ( PI * c) + ( 2 * c ) + 2 ;

/* The following macro definition uses parentheses to ensure that it is
```

```
   evaluated as the programmer intended it to be. */

#define CIRCLE_AREA(x) ( PI * ( x ) * ( x ) )
area = CIRCLE_AREA ( c + 2 ) ;

/* The preprocessor will expand this macro to: */

area = PI * ( c + 2 ) * ( c + 2 ) ;
```

This will evaluate in the correct order.

3.4.3.3 Structures

Structures (also known as records) are useful for creating collections of variables of different types. Their use also makes I/O operations more efficient. Structures are also useful for passing arguments to a function. For example, suppose a function takes 15 arguments, ten of which are option flags. Instead of passing these option flags individually into the function, it is better (from software engineering and code maintainability standpoints) to create a structure which contains these option flags and pass a pointer to this structure into the function. This has the advantage of increasing the efficiency of the function call. It also makes it easier to add new options in the future without having to modify function prototypes.

Example:

```
/* This function takes several closely related calling arguments.
   Having multiple calling arguments increases the overhead of
   calling this function. It also makes it more difficult to add
   new calling arguments as may required by future enhancements.*/

static int get_river_data ( char * PE,
                            char * TS,
                            int maximum_observed_forecast,
                            int show_only_data_above_floodstage )
{
   …
}

/* This version of the get_river_data routine takes a
   pointer to the RiverData structure as its only argument.
   This structure contains all of the elements passed into
   the get_river_data routine above.  Note that this pointer
   is qualified with the const keyword.  Passing the structure
   as a pointer reduces the overhead of calling this function.
   The const keyword ensures that the function does not
   modify the structure passed into it. */

typedef struct RiverData
{
   char PE [ SHEF_PE_LEN + 1] ;
   char TS [ SHEF_TS_LEN + 1] ;
   int maximum_observed_forecast ;
   int show_only_data_above_floodstage ;
```

```
} RiverData ;

static in get_river_data ( const RiverData * pRiverData )
{
   …
}
```

3.5 Pointers and Dynamic memory

Misuse of pointers and dynamic memory allocation is the cause of many C program bugs. The following are guidelines to help the programmer avoid software bugs associated with the incorrect use of pointers.

3.5.1 Use Pointers as Arguments to Functions

User-defined types can be large. C implements pass by value for function arguments. This means that the function receives a copy of each calling argument. To increase the speed with which arguments are passed, use pointers to pass user-defined types to routines. If a user-defined type should not be modified by the routine it is being passed to, make sure that the routine receives it as a `const` pointer.

3.6 Functions

3.6.1 Functions versus Macros

Consider using macros instead of invoking functions. This is especially true when:

- The amount of source code in the function is small
- The function is called a small number of times.

3.6.2 Inline Functions

Considering inlining functions if the functions are:

- Small
- Called many times, as from a loop

```
/* This function writes a message to a log file.
 * Since it is called many times from within the
 * program, it has been assigned the ‘inline’ qualifier.
 * Based on this qualifier, the compiler may or may
 * not inline the function. */

inline void log_message ( FILE * pLogFile,
                          const char * pMessage )
{
   if ( pLogFile == NULL )
   {
      fprintf ( stderr, “NULL log file argument.\n” );
      return;
   }

   if ( pMessage == NULL )
```

```
   {
      fprintf ( stderr, "NULL log message argument.\n" );
      return;
   }

   /* Write the log message to the log file. */
   fprintf ( logFile, message );
}
```

Remember that the `inline` keyword is only a suggestion to the compiler. The compiler may or may not perform the inline based on how it optimizes code.

3.6.3 Functions in Loops

Limit use of library functions, especially in loops. For example, consider using the form `x*x` rather than the math library function `pow (x, 2 )`.

3.6.4 Embedded Statements

An embedded assignment statement is a form of side effect. Avoid using embedded assignment statements.

Example:

Bad:

```
if ( (status = strcmp ( string1, string2 ) ) )
{
    fprintf ( stderr , "\nThe strings are not equal.\n" ) ;
}
```

Better:

```
status = strcmp ( string1, string2 ) ;

if ( status != 0 )
{
    fprintf ( stderr , "\nThe strings are not equal.\n" ) ;
}
```

3.7 Program Control

3.7.1 Avoid Unnecessary Code in Loops

Whenever possible, move code out of loops when it does not depend on the loop index.

3.7.2 Combining Loops

Consider combining two or more loops to reduce total loop overhead costs.

3.7.3 Loop Direction

Loops which count down to 0 can be faster than loops which count up from 0 (especially when loop counters are declared using the `register` keyword)

3.7.4 Conditional Tests

Optimize logical tests by reordering. When a series of logical tests is performed and one of the tests is significantly faster and capable of determining the result, perform the faster, most capable test first.

Example

Acceptable:

```
if ( ( ( a == 1 ) || ( b == 1 ) || ( c == 1) || ( d == 1 ) )&&
     ( f == 1 ) )
{
      …
}
```

Better:

```
/* This is better because the test of f == 1 determines immediately
   whether the entire condition will evaluate to true. This is an
   example of short circuit evaluation.*/
if ( ( f == 1) && ( ( a == 1) || ( b == 1) || ( c == 1) ||
                    ( d == 1) ) )
{
      …
}
```

3.7.5 Explicit Conditional Tests

Defaulting tests for non-zero is generally a bad idea if the variable being tested is not being used as a Boolean variable. We recommend that explicit tests against a set value be coded, as:

Example:

```
if ( result )   /* Ok, but … */

if ( result != FAIL )  / * This is much better from a readability
                           standpoint. */
```

Generally, it is better to test values for inequality with 0 (FALSE) rather equality with 1 (True). Most functions (but not all) are guaranteed to return 0 (zero) if false, but only non-zero (i.e. *any* value other than 0) if true.

Example:

```
status = some_function ( arg1 , arg2 , arg3 ) ;
```

```
/* Check return status value for an error condition. */
if ( status != 0 )
{
   /* An error occurred. */
}
```

If the variable is being used as a Boolean (that is, it has a value of 1 if true and a value 0 if false) and the variable is named in a fashion that indicates its Boolean nature, then an explicit value is not required in the conditional test. In fact, using an explicit value in this instance can lessen the program's readability.

Example:

```
int is_empty = 1 ; /* This variable may have a value of 0 or 1.
                      Its name indicates its Boolean nature.
                      When used in a conditional test its
                      Meaning is clear without the need for
                      an explicit value. */

if (is_empty)
```

is the same as

```
if ( is_empty != 0 )
```

from an efficiency point of view. But, in this case, the first form is more readable.

3.7.6 Goto

Use the `goto` statement very sparingly. Structured programming techniques have practically eliminated the need for the `goto` statement. In the opinion of many programmers, the `goto` statement should never be used.

With that said, there are a couple of situations where a `goto` may be employed if absolutely necessary. It may be employed to break out of several levels of for/while/switch nesting (though you should rethink any code that meets this condition.) It may also be used in source code that performs a lot of error testing the result of which are premature returns from the source module. The `goto` statement can be used to redirect the error statements to a common exit or return point in the module.

Example:

```
int test_value_validity ( int value )
{
   int status = OK ;

   /* First error test */
   if ( value < 0 )
   {
      status = NotOk ;
```

```
      goto RETURN_BLOCK ;
   }

   /* Some work happens here. */

   /* Second error test */

   if ( value > 100 )
   {
      status = NotOk ;
      goto RETURN_BLOCK ;
   }

   /* Some work happens here. */

   /* Third error test */
   if ( value != 42 )
   {
      status = NotOk ;
      goto RETURN_BLOCK ;
   }

   /* Some work happens here. */

   RETURN_BLOCK:

      printf ( "An invalid value, %d, has been passed into function "
               "test_value_validity.\n", value );
      return status ;
}
```

3.8 Portability

Portability means that a source code file can be compiled and executed on different machines with the only changes being the possible inclusion of different header files and the use of different compiler flags. The header files contain `#define` and `typedef` constructs that may vary from machine to machine (new machines may mean different hardware, a different operating system, a different compiler, or any combination of these).

3.8.1 Organize Files According to Portability

Try to organize machine-independent code in separate files.

3.8.2 Big Endian/Little Endian

Recognize that some machines are little endian and some are big endian. Byte ordering in words (and further, word ordering) is important.

3.8.3 Bit Manipulations

Bit shifts and bit masks are affected by word size. Do not assume all machines have the same word size.

3.8.4 Casting

Avoid downcasting from larger to smaller types. If you must downcast to a smaller type, use an explicit cast.

```
/* Some work happens here. */
int four_byte_number = 45;
short two_byte_number;

/* An explicit cast from integer to short is used. */
two_byte_number = ( short ) four_byte_number;
```

3.8.5 Numbers

3.8.5.1 Floating Point Numbers

Include at least one digit on either side of the decimal point for floating point variables.

```
const float GRAVITY_EFFECT = 0.937 ;
float salary = 0.0 ;
```

3.8.5.2 Hexadecimal Numbers

Start hexadecimal variables with 0x and use uppercase for A-F.

Example:

```
const int MAX_WEIGHT = 0x1F4A;
```

3.8.5.3 Octal Numbers

Use caution when initializing values for numeric constants and variables. A value with a leading zero is assumed to be octal, not decimal.

Example:

```
/* This defines the number of minutes in an hour as a decimal 60. */
#define NUM_MINUTES_IN_HOUR 60
/* This defines the number of minutes in an hour as a decimal 48. */
#define NUM_MINUTES_IN_HOUR 060
```

3.9 C Program Performance

Modern compilers have sophisticated code optimization features. While it is likely that a compiler will optimize code to take into account many of the following items, the programmer is encouraged to understand these simple practices which can make a program run faster.

Program readability needs to be considered along with program performance. If the performance of a program is acceptable, then it is better to avoid performance enhancing code which can be tricky and harder to read.

3.9.1 The Register Keyword

Use the "`register`" keyword to indicate variables, such as loop indexes, which are used frequently in a program. The `register` keyword instructs the compiler to place the values of these variables in registers on the CPU. This makes their use more efficient. Modern compilers are able to recognize variables which will benefit from being stored in registers. Modern CPUs generally have many registers which are at the disposal of the compiler. The register keyword is just a suggestion to the compiler. The compiler is free to ignore it.

Example:

```
register int i ;  /* A loop index variable. */
```

3.9.2 Shorts, Ints, and Longs

If `int` is shorter than `long`, it is usually faster to perform operations on `int`.

Division and right-shift of `unsigned short` can be as slow as the same operations on a long integer.

3.9.3 Boolean Values

Consider using multiple `char` variables to store Boolean values (unpacked) rather than storing multiple Boolean values in a single `char` (packed). Doing this eliminates bit shifts, bit AND, and bit OR operations (but it uses more space).

3.9.4 Initializing Memory

Consider using `memset` to initialize large areas of memory (any variable type), where all values must be the same.

3.9.5 Copying Memory

Consider using `memcpy` to copy an entire multidimensional array to another multidimensional array all at once instead of using nested loops.

3.9.6 Logical NOT

The Logical NOT does not usually introduce additional calculations. The statements

```
if (is_empty) and

if (!is_empty)
```

typically generate the same amount of code and require the same number of CPU cycles.

3.9.7 I/O

I/O Operations almost always result in lost program efficiency. This includes writing and reading data to and from files and databases. If the data in the file or database table needs to be frequently read, consider buffering the data in memory after the first time it is read. This results in increased memory usage but reading data from memory is much faster.

3.9.8 Unary operators versus binary operators

The form `x++` is more efficient that `x=x+1` or `x+=1` (ditto for `--`).

3.9.9 Binary Multiplication

Consider replacing multiplication of integer variables by powers of 2 with left shifts (e.g., `i=j<<2;` rather than `i=j*4;`). This is also applicable for division by powers of 2 and right shifts.

On some machines, shifts of one bit can be faster than shifts of multiple bits. So consider an implementation which involves a single bit shift instead of one which involves a multi-bit shift.

3.9.10 Searching and Sorting

When sorting and searching large amounts of data, use sort and search technique patterns most appropriate for the type and use of data (e.g., sequential search, binary search, heap sort, shell sort, and quick sort).

3.9.11 Store Frequently Used Results

Reduce repetitive computations by only doing them once and saving the result in a temporary variable for future access

Also, consider storing frequently manipulated strings into a table and use table indexes or pointers to reference these strings.

4. REFERENCES

- *Strategies for Streamlining Software Performance*
- *RPG C Coding Standards – Draft # 1*
- *Standards and Style for Coding in ANSI C*
- *The C Programming Language, 2nd Edition, Brian Kernighan and Dennis Ritchie*