NOAA NESDIS CENTER for SATELLITE APPLICATIONS and RESEARCH

TRAINING DOCUMENT

TD-11.2 C PROGRAMMING STANDARDS and GUIDELINES

Version 3.0

TITLE: TD-11.2: C PROGRAMMING STANDARDS AND GUIDELINES VERSION 3.0

AUTHORS:

Alward Siyyid (formerly Raytheon Information Solutions – version 1)

Ken Jensen (Raytheon Information Solutions)

Peter Keehn (PSGS)

Shanna Sampson (PSGS)

C PROGRAMMING STANDARDS AND GUIDELINES
VERSION HISTORY SUMMARY

Version	Description	Revised Sections	Date
1.0	New Work Instruction (WI) adapted from Raytheon SOI 506 by Alward Siyyid (Raytheon Information Solutions)	New Document	05/05/2006
1.1	Revision by Ken Jensen (Raytheon Information Solutions). Applied STAR standard style to entire document.	All	06/02/2006
2.0	Revision by Shanna Sampson (Perot), Peter Keehn (Perot), and Ken Jensen (Raytheon Information Solutions). Changed from WI-12.1.2 to Training Document TD-12.1.3 for version 2 of the STAR Enterprise Product Lifecycle (EPL). Numerous revisions in response to peer review comments.	All	09/30/2007
3.0	Renamed TD-11.2 and revised by Ken Jensen (RIS) for version 3.	1, 2	10/1/2009

TABLE OF CONTENTS

LIST OF ACRONYMS

ANSI	American National Standards Institute
CICS	Cooperative Institute for Climate Studies
CIMSS	Cooperative Institute for Meteorological Satellite Studies
CIOSS	Cooperative Institute for Oceanographic Satellite Studies
CIRA	Cooperative Institute for Research in the Atmosphere
COTS	Commercial Off-The-Shelf
CREST	Cooperative Remote Sensing and Technology Center
EPL	Enterprise Project Lifecycle
IEC	International Engineering Consortium
ISO	International Organization for Standardization
NESDIS	National Environmental Satellite, Data, and Information Service
NOAA	National Oceanic and Atmospheric Administration
PAL	Process Asset Library
SLOC	Source Lines Of Code
STAR	Center for Satellite Applications and Research
TBR	To Be Reviewed
TBS	To Be Specified
TD	Training Document

1. INTRODUCTION

The NOAA/NESDIS Center for Satellite Applications and Research (STAR) develops a diverse spectrum of complex, often interrelated, environmental algorithms and software systems. These systems are developed through extensive research programs, and transitioned from research to operations when a sufficient level of maturity and end-user acceptance is achieved. Progress is often iterative, with subsequent deliveries providing additional robustness and functionality. Development and deployment is distributed, involving STAR, multiple cooperative institutes (CICS, CIMSS, CIOSS, CIRA, CREST) distributed throughout the US, multiple support contractors, and NESDIS operations.

NESDIS/STAR is implementing an increased level of process maturity to support the exchange of these software systems from one location or platform to another. The purpose of this coding standards guideline is to assist software developers reliably and repeatably develop, port, and deliver NOAA/NESDIS environmental software systems across platforms, locations, and organizations.

1.1. Objective

The objective of this Training Document (TD) is to provide the STAR standard for C language code that is developed, tested, and reviewed during the STAR Enterprise Product Lifecycle (EPL)[1]. The intended users of this TD are programmers of C code that will be used to implement an algorithm that creates an operational product from remote sensing satellite data. To achieve the objective, this TD shall:

- Establish C programming standards for STAR, drawn from international standards
- Provide C programming guidelines standards for the C language to support software modularity, readability, reliability and maintainability.
- Provide examples of good C programming practices
- Serve as a common reference for programming practices within the STAR Enterprise.

[1] For a description of the STAR EPL, refer to the STAR EPL Process Guidelines (PG-1 and PG-1.A).

1.2. Background

This Training Document (TD) defines programming standards and provides implementation guidelines for coding in the C programming language for software programs within STAR. This TD has been adapted from Raytheon Software Operating Instruction 506 "C Programming Standards and Guidelines" and has benefited from review and comments by Alex Pozniak, Hongming Qi, C. Meng and Walter Wolf.

The development of code that will be used to implement an algorithm that creates an operational product from remote sensing satellite data is part of a unified STAR EPL. As such, it takes place in a series of defined steps:

Basic Research Code: In this step, a new or improved algorithm is being developed by a scientist. Usually, some coding is needed to implement the Basic Research algorithm so that the algorithm developer can do sufficient testing to determine whether the algorithm has operational potential. At the discretion of the Basic Research organization, Basic Research code can be prototype "throwaway" code that does not have to conform to standards, and there will be no code review at this step. If the programmer intends to reuse his Basic Research code in future steps, he should be aware that the reused code will be required to conform to standards.

Research Grade Code: In this step, the algorithm has been identified as having operational potential and additional development has been authorized to determine whether a STAR Research Project proposal should be submitted. Research grade code is a required artifact for the STAR review of a Research Project Proposal. STAR reviewers will expect that this code can be re-used in the development of pre-operational code. The conformance of the code to these standards may be a factor in STAR's decision to approve the project for development.

Pre-operational Code: In this step, the algorithm has been approved for development and has passed a Critical Design Review. The code is developed from research grade to pre-operational status. The conformance of the pre-operational code to the standards in this TD shall be a factor in a decision to approve its installation in an operations environment.

Operational Code: In this step, the pre-operational code has been successfully integrated into the operational environment and is ready for approval for operations. There are no additional programming standards for operational code.

1.3. C Versions

C89 (ANSI X3.159-1989 "Programming Language C.) was the very first version by the ANSI standards committee and is often referred to as ANSI C, or sometimes C89. In 1990, the ANSI C standard was adopted by the International Organization for Standardization (ISO) as ISO/IEC 9899:1990. This version is also known as C90.

In the late 1990s, the standard underwent revision, leading to the publication of ISO 9899:1999 in 1999, also known as C99.

There are several new features in C99 such as inline functions, variable length arrays, new data types such as complex type to represent complex numbers, etc.

This TD is tailored for STAR, based on the recognition that many C programmers and potential reviewers in the STAR Enterprise are accustomed to the C99 version, although that is not required, since C89 is fully compatible. Appendix A-D of this TD contains programming examples in C99.

1.4. Benefits

Code developed in accordance with the standards in this TD assists the programmers and testers by increasing the efficiency of code testing and debugging.

Code developed in accordance with the standards in this TD assists code reviewers by ensuring that the code presented for review is well documented, readable, and traceable to design.

Code developed in accordance with the standards in this TD makes it easier to perform code maintenance during operations.

Most important to the programmer, it is a STAR requirement that C code be developed in accordance with the standards in this TD. Failure to do so may result in disapproval and the need to rewrite the code.

1.5. Overview

This TD contains the following sections:

Section 1.0 -	Introduction

Section 2.0 -	Reference Documents
Section 3.0 -	Definitions
Section 4.0 -	Programming Standards and Guidelines
Appendix A -	Example Comments
Appendix B -	Example Function Declarations and Definitions
Appendix C -	Example Control Structures
Appendix D -	Include File Usage
Appendix E -	C Coding Standards Quick Reference

2. REFERENCE DOCUMENTS

Programming Languages - C (formerly ANSI/ISO/IEC 9899-1999) is the current international standard for C code. It is accessible at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf. This is a very large document (> 3 MB) that can be used as a reference at the programmer's discretion.

K&R: Kernighan, Brian W. and Dennis M. Ritchie. *The C Programming Language*, Second Edition, Prentice-Hall, Inc., 1988, 1978.

Plum: Plum, Thomas. *C Programming Guidelines*, Second Edition, Plum Hall Inc., 1989.

Harbison III and Steele: Samuel P. Harbison III and Guy L. Steele, C: A Reference Manual, 5th Edition, Prentice Hall, February 2002, ISBN 013089592X

The following references are STAR EPL process assets that are accessible in a STAR EPL Process Asset Repository (PAR) on the STAR web site:

http://www.star.nesdis.noaa.gov/star/EPL_index.php.

PG-1: STAR EPL Process Guideline provides the definitive description of the standard set of processes of the STAR EPL.

PG-1.A: STAR EPL Process Guideline Appendix, an appendix to PG-1, is a Microsoft Excel file that contains the STAR EPL process matrix (Stakeholder/Process Step matrix), listings of the process assets and standard artifacts, descriptions of process gates and reviews, and descriptions of stakeholder roles and functions.

3. DEFINITIONS

Compilation Unit. A compilation unit is the code file, which is submitted to the compiler to produce an object file. A compilation unit consists of comments, preprocessor statements (optional), declaration lists (optional), and functions. A compilation unit may also be a complete program.

Compound Statement. A compound statement consists of a beginning left brace "{", preprocessor statements (optional), a declaration list (optional), a statement list (optional), and a closing right brace "}".

Function. A function consists of a function declaration (used to name the function and declare its type), formal argument, and a compound statement.

Functional Description. A functional description is a summary of the function's purpose and consists of information needed by the user.

Line of Code. For the purposes of counting software size, a source line of code (SLOC) is defined as:

- A semicolon terminator outside of comments, parentheses and string/character literals
- Compiler directives ('#')
- One of the following statements:
    - if
    - switch
    - while
    - case
    - for
    - default

Mixed Mode. This term is used in reference to variables of differing types, which are equated, compared, or otherwise used in arithmetic expressions.

Process Description. A process description is a detailed expansion of the functional description and consists of information needed by the maintainer.

Program. A C program is composed of functions one of which must be *main*. The functions may be in one or more software units.

For additional definitions, refer to **K&R**

4. PROGRAMMING STANDARDS AND GUIDELINES

This section contains the STAR C programming standards and associated guidelines.

4.1. Language Features

Only language features and capabilities that are documented or defined in the Programming Languages - C (formerly ANSI/ISO/IEC 9899-1999) shall be used.

One of the requirements for code to pass Gate 2 and Code Unit Test reviews is that the code is able to compile on a standard C compiler. It is recommended that C code developers use a STAR-approved C compiler while they are developing the code.

4.2. Readability

STANDARD: Formatting style shall be defined and used consistently to enhance readability throughout a program (e.g., alphabetic case, blocking with blank lines, parentheses and indentation).

JUSTIFICATION: A consistent and readable programming style will enhance the maintainability of the software, thus driving down maintenance costs.

STANDARD: Lines within a compilation unit should fit a listing (or screen) width of 80 characters. Any expression that is too long to fit this size should be broken into multiple lines.

JUSTIFICATION: A line of code should be able to be viewed on a single display line without having to scroll left and right. Such scrolling is distracting when trying to read or maintain code.

GUIDELINE: For each level of indentation, use three spaces. The number of spaces used should be applied to the entire program.

JUSTIFICATION: Uniform indentation gives a uniform appearance and makes levels of nesting more obvious.

GUIDELINE: Upper case should be used for user ***#define***'d identifiers and Macro names.

JUSTIFICATION: Such visual clues are very valuable in understanding the code, especially where it clearly separates variables from constants and functions from macros.

GUIDELINE: Mixed case or underscores should be used for function and variable names in one of the following forms:

i. The first character of each word in upper case with the remaining characters in lower case.

ii. Each word separated by the underscore character.

iii. A combination of the above.

Example:

```
#defines
        #define  END_OF_FORM    61    /* Last printable line  */
        #define  START_OF_FORM   5    /* First printable line */
VARIABLES AND FUNCTIONS
        get_data();
        GetData();
        Get_Data();
        FirstLine = START_OF_FORM;
        First_Line = START_OF_FORM;
        first_line = START_OF_FORM;
```

JUSTIFICATION: The use of a consistent style for different parts of the language will provide instant recognition. The ability to recognize one language structure versus another will greatly enhance the readability.

4.3. Naming Conventions

STANDARD: Names shall be mnemonically descriptive, given limitations within any C language implementation.

JUSTIFICATION: Using a naming convention, which describes the purpose and not the structure, will enhance understanding. This level of understanding makes maintenance easier.

GUIDELINE: Names should not resemble C reserved words or implementation supplied function names.

JUSTIFICATION: Using names, which resemble reserved words, will only make the job of understanding the software more difficult.

4.4. Compound Expressions

STANDARD: The evaluation of logical and arithmetic expressions shall be clarified through the use of parentheses:

Example 1:

The following example shows how the use of parentheses clarifies the intent of the code:

```
if (flags & BIT1 && state == PARSE)           /* bad example            */
        if ((flags & BIT1) && (state == PARSE))  /* use of parentheses     */
```

Example 2:

The following example shows how the use of parentheses, as well as line breaks, improves code readability.

```
/* bad example */
value =         option_base + OPTION_OFFSET + array_value *
                    ENTRY_SIZE + VALUE_OFFSET;

/* grouping by use of parentheses and line breaks */
value =         (option_base + OPTION_OFFSET) +
                    (array_value * ENTRY_SIZE) +
                    VALUE_OFFSET;
```

JUSTIFICATION: The ability to understand the intent of the software will be increased with the readability of the structure. Explicitly defining the order for evaluation will eliminate the possibility of the programmer getting something he or she did not want. Readability of the code is enhanced by a uniform layout of the operators.

STANDARD: Multiple line expressions should be broken in a manner that enhances the readability of the expression.

JUSTIFICATION: Care must be taken when creating multiple line expressions to maintain the readability.

GUIDELINE: The indentation of multiple line expressions should enhance the readability of the expression.

Example:

```
if ((strcmp(Direct->Name, ".") == MATCH) ||

   (strcmp(Direct->Name, "..") == MATCH))
```

JUSTIFICATION: Complex expressions will become clearer when properly distributed into multiple lines. This guideline is for enhancing the understanding, through readability, of long complex expressions.

4.5. Organization

STANDARD: Compilation units shall be organized as follows:

i. Compilation unit *header*.

iv. Include files (#include).

v. Declarations (those not in Include files or functions).

vi. Functions.

JUSTIFICATION: Properly organized software can save many hours of debug time and make the software easier to understand.

STANDARD: Functions shall be organized as follows:

i. Function declaration.

vii. Function *header* (optional in single function compilation units).

viii. Compound statement.

JUSTIFICATION: Properly organized software can save many hours of debug time and make the software easier to understand.

STANDARD: Each function shall contain a maximum of 150 lines of code**.**

JUSTIFICATION: Excessively large blocks of code have been shown to be a detriment to both the understanding and maintenance of software.

4.6. Size

GUIDELINE: Each program unit should contain no more than 200 statements.

JUSTIFICATION: Larger functions and programs tend to loose logic procession and become difficult to understand, maintain and are more prone to error.

4.7. Entry/Exit

STANDARD: Exit points of a function, not occurring as the last executable statement, shall contain a comment that begins with the word: *EXIT*.

JUSTIFICATION: Clear commenting of the exit point will make the algorithm easier to follow. This becomes more important for clarity when the exit point is not at the end of the function.

GUIDELINE: Each function should contain a single exit point as the last executable statement. If the exit point is not the last executable statement, the last line of the function should be a comment identifying where the exit point(s) are located within the function.

JUSTIFICATION: Multiple exit points can make software hard to understand and debug. Single exit points to functions are desirable to ease understanding of the algorithm.

4.8. Declarations

GUIDELINE: Declaration sections should be ordered in one of two ways.

i. Creating logical groups based on usage.

ii. The following format:

- Constants (#define).
- Type definitions (typedefs, unions, structs)
- Global variable declarations of simple types (int, long, float, char, etc.).
- Global variable declarations of compound types (typedefs, structs, unions, and arrays).
- Non-global (static) declarations

JUSTIFICATION: The declaration section is key to understanding a program. Efforts made to make this section easy to follow will pay dividends in the maintenance stage of software development.

4.8.1. Conditional Compilation

STANDARD: The ***#ifdef*** directive shall not be used to disable unused code. The directive is to be used for conditional compilation (e.g., debug code, platform dependencies, compiler dependencies).

JUSTIFICATION: The purpose of the ***#ifdef*** directive is for conditional compilation; not to allow disabling of code for version control, CR fixes, etc. This directive when abused can have the same effect as the comment statement.

4.8.2. Include Files

STANDARD: An include file shall contain only definitions, declarations, macros, function prototypes, comments, and conditional compilation statements that are needed by more than one compilation unit.

JUSTIFICATION: The include file or "*header*" should be used to control declarations for the program. The use of an include file to segregate sections of coding logic, for whatever reason, only makes the code harder to understand.

GUIDELINE: Any file that uses definitions provided by another file should explicitly include that file.

JUSTIFICATION: Files should not depend on "inherited" code from other files. This can create sequential dependencies and maintenance problems.

NOTE: See Appendix D for examples of proper include file usage.

GUIDELINE: Use ***#ifdef*** statements in include files to prevent redefinition of values.

JUSTIFICATION: ***#ifdef*** statements prevent accidental redefinition of values when include files are nested.

NOTE: See Appendix D for examples of proper include file usage.

GUIDELINE: Programs should take advantage of include files which contain program wide standard definitions (#defines, typedefs, macros, etc.).

JUSTIFICATION: Program-wide include files will serve to enhance the use of standard data types. Program-wide include files are also a good way to segregate machine dependent data types, which may be changed easily during porting efforts.

GUIDELINE: Include files should not contain data storage.

JUSTIFICATION: Placing a variable in an include file can accidentally generate several copies of the variable in memory, causing subtle debugging problems.

GUIDELINE: A multiple include guard should be used in every #include file.

```
#ifndef FILENAME_H
#define FILENAME_H
….
   #endif
```

JUSTIFICATION: Using a multiple include guard avoids problems with multiple includes.

4.8.3. Function Declarations

STANDARD: ANSI C function definitions shall be used when supported by the compiler.

JUSTIFICATION: ANSI C style function definitions explicitly supply the type of the function and its parameters. This allows the compiler to assist in type checking. ANSI C definitions also encourage the use of the "VOID" type, which is used to indicate a function has no return value and/or no parameters. This style of definition aids in code readability and understandability.

STANDARD: Function prototypes shall be used when supported by the compiler.

Example:

First the function declaration or "prototype"

```
int power(int Base, int Nth_power);
```

then the function definition.

```
int power(int Base, int Nth_power)
{
  int Current_power;      /* Result accumulator */
  for (Current_power = 1; Nth_power > 0; --Nth_power)
  {
    Current_power = Current_power * Base;
  }
  return Current_power;
}
```

JUSTIFICATION: The use of function prototypes will allow the compiler to detect errors that relate to the number of arguments or their types. This will enhance the integrity of the software at an early stage of development, thus saving software debug time.

NOTE: See Appendix B for examples of new ANSI C versus old-style function declarations and definitions.

4.8.4. Data Declarations

4.8.4.1. Constants

GUIDELINE: Character constants should not contain more than one character.

```
short Crlf = '\r\n';    /* bad - uses char constant    */
```

If the characters are simply being used as a string value, use a proper C string:

```
char Crlf[] = "\r\n";   /* good - uses string constant */
```

JUSTIFICATION: The differences in machine byte order may cause multi-character constants to differ either in numeric value or in character sequence. The multi-character constants are intrinsically non-portable.

GUIDELINE: Constants should be initialized with values of the same type.

JUSTIFICATION: Unexpected and unwanted type casting may occur.

GUIDELINE: Any constant, which might change during revision or modification, should be made obvious. Specifically, it should be given an uppercase name (via ***#define***). If it is only used in one file, it should be ***#define***'d at the head of that file; if used in multiple files, it should be ***#define***'d in an include file.

Example from *Plum*:

```
if (Index < 100)       /* bad example - uses magic number */

#define SIZE 100       /* good example - defines SIZE     */
if (Index < SIZE)      /* then uses SIZE                  */
```

JUSTIFICATION: Constants that are hard coded (otherwise known as "magic numbers", because they mysteriously appear with no explanation), are hard to locate when modifying the program. Furthermore, instances of "constant minus one" or "constant plus one" are even more elusive to the maintenance programmer. Also, the use of ***#define***'s increases readability.

4.8.4.2. Variables

STANDARD: The programmer shall guarantee that all variables, except loop counter variables, be explicitly set or initialized before use.

JUSTIFICATION: Relying on the compiler, operating system, or host system to preset variable values causes code to be non-portable and can cause subtle debugging problems.

STANDARD: Initializers shall be written with only one variable per line.

```
int DefaultMenuSelection = EDIT; /* Defaults to edit the current */

 /* record                      */

int MenuMode = NOVICE;          /* Initialize menu mode for     */

 /* users who are not           */

 /* experienced in using this   */
```

```
        /* product                 */
```

JUSTIFICATION: Readability is greatly enhanced by limiting one variable to a line.

GUIDELINE: Type specifiers, variable names, and descriptive comments should be aligned in a column:

```
    int ForkFlag;          /* Flag to determine which processing path   */
                                      /* to take (fork)                  */
    int MacroFlag = NO;/* Indicates when pre-processed macros exist */
```

JUSTIFICATION: These rules pinpoint the location of declarations, avoid conflicts of upper and lower case names and encourage documentation of meaningful variable names.

4.8.4.3. Data Types

GUIDELINE: Typedefs should be used to isolate machine dependent data types.

JUSTIFICATION: Since it may not be possible to eliminate all machine dependency from a program, isolating those occurrences will make changes easier. The use of well commented conditional compilation statements can be used to isolate machine dependent data. The use of conditional compilation will enhance the portability of the program regardless of machine specific data.

4.8.4.4. Pointers

STANDARD: Pointer conversions shall be explicitly cast.

JUSTIFICATION: Carelessness in the use of pointer types can needlessly cause code to be compiler dependent, machine dependent, or both. When all variables are typed and all conversions between types are specified, program correctness is enhanced and program verification is made easier.

STANDARD: Pointers should be initialized to a valid value prior to use. If a valid value is not known at the time of declaration, a NULL should be used.

JUSTIFICATION: The use of uninitialized pointers can cause unexpected results. This side effect will be hard to find when the function is being debugged.

4.8.5. Expressions

4.8.5.1. Mixed Mode Operations

STANDARD: Mixed mode operations, when used, shall be clearly identified and described using either type casting or comments within the source code.

JUSTIFICATION: The C language ability to enable a variable to interpret data as a different type can make software difficult to debug if the intent is not clearly understood. The commenting of different type conversions and evaluations will greatly enhance the understanding of the code.

GUIDELINE: Type casting should be used to clarify mixed mode operations.

JUSTIFICATION: For the same reason that commenting is required to clarify mixed mode operations, the use of the C language feature of casting will greatly enhance the ability to understand data type conversions.

4.8.5.2. Byte Ordering

STANDARD: Programs shall not depend upon the order of bytes within an integer or floating number.

JUSTIFICATION: For example, on machines such as VAX, the low-order byte of a short integer is stored in memory before the high-order byte; but on other machines, such as MC68000, the high-order byte is stored first.

```
    char  *Dot;           /* pointer to a byte          */

    short Point;          /* multi-byte variable         */

                     /* non-portable casting:        */

    Dot = (char *)&Point;  /* gives machine-dependent byte */
```

4.8.5.3. Byte Size

STANDARD: A program shall not rely on data size to truncate expressions to a specific number of bits.

Example from *Plum*:

```
        bit &= ~1;            /* good, ones complement operator  */
        bit &= 0177776;       /* bad turns off high bits         */
                              /* on 32 bit machine               */
```

JUSTIFICATION: The object of this standard is portability. Characters can be 8, 9, or 10 bits. Short integers can be 16, 18, 20, or 36 bits. Long integers can be 32, 36, or 40 bits. These depend on the hardware used.

4.8.5.4. Macros

GUIDELINE: Use subroutines/functions instead of complex user defined macros. System defined macros (e.g. “MIN”, type definition macros) are OK.

JUSTIFICATION: Functions provide automatic type checking and scoping of local variables, whereas macros do not.

GUIDELINE: Macro names should be fully capitalized. When writing macros, parentheses should be used around parameters in the replacement text to guard against precedence surprises.

Example 1:

```
    #define A_BAD_MACRO 1 + 1
    Row = Current_Point * A_BAD_MACRO
  maps into
    Row = Current_Point * 1 + 1
```

Issue: Row equals Current_Point + 1 not Current_Point * 2 as you probably wanted. It should have been coded as:

```
    #define A_GOOD_MACRO (1 + 1)
```

Example 2:

```
    #define SQUARE(x)  x*x
```

When invoked as SQUARE(z+1), the results are not what would be expected. However, just surrounding the arguments with parentheses like the following is not enough:

```
    #define SQUARE(x)  (x*x)
```

This results in the same problem. The macro should be defined as follows to ensure proper operation:

```
    #define SQUARE(x)  ((x)*(x))
```

JUSTIFICATION: The problems with precedence and side effects cause macros to behave differently from functions. The user must be protected when possible and otherwise warned. Problems with lack of type safety, disappearance in the debugger, ambiguity when the arguments are evaluated multiple times can be introduced by macro usage in arithmetic operations and should be avoided.

4.8.6. Control Constructs

4.8.6.1. Statements

STANDARD: Each statement shall begin on a separate line.

JUSTIFICATION: This aids in making the code readable and understandable.

STANDARD: Braces shall always be used around control statement clauses, even if the clause is a single statement.

JUSTIFICATION: A common subtle error is a statement which is indented to the same level as the clause of a control statement, but which is not part of the statement:

```
if (Requested_Value == TERMINATE)
  printf("Terminated abnormally");
  exit(ABNORMAL);
```

This example appears at first glance to exit abnormally only if the request is to terminate. A closer examination shows it will exit abnormally every time regardless of the request. This problem can be avoided with the use of braces. This can occur during maintenance when statements are added.

GUIDELINE: The nesting of statements should be limited to five (5) levels. Level 0, or not nested, is the topmost level under the function declaration.

Example:

The following example shows nesting levels:

```
void Nesting_Level_Example (int x, int y)
{
  int index;          /* loop control variable */
  /* Code at this level is not nested (level 0) */
  if ((x > 0) && (y > x))
  {  /* Level 1 */
    for (index = x; index <= y; index++)
    {  /* Level 2 */
       ...
    }
  }
  else
  {  /* Level 1 */
     ...
  }
}
```

JUSTIFICATION: Excessive levels of nesting tends to generate code which is difficult to understand and maintain.

GUIDELINE: Each statement that is part of the body of a C control structure (*if*, *while*, *do while*, *for*, and *switch*) should be indented from the margin of its controlling statement. The same rule applies to function definitions, structure definitions, union definitions and aggregate initializers.

JUSTIFICATION: Consistent application of indentation improves readability and maintainability.

GUIDELINE: The layout of control structures should follow the rule above regarding indentation, and should further contain one of the two following styles, (style one being the preferred style).

The two styles are illustrated below.

Example: Style one

```
if (Start_Point == ZERO)
{
  ++Start_Point;
}
```

Example: Style two

```
if (Start_Point == ZERO) {
  ++Start_Point;
}
```

JUSTIFICATION: The choice of a layout standard and paying attention to detail will pay off in readability and in the ability to create automated text handling tools like editor scripts, search commands, and "beautifiers". The job of maintenance programming will become easier if a standard coding style is used.

NOTE: See Appendix C for examples of all control structures.

STANDARD: The *goto* statement shall be used only on a case-by-case basis as approved by program management where required to meet specific execution time requirements or memory constraints.

JUSTIFICATION: The *goto* statement restriction exists for the empirical reason that its use is highly correlated with errors and hard-to-read code.

STANDARD: Each *goto* statement shall be accompanied by the following:

i. Comments placed near the *goto* statement to document the applicable constraints.

ii. Comments placed near the statement receiving control to document the origin of the transfer of control.

JUSTIFICATION: Such comments save much time in reading and understanding the code.

STANDARD: The *goto* statement shall not be used to transfer control into loops.

JUSTIFICATION: Algorithms should be expressed in structures that facilitate checking the program against the structure of the underlying process.

STANDARD: The *default* label shall be used in all *switch* statements.

JUSTIFICATION: Providing a *default* label for every *switch* statement will ease the maintenance stage. The *default* statement will ensure that all values passing through the *switch* statement are processed. Possible errors will be caught in the unit test stage and not in later stages.

STANDARD: When using the *switch* statement, all cases with processing should be terminated with a break statement. When a case is not terminated with a break statement, a comment should be added to explain the reason for "fall through".

JUSTIFICATION: Terminating a switch case with a break statement ensures a switch statement executes as expected. "Falling through from one case to another is not robust, being prone to disintegration when the program is modified" (*K&R*). However, there may be instances where "falling through" case statements allow a more compact design. In these instances, comments explaining which cases are to be "fallen through", and why, aid in understanding and debugging.

GUIDELINE: In the test expression of *while*, *for*, *do while*, or *if* control structures, for non-boolean values, the comparison should be written explicitly rather than relying upon the default comparison to zero or non-zero. The comparison of a pointer to *null* should be written as an explicit comparison.

Example:

```
/* bad, no explicit evaluation for the number of bytes read */
while (fgets(buffer, BUFFER_SIZE, stdin))
{
  process(buffer);
}
/* good, explicit evaluation for the number of bytes read */
while (fgets(buffer, BUFFER_SIZE, stdin) != 0)
{
  process(buffer);
}
```

JUSTIFICATION: Explicit comparisons in test expressions will enhance the readability of software.

4.8.6.2. Loop Constructs

STANDARD: Loop variables which control the execution and exit conditions of a *for* loop shall not be altered in the context of the *for* loop.

Example:

```
for (Row = 0; Row < Dot; Row++)      /* Bad              */
{                                /* increments test   */
  Dot++;                            /* condition in loop */
}
for (Row = 0; Row < Dot; Row++)      /* Bad              */
{                                /* Explicit change   */
  Dot = 3;                          /* to test condition */
}
```

JUSTIFICATION: Both examples above show that altering the loop variable or loop condition variable will produce different effects than were originally intended. This greatly increases the level of understanding required to follow the flow of the software.

4.9. Error Handling

STANDARD: The following practices shall be employed:

- Check for error return values, even from functions that "can't" fail. Consider that close() and fclose() can and do fail, even when all prior file operations have succeeded. Write your own functions so that they test for errors and return error values or abort the program in a well-defined way. Include a lot of debugging and error-checking code and leave most of it in the finished product.

- Use the *assert* facility to insist that each function is being passed well-defined values, and that intermediate results are well-formed.

- Include the system error text for every system error message.
- Check every call to malloc or realloc unless you know your versions of these calls do the right thing. You might want to have your own wrapper for these calls, so you can do the right thing always and developers don't have to make memory checks everywhere.

JUSTIFICATION: Consistent error handling makes way for more robust code. Use of asserts() is invaluable to trap software bugs in the development and unit testing phase.

4.10. Common Libraries

GUIDELINE: There's no excuse for writing code which already exists in a common library. Not only will the standard library's code be tested, often it will be more efficient, and will certainly be more familiar to your fellow programmers. Some notes on using particular functions:

> *gets*: Never use this. Use fgets instead so that you can be sure that you don't overflow your buffer.
>
> *malloc*: Don't cast the return value of calls to malloc, It has type void *, so it will be compatible with anything.

JUSTIFICATION: Encourages code reuse and consistency.

4.11. Use of Standard Constants

STANDARD: Numerical constants shall not be coded directly. The #define feature of the C preprocessor shall be used to give constants meaningful names. Defining the value in one place also makes it easier to administer large programs since the constant value can be changed uniformly by changing only the #define. The enumeration data type is a better way to declare variables that take on only a discrete set of values, since additional type checking is often available. At the very least, any directly-coded numerical constant must have a comment explaining the derivation of the value. Use standard mathematical and geophysical constants (e.g. PI).

JUSTIFICATION: Symbolic constants make the code easier to read. Consistent use of standard constants benefits module interoperability.

4.12. Efficient Use of Memory

STANDARD: Use dynamic allocation of memory wherever possible. Do not allocate memory for local variables until they are used in a subprogram, and deallocate the memory for a local variable as soon as its use in the program is finished. This is especially important when handling large, multi-dimensional arrays.

Example: TBS

JUSTIFICATION: Efficient use of memory as well as protection against hard to find memory leaks and violations, leading to unexpected, and sometimes hard to reproduce results.

4.13. Interoperability

4.13.1. C/Fortran Interoperability

When calling C from Fortran and vice versa, several issues must be addressed. These may vary from compiler to compiler so verification with your compilers documentation is recommended. In general follow these guidelines:

- Data types must match between languages.
    - i.e. (Fortran) real*4 = (C) float
    - i.e. (Fortran) integer*4 = (C) int
    - Not all types have a match.
- C is case sensitive, Fortran is not (all routine names get converted to lowercase when compiled).
    - Always use lower case names for routines to avoid problems.
- Fortran appends an "_" character to all routine names after compilation.
    - Add the "_" character to your called routine name in C code (see example).
    - Add the "_" character to your C routine name when called from Fortran (see example).
    - NOTE: IBM needs to set compiler flag for "_"
- Fortran passes arguments by reference, C passes by value or reference.
    - Use the C operators "&" and "*" accordingly when passing arguments.
- Use the "extern" command in C to declare external Fortran routines

- Use the “external” command in Fortran to declare external C routines.
- Avoid passing global data between languages, such as common blocks.

Example of Fortran calling C:

```
program f_code
   implicit none
   external c_code
   integer :: length
   integer :: width
   integer :: area
   length = 15
   width = 10
   area  = 0
   call c_code(length, width, area)
   write(*,*) “Area = “,area
end

void c_code_ (int *length, int *width, int *area) {
   *area = (*length)*(*width);
}
```

Example of C calling Fortran:

```
#include<stdio.h>
extern void f_code_ (int *, int *, int *);
void main () {
   int length;
   int width;
   int area;
   length = 15;
   width = 10;
   area = 0;
   f_code_ (&length, &width, &area);
   printf(“Area = %d\n”,area);
}

subroutine f_code (length, width, area)
   integer :: length
   integer :: width
   integer :: area
   area = length*width
end subroutine
```

4.13.2. C/++ Interoperability

In order to be backward compatible with dumb linkers C++'s link time type safety is implemented by encoding type information in link symbols, a process called name mangling. This creates a problem when linking to C code as C function names are not mangled. When calling a C function from C++ the function name will be mangled unless you turn it off. Name mangling is turned off with the extern "C" syntax. If you want to create a C function in C++ you must wrap it with the above syntax. If you want to call a C function in a C library from C++ you must wrap in the above syntax. Here are some examples:

Calling C Functions from C++

```
extern "C" int strncpy(...);

extern "C" int my_great_function();

extern "C"

{

  int strncpy(...);

  int my_great_function();

};
```

Creating a C Function in C++

```
extern "C" void

a_c_function_in_cplusplus(int a)

{

}
```

__cplusplus Preprocessor Directive

If you have code that must compile in a C and C++ environment then you must use the __cplusplus preprocessor directive. For example:

```
#ifdef __cplusplus

extern "C" some_function();

#else

extern some_function();

#endif
```

4.14. Documentation

4.14.1. Headers

STANDARD: Compilation unit header shall contain the following information:

1. NAME
2. FUNCTION
3. DESCRIPTION
4. REFERENCE
5. CALLING SEQUENCE
6. INPUTS
7. OUTPUTS
8. DEPENDENCIES
9. RESTRICTIONS
10. HISTORY

JUSTIFICATION: Properly documented software can save many hours of debug time. This will provide a central point for others to review the functionality of the software without sifting through lines and lines of code. Maintenance will become cost-effective when properly documented code exists.

4.14.2. Comments

STANDARD: Comments shall not be used to disable executable statements.

JUSTIFICATION: Code disabled by comments will only confuse the issue of what should or does exist in a function. The proper configuration control of software revisions will provide the ability to quickly modify software to previous versions, thus eliminating the need to keep commented-out code.

STANDARD: The characters **/*** shall introduce a comment, while the characters ***/** terminate the comment.

JUSTIFICATION: Compiler dependent comment symbols are not portable.

STANDARD: Comments shall not be nested.

JUSTIFICATION: Such nesting is not necessary, and is a hazard during maintenance. Not all compilers handle this condition in the same manner.

GUIDELINE: Each logical grouping of statements should be made more readable by a comment prior to the block. A block comment should be indented at the same level as the statements, or begin at the left margin.

JUSTIFICATION: Such comments prevent having to scroll back and forth to the function block *header*, aid in a quicker understanding of the code, and are relatively easy to maintain.

GUIDELINE: Block comments, which consist of several lines of text, should be consistently formatted in a style selected for the software program. Commonly used styles include:

i. Boxed comments: Each line contains the characters /* and the characters */

which are all vertically aligned with the other lines of the comment.

ii. Bracketed comments: The comment is opened by a line containing only the characters /* and closed at the end of the comment by a line containing only the characters */ which are vertically aligned with the opening /* characters.

iii. Flagged comments: A long, distinctive, repeating character string (the flag) is used on the first and last lines of the comment. The characters /* precede the flag on the first line of the comment and the characters */ are appended to the flag which ends the comment.

JUSTIFICATION: Since comments have their greatest value late in the program life cycle, they should be organized with a view toward making comments stand out from the code and encouraging future maintainers to keep them current.

GUIDELINE: When comments, other than block comments, exceed one line and begin on lines with other program elements, each comment line should contain delimiters.

JUSTIFICATION: Multiple line comments not properly delimited often cause compiler errors.

GUIDELINE: All variable declarations should have a descriptive comment following the semicolon, with one variable per line.

JUSTIFICATION: Commenting is the only way that the author has of explicitly identifying the intent of his code. The proper use of comments will make the code easier for others to maintain. One practical test of the amount of commenting needed is: Can the reviewer understand the function without detailed coaching from the author?

NOTE: See Appendix A for comment examples.

4.15. Grandfathering

This section explains what can be excluded from these Programming Standards and Guidelines.

4.15.1. COTS

Commercial Off The Shelf (COTS) software currently in use is grandfathered and does not have to comply with the standards and guidelines documented in this TD. If adding additional functionality to COTS software, consider implementing the standards and guidelines documented in this TD wherever possible.

4.15.2. Reuse

Software reuse from a common Product Line baseline is grandfathered and does not have to comply with the standards and guidelines documented in this TD.

STAR-unique Software Components that are developed for use with the reuse software shall follow the standards and guidelines in this TD.

STAR-unique Software Units that are developed to integrate with reuse software shall follow the standards and guidelines in this TD, if possible, given the reuse software architecture and reuse software standards involved.

Newly developed STAR software that is deemed to be generic in nature and suitable for addition to the reuse software baseline will follow the standards and guidelines established for the reuse software.

APPENDIX A. EXAMPLE COMMENTS

```
#include <xref.h>        /* XREF data structures and constants   */
#include <math.h>        /* Standard C math functions            */

/*****************************************************************/
/*                                                               */
/*  Start of global function and an example comment.             */
/*                                                               */
/*****************************************************************/

char get_zone (long coordinate, XREF_ZONE *zone_pointer,
               int *zone_index, char *zone)
{
  char status_flag = NO_ERRORS;   /* Function return status */

  /*
  ** Check pointer before processing, NULL pointers are not an error.
  */

  if (zone_pointer != NULL)
  {
    *zone_index = (int) ceil ((coordinate - zone_pointer->offset) /
                              (double) zone_pointer->width);

    if ((*zone_index < lower_limit) || (*zone_index > upper_limit))
    {
      status_flag = WARNING;
    }
    else
    {
      if (zone_pointer->format_type == ALPHA_FORMAT)
      {
        zone [0] = 'A' + (*zone_index - 1);
        zone [1] = '\0';
      }

      else
      {
        itoa (*zone_index, zone);
      }

    } /* END IF zone is out of range ELSE ... */

  } /* END IF zone pointer is not NULL ... */

  return (status_flag);
} /*** END GET_ZONE ***/
```

APPENDIX B. EXAMPLE FUNCTION DECLARATIONS AND DEFINITIONS

Old Style Declaration

```
STRING_POINTER *Insert_string();

Print_string();
```

ANSI C Style Declaration

```
STRING_POINTER *Insert_string( char *String_Address,
                               unsigned int Length);

void Print_string(STRING_POINTER String);
```

Old Style Definition

```
STRING_POINTER *Insert_string(String_Address, Length)
char *String_Address;
unsigned int Length;
{
 ...
}

Print_string(String)
STRING_POINTER String;
{
 ...
}
```

ANSI C Style Definition

```
STRING_POINTER *Insert_string(char *String_Address, unsigned int Length)
{
 ...
}

void Print_string(STRING_POINTER String)
{
 ...
}
```

APPENDIX C. EXAMPLE CONTROL STRUCTURES

In each case, style 1 is preferred.

DO WHILE

Style 1

```
do
{
   statement1;
   statement2;
}
while (condition);
```

Style 2

```
do {
   statement1;
   statement2;
}
while (condition);
```

FOR

Style 1

```
for (expr1; expr2; expr3)
{
   statement1;
   statement2;
}
```

Style 2

```
for (expr1; expr2; expr3){
   statement1;
   statement2;
}
```

IF

Style 1

```
if (condition)
{
   statement1;
   statement2;
}
```

Style 2

```
if (condition){
   statement1;
   statement2;
}
```

IF ELSE

Style 1

```
if (condition)
{
   statement1;
   statement2;
}
else
{
   statement3;
   statement4;
}
```

Style 2

```
if (condition){
   statement1;
   statement2;
}
else {
   statement3;
   statement4;
}
```

SWITCH

Style 1

```
switch (variable)
{
   case condition1:
      statement1A;
      break;
   case condition2:
      statement2A;
      break;
   default:
      statement3A;
      break;
}
```

Style 2

```
switch (variable){
   case condition1:
      statement1A;
      break;
   case condition2:
      statement2A;
      break;
   default:
      statement3A;
      break;
}
```

WHILE

Style 1

```
while (condition)
{
   statement1;
   statement2;
}
```

Style 2

```
while (condition){
   statement1;
   statement2;
}
```

APPENDIX D. INCLUDE FILE USAGE

The following example demonstrates proper usage and nesting of include files:

A program-wide *header* file, "global.h" is used to define globally used definitions. Note that the entire file is protected by an ***#ifndef*** to protect against multiple includes of this file, since it is highly likely that this file will end up being multiply included in any given 'C' source file.

global.h:

```
#ifndef _GLOBAL_H

#define NUMBER_OF_TARGETS   50

struct vector_type              /* position vector        */
{
  unsigned int x_position;      /* x position from origin */
  unsigned int y_position;      /* y position from origin */
  unsigned int z_position;      /* z position from origin */
}

#define _GLOBAL_H
#endif
```

A specific *header* file, "position.h", defines prototypes for the source file "position.c". The function prototypes need the vector_type structure definition, so the "global.h" *header* file is included.

position.h:

```
#include "global.h"

struct vector_type get_target_position(unsigned int target_id);
void initialize_target_position(struct vector_type target_position);
```

The 'C' source file "position.c" requires the vector_type structure as well as the "position.h" *header* file. Both these includes are used.

position.c:

```
#include "global.h"
#include "position.h"

struct vector_type target_table[NUMBER_OF_TARGETS];
                                /* location for each target */

struct vector_type get_target_position(unsigned int target_id)
{
  .
  .
  .
}
```

The following implementations would be considered bad practice and should be avoided:

a) Omitting the include of "global.h" in "position.h" by assuming that all 'C' source files would automatically include "global.h" before including "position.h".

b) Omitting the include of "global.h" in "position.c" by noting that "global.h" is already included by virtue of being included in "position.h".

APPENDIX E. "C" CODING STANDARDS - QUICK REFERENCE

The following quick reference was provided by Walter Wolf (STAR).

Compilation unit elements and order:

1. Compilation unit *header*.
2. Include files (#include).
3. Declarations (those not in Include files or functions).
4. Functions.

Functions shall be organized as follows:

1. Function declaration.
2. Function *header* (optional in single function compilation units).
3. Compound statement.

Compilation unit header shall contain the following information:

1. NAME
2. FUNCTION
3. DESCRIPTION
4. REFERENCE
5. CALLING SEQUENCE
6. INPUTS
7. OUTPUTS
8. DEPENDENCIES
9. RESTRICTIONS
10. HISTORY

Developer is responsible to keep file and function headers up-to-date**.**

Each statement shall begin on a separate line.

Lines within a compilation unit should fit a listing (or screen) width of 80 characters.

Variables shall be explicitly set or initialized before use.

Initializers shall be written with only one variable per line.

Configuration parameters should be placed in a separate config. file rather than hard coded.

Formatting style shall be defined and used consistently to enhance readability throughout a program (e.g., alphabetic case, blocking with blank lines, parentheses and indentation).

Names shall be mnemonically descriptive, given limitations within any C language implementation.

Multiple line expressions should be broken in a manner that enhances the readability of the expression.

Parentheses and spaces shall be used to help clarify evaluation of logical and arithmetic expressions.

Braces shall always be used around control statement clauses.

Include files shall contain only definitions, declarations, macros, function prototypes, comments, and conditional compilation statements that are needed by more than one compilation unit.

ANSI C function definitions shall be used when supported by the compiler.

Function prototypes shall be used when supported by the compiler.

Exit points of a function, not occurring as the last executable statement, shall contain a comment that begins with the word: *EXIT*.

The *#ifdef* directive shall not be used to disable unused code. The directive is to be used for conditional compilation (e.g., debug code, platform dependencies, compiler dependencies).

Constant should be given an uppercase name (via ***#define***). If it is only used in one file, it should be ***#define***'d at the head of that file; if used in multiple files, it should be ***#define***'d in an include file.

Pointers:

- Pointer conversions shall be explicitly cast.
- Pointers should be initialized to a valid value (or NULL) prior to use.

Mixed mode operations, when used, shall be clearly identified and described using either type casting or comments within the source code.

STANDARD: Programs shall not depend upon the order of bytes within an integer or floating number.

STANDARD: A program shall not rely on data size to truncate expressions to a specific number of bits.

Comments:

- The characters **/*** shall introduce a comment, while the characters ***/** terminate the comment.
- Comments shall not be used to disable executable statements.
- Comments shall not be nested.

Loop variables which control the execution and exit conditions of a *for* loop shall not be altered in the context of the *for* loop.

Goto Statement:

- The *goto* statement shall be used only on a case-by-case basis as approved by program management where required to meet specific execution time requirements or memory constraints.
- The *goto* statement shall not be used to transfer control into loops.
- Each *goto* statement shall be accompanied by the following:
  - Comments placed near the *goto* statement to document the applicable constraints.
  - Comments placed near the statement receiving control to document the origin of the transfer of control.

Switch statements

- The *default* label shall be used in all *switch* statements.

- All cases with processing should be terminated with a break statement. When a case is not terminated with a break statement, a comment should be added to explain the reason for "fall through".

Error Handling

- Check for error return values, even from functions that "can't" fail. Consider that close() and fclose() can and do fail, even when all prior file operations have succeeded. Write your own functions so that they test for errors and return error values or abort the program in a well-defined way. Include a lot of debugging and error-checking code and leave most of it in the finished product.
- Use the assert facility to insist that each function is being passed well-defined values, and that intermediate results are well-formed.
- Include the system error text for every system error message.
- Check every call to malloc or realloc unless you know your versions of these calls do the right thing. You might want to have your own wrapper for these calls, so you can do the right thing always and developers don't have to make memory checks everywhere.

Standard Mathematical and Geophysical Constants shall be used (e.g. PI).

Dynamic allocation of memory shall be used wherever possible. Do not allocate memory for local variables until they are used in a subprogram, and deallocate the memory for a local variable as soon as its use in the program is finished.

STANDARD: Follow compiler documentations and options for mixed code programming.

STANDARD: All code should be subjected to a code checker, such as Lint, and the output presented at subsequent code reviews for official acceptance after every change. Lint is available on most operating systems.

Guidelines:

- Type casting should be used to clarify mixed mode operations.

- The indentation of multiple line expressions should enhance the readabil-ity of the expression.
- For each level of indentation, use three spaces. The number of spaces used should be applied to the entire program.
- Upper case should be used for user ***#define***'d identifiers and Macro names.
- Mixed case or underscores should be used for function and variable names.
- Names should not resemble C reserved words or implementation supplied function names.
- Each function shall contain a maximum of 150 lines of code**.**
- Each program unit should contain no more than 200 executable statements.
- Each function should contain a single exit point as the last executable statement. If the exit point is not the last executable statement, the last line of the function should be a comment identifying where the exit point(s) are located within the function.
- Declaration sections should be ordered
  - Constants (#define).
  - Type definitions (typedefs, unions, structs)
  - Global variable declarations of simple types (int, long, float, char, etc.).
  - Global variable declarations of compound types (typedefs, structs, unions, and arrays).
  - Non-global (static) declarations
- Any file that uses definitions provided by another file should explicitly include that file.
- Use ***#ifdef*** statements in include files to prevent redefinition of values.
- Programs should take advantage of include files which contain program wide standard definitions (#defines, typedefs, macros, etc.).
- Include files should not contain data storage.
- Character constants should not contain more than one character.
- Constants should be initialized with values of the same type.

- Type specifiers, variable names, and descriptive comments should be aligned in a column.
- Avoid the use of raw C types like int, long, float, double when using data that might be written to disk. For example, the sizes of int and long are machine dependent. On 32 bit machines int's and long's are 32 bits, but on 64 bit processors an int can be either 32 or 64 bits and a long 64 bits, depending on the processor. For portability reasons and consistent numerical results use typedefs for the basic raw C types.
- One should write functions and not macros.
- Macro names should be fully capitalized. When writing macros, parentheses should be used around parameters in the replacement text to guard against precedence surprises.
- The nesting of statements should be limited to five (5) levels. Level 0, or not nested, is the topmost level under the function declaration.
- Each statement that is part of the body of a C control structure (*if*, *while*, *do while*, *for*, and *switch*) should be indented from the margin of its controlling statement. The same rule applies to function defini-tions, structure definitions, union definitions and aggregate initializers.
- The layout of control structures should follow the rule above regarding indentation, and should further contain one of the two following styles, (style one being the preferred style).
- In the test expression of *while*, *for*, *do while*, or *if* control structures, for non-boolean values, the comparison should be written explicitly rather than relying upon the default comparison to zero or non-zero. The comparison of a pointer to *null* should be written as an explicit comparison.
- There's no excuse for writing code which already exists in a common library. Not only will the standard library's code be tested, often it will be more efficient, and will certainly be more familiar to your fellow programmers. Some notes on using particular functions:
- Recursive routines should be avoided on efficiency grounds.
- Each logical grouping of statements should be made more readable by a comment prior to the block. A block comment should be indented at the same level as the state-ments, or begin at the left margin.

- Block comments, which consist of several lines of text, should be consistently formatted in a style selected for the software program.
- When comments, other than block comments, exceed one line and begin on lines with other program elements, each comment line should contain delimiters.
- All variable declarations should have a descriptive comment following the semicolon, with one variable per line.
- Include a reference to the reason for code changes.

END OF DOCUMENT