**SPSRB Common Standards Working Group**

**Standards, Guidelines and Recommendations for**

**Writing C Code**

**Thomas S. King**

**Dell Systems for NOAA/NESDIS/STAR**

**Liqun Ma**

**NOAA/NESDIS/OSDPD**



**Version 1.0**

**September, 2010**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**VERSION NUMBER IDENTIFIER**

The document version number which also corresponds to the Document Control Number (DCN) identifies whether the document is a working copy, final, revision, or update, defined as follows:

**Working Copy or Draft:** a document not yet finalized or ready for distribution; sometimes called a draft. Use 0.1A, 0.1B, etc. for unpublished documents.

 **Final Copy:** the first definitive edition of the document after it passes through the drafting stage. This first edition is always identified as Version 1.0.

**Revision:** an edition with minor changes from the previous edition, defined as changes affecting less than one-third of the pages in the document. The version numbers for revisions 1.1 through 1.9, 2.1 through 2.9, and so forth. After nine revisions, any other changes to the document are considered an update. A revision in draft, i.e. before being re-baselined should be numbered as 1.1A, 1.1B, etc.

**Update:** an edition with major changes from the previous edition, defined as changes affecting more than one-third of the pages in the document. The version number for an update is always a whole number (Version 2.0, 3.0, 4.0, and so forth).

**DOCUMENT HISTORY**

**DOCUMENT REVISION LOG**

The Document Revision Log identifies the series of revisions to this document since the baseline release. Please refer to the above page for version number information.

|  |
| --- |
| **DOCUMENT TITLE: General Programming Principles and Guidelines** |
| **DOCUMENT CHANGE HISTORY** |
| **Revision No.** | **Date** | **Revision Originator Project Group** | **CCR Approval # and Date** |
| 1.0 | September 2010 | Initial Release by CSWG | Pending |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**LIST OF CHANGES**

Significant alterations made to this document are annotated in the List of Changes table.

|  |
| --- |
| **DOCUMENT TITLE: General Programming Principles and Guidelines** |
| **LIST OF CHANGE-AFFECTED PAGES/SECTIONS/APPENDICES** |
| **Version Number** | **Date** | **Changed By** | **Page** | **Section** | **Description of Change(s)** |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

**TABLE OF CONTENTS**

1. **Introduction ………………………………………………….**
	1. Programming Standards and Guideline Definitions …………….......
	2. Key Word Definitions ……………………………………………….
	3. Reference Documents ….....…………………………………………
2. **Formatting …………………………………………………...**
	1. Organization …………………………………….…………………...
	2. Size ….………………………...……………………………..….…...
	3. Naming Convention ………………………………………...……….
	4. Compound Expressions ……………………………………………...
		1. Example 1: Use of Parentheses ….…………………………………
		2. Example 2: Use of Parentheses and Line Breaks …………………..
		3. Example 3: Multiple Line Expressions …………………………….
3. **Declarations and Statements …..……………………...…….**
	1. Basic Declarations …………………………………………………...
	2. Conditional Compilation …………………………………………….
	3. Include Files …………………………………………………………
	4. Function Declarations and Definitions ……………………………....
		1. Example 4: Function Prototypes …………………………………..
	5. Data Declarations …………………………………………………….
		1. Constants …………………………………………………...………
			1. Example 5: Use of Character Constants ………………….………...
			2. Example 6: Use of Revised Constants from *Plum* …………………
		2. Variables ………………………………………………………...…
			1. Example 7: Correct Use of Initializers ………………………….....
			2. Example 8: Correct Use of Flags and Comments ………………....
		3. Data Types ………………………………………………...……….
		4. Pointers …………………………………………………………….
		5. Expressions …………………………………..…………………….
			1. Mixed Mode Operations ……………………………………….…..
			2. Byte Ordering …………………………………………………..…..
				1. Example 9: Byte Ordering …………………………………………..
			3. Byte Size ………………………………………………………..…..
				1. Example 10: Byte Size ……………………………………………….
			4. Macros …………………………………………………………..…..
				1. Example 11: Improper Macro Usage …………………………………
				2. Example 12: Macro Definition ……………………………………….
	6. Statements ………………………………………………...………….
		1. Example 13: Braces in Statements ………………………………....
		2. Example 14: Nesting Levels ..…...………………………………....
		3. Example 15: Styles of Control Structures..…...………………….....
		4. Example 16: Comparison Written Explicitly..…...………………....
	7. Use of Standard Constants …………………………...…………...….

**7**

8

8

9

**10**

10

10

10

11

11

11

11

**12**

12

12

12

13

13

14

14

14

15

15

15

15

16

16

16

16

16

16

16

17

17

17

17

18

18

19

19

20

20

 **TABLE OF CONTENTS (Continued)**

1. **Readability and Documentation ………………….………...**
	1. Readability …………………………………….……………………..
		1. Example 18: Mixed Case Underscores …..….……………….
	2. Comments …………………………………………………………...
2. **Libraries and Memory …………………….………………...**
	1. Common Libraries …………………………….…………………….
	2. Efficient Use of Memory …...……………………………..….……..
3. **Interoperability ……………………………………………...**
	1. C/Fortran Interoperability …………………….……………………...
		1. Example 19: Fortran Calling C ….…………………………………
		2. Example 20: C Calling Fortran ….…………………………………
	2. C/C++ Interoperability ………...……………………………..….…...
		1. Example 21: C Syntax Wrapping ….……………………………..
		2. Example 22: Creating a C Function in C++ ……………………….
		3. Example 23: Preprocessor Directive ….……………………………
4. **Errors ………………………………………………………...**
	1. Error Handling ……………………………….…………………........
5. **Items to Avoid in C ..………………………………………...**
	1. Library Functions to Avoid ………………………………………….
	2. Manage Interprocess Communications (IPC) Resources …………..
	3. Grandfathering ………………………………………………...…….
		1. COTS ………………………...….…………………………………
		2. Reuse ……………………………………………………………….

**21**

21

22

22

**22**

22

22

**23**

23

24

24

24

25

25

25

**25**

25

**26**

26

26

27

27

28

# **1. INTRODUCTION**

The National Environmental Satellite Data Distribution Service (NESDIS) develops and implements algorithms that transform environmental satellite images of the Earth into meaningful environmental data which are then employed in a full-time operational setting. In the past, software developed within NESDIS was created by the differing entities throughout the service, each creating code to fulfill various research, operational and archival needs. This meant software was written in various programming languages and idiosyncratic styles, moreover suffering from a lack of coordinating documentation in most cases. The resulting software is consequently often costly to maintain as the source code may have been mislaid, the code may be difficult to read and understand, documentation may be inadequate, or the original developers may no longer be able to maintain their code.

The purpose of developing common software programming standards is to reduce the cost of the software lifecycle and streamline the algorithm implementation process. This follows a trajectory from initial research and software development to operational use and finally through to divestiture and retirement where costs accumulate throughout the lifecycle. Implementation of these Satellite Products and Services Review Board (SPSRB) approved coding standards will shift costs away from operations and maintenance as the problems are resolved upstream. Promoting the accountability of the developers and scientists to create standardized software programs will benefit NESDIS as a whole. Higher front-end expenditure will be repaid in the form of lower operational and maintenance costs over subsequent years. It is intended that the implementation expenses of the common software standards will be funded through the Office of Systems Development (OSD) Product System Development and Implementation (PSDI) process, and must be included in relevant budgets and projects plans when applying for PSDI funds.

Having common programming standards used by all SPSRB stakeholders will aid in cross-organization communication and implementation of codes. It will also produce a software catalog that:

* Is robust
* Is readily portable (platform independent)
* Is modular and reusable
* Is inexpensive to implement and maintain operationally
* Is written in a widely used and supported language
* Has a common look and structure
* Adheres to best programming practices
* Is well documented
* Is easily readable and understandable
* Behaves in a standard manner (exception handling, file input/output)
* Uses common shared libraries

**1.1 Programming Standards and Guideline Definitions**

It is recognized that certain stylistic suggestions which make code easier to read (e.g. lining up attributes, or using all lower case or mixed case) are subjective and therefore should not have the same weight as techniques and practices that are known to improve code quality. For this reason, the standards within documents produced by the SPSRB Common Standards Working Group are divided into three components; Standards, Guidelines and Recommendations (SGRs):

***Standard***:Aimed at ensuring portability, readability and robustness. Compliance with this category is mandatory.

***Guideline***:Good practices. Compliance with this category is strongly encouraged. The case for deviations will need to be argued by the programmer.

***Recommendation***:Compliance with this category is optional, but is encouraged for consistency.

These three standards will thus be found in the above format throughout this document, indicating the weight of a particular standard. If possible, all standards, guidelines and recommendations should be followed when programming. Else, programmers should include these components whenever possible, keeping in mind their respective weight. Please refer to these definitions as needed.

**1.2 Key Word Definitions**

These words and phrases are vital to the comprehension of this document. These are language-specific and general terms with which a user may not be readily familiar. Listed below in alphabetical order are the terms specified by the CSWG to be helpful to users and may be referenced as necessary.

***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.

**1.3 Reference Documents**

***Programming Languages - C*** (formerly ANSI/ISO/IEC 9899-1999). American National Standards Institute, Second Edition, 2000. Found at (http://www.nirvani.net.nyud.net:8090/docs/ansi\_c.pdf)

1. ***TD-12.1.3: C programming Standards and Guidelines.*** Version 2.0 (A NOAA/NESDIS/STAR Training Document)

**2. Formatting**

**2.1 Organization**

***Standard****: Compilation units shall be organized as follows:*

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

***Standard****: Functions shall be organized as follows:*

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

***Standard***: *Each function shall contain a maximum of 200 lines of code****.***

**2.2 Size**

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

**2.3 Naming Conventions**

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

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

**2.4 Compound Expressions**

 ***Standard***: *The evaluation of logical and arithmetic expressions shall be clarified through the use of parentheses.*

### **2.4.1 Example 1: Use of Parentheses**

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

 Bad example of parentheses usage:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if (flags & BIT1 && state ==PARSE)

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Good example of parentheses usage:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 if ((flags & BIT1) && (state == PARSE))

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### **2.4.2 Example 2: Use of Parentheses and Line Breaks**

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

Bad example of grouping by use of parentheses and line breaks:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

value = option\_base + OPTION\_OFFSET + array\_value \*

 ENTRY\_SIZE + VALUE\_OFFSET;

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Good example of grouping by use of parentheses and line breaks:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

value = (option\_base + OPTION\_OFFSET) +

 (array\_value \* ENTRY\_SIZE) +

 VALUE\_OFFSET;

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

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

###

### **2.4.3 Example 3: Multiple Line Expressions**

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

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

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**3. Declarations and Statements**

 **3.1 Basic Declarations**

***Guideline***:  *Declarations should be ordered in one of the two following ways:*

1. Creating logical groups based on usage.
2. Following the prescribed basic declaration format (see next guideline).

***Guideline***:  *The prescribed basic declaration format is as follows:*

* 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

**3.2 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).

**3.3 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.*

***Standard***: #include *statements shall not use absolute path names.*

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

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

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

***Guideline****: Include files should not contain data storage.*

**3.4 Function Declarations and Definitions**

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

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

### **3.4.1 Example 4: Function Prototypes**

First write the function declaration or "prototype":

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int power(int Base, int Nth\_power);

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

followed by 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;

}

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***Standard****: The return value of all functions shall be explicitly typed*.

Functions that do not return values shall be declared to be of type void.

***Standard***: *Each argument shall be explicitly declared in the argument list, with a name and a type.*

If a function does not require any arguments its argument list shall be declared void.

***Standard***: *The keyword const shall be used to modify the arguments wherever appropriate.*

***Standard***: *A function should not have an excessive number of calling parameters.*

If a function has too many calling parameters, it may indicate that the function is too complex or performing too many actions. An alternative is to group the parameters into a struct; this also saves stack space at each function call if the struct is passed by reference.

***Standard***: *A function should do one and only one thing, and it should be kept short.*

***Guideline***: *File scope functions shall be declared as static.*

**3.5 Data Declarations**

***Standard****: Align each declaration type name.*

Avoid extremely long or continuation lines in a declaration statement by using multiple statements.

***Guideline***: *Explicitly dimension all arrays using parameters as much as possible to specify array dimensions/sizes. We recognize that it is not always possible to do so.*

Avoid extremely long or continuation lines in a declaration statement by using multiple statements.

###

### **3.5.1 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).

***Guideline***: *Character constants should not contain more than one character.*

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

###  **3.5.1.1 Example 5: Use of Character Constants**

 Bad example of character constant usage:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

short Crlf = '\r\n';

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Good example of character constant usage via a string constant:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 char Crlf[] = "\r\n";

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

***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.

###

### **3.5.1.2 Example 6: Use of Revised Constants from *Plum***

 Bad example of revised constant usage—this example employs a magic number:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if (Index < 100)

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Good example of revised constant usage—this example defines size then uses it:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 #define SIZE 100

 if (Index < SIZE)

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### **3.5.2 Variables**

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

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

###  **3.5.2.1 Example 7: Correct Use of Initializers**

The first statement defaults to edit the current record and the next initializes the menu mode for users who are not experienced with using the product:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int DefaultMenuSelection = EDIT;

int MenuMode = NOVICE;

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

###  **3.5.2.2 Example 8: Correct Use of Flags and Comments**

The first statement is a flag to determine which processing path to take and the next indicates when pre-processed macros exist:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int ForkFlag; /\* Flag to determine which processing path \*/

 /\* to take (fork) \*/

 int MacroFlag = NO; /\* Indicates when pre-processed macros exist\*/

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***Guideline***: *File scope variables shall be declared static.*

###

### **3.5.3 Data Types**

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

***Guideline***: *Code should not assume sizes (number of bytes) for standard types (like long) since some compilers will change these default sizes when code is compiled as 32 versus 64 bits.*

It is often safer to typedef based on system defined types (e.g. typedef int32\_t SLONG;) which are explicitly sized.

###

### **3.5.4 Pointers**

***Standard****: Pointer conversions shall be explicitly cast.*

***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.

### **3.5.5 Expressions**

###

###  **3.5.5.1 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.*

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

###

### **3.5.5.2 Byte Ordering**

***Guideline***: *Programs that may be run on multiple platforms should not assume byte order (endian type) within integer or floating pointer data.*

###  **3.5.5.2.1 Example 9: Byte Ordering**

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 \*/

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### **3.5.5.3 Byte Size**

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

Programmers should not develop code that assumes that a byte is 8 bits in length. Although this is a standard size, it can vary across hardware.

###  **3.5.5.3.1 Example 10: Byte Size**

An example from *Plum*:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

bit &= ~1; /\* good, ones complement operator \*/

bit &= 0177776; /\* bad turns off high bits \*/

 /\* on 32 bit machine \*/

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### **3.5.5.4 Macros**

***Guideline***: *Use subroutines/functions instead of complex user defined macros.*

System defined macros (e.g. “MIN”, type definition macros) are OK.

***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.

###  **3.5.5.4.1 Example 11: Improper Macro Usage**

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 #define A\_BAD\_MACRO 1 + 1

 Row = Current\_Point \* A\_BAD\_MACRO

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

This then maps into:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 Row = Current\_Point \* 1 + 1

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

However, Row equals Current\_Point + 1 not Current\_Point \* 2 as desired. It should have been coded as:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#define A\_GOOD\_MACRO (1 + 1)

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

###  **3.5.5.4.2 Example 12: Macro Definition**

When invoked as SQUARE(z+1), the results are not what would be expected:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#define SQUARE(x) x\*x

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

However, just surrounding the arguments with parentheses like the following is not enough:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 #define SQUARE(x) (x\*x)

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 #define SQUARE(x) ((x)\*(x))

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**3.6 Statements**

 ***Standard****: Each statement shall begin on a separate line.*

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

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.

### **3.6.1 Example 13: Braces in Statements**

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 and can occur during maintenance when statements are added.

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if (Requested\_Value == TERMINATE)

 printf("Terminated abnormally");

 exit(ABNORMAL);

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***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.*

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

1. Comments placed near the goto statement to document the applica­ble constraints.
2. Comments placed near the statement receiving control to document the origin of the transfer of control.

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

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

***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".

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

***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.

***Guideline***: *Because a switch statement is easier to understand, it should be used instead of* if*or* else *chains whenever possible*.

***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.*

### **3.6.2 Example 14: Nesting Levels**

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 \*/

 ...

 }

}

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***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.

***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 in example 15.

### **3.6.3 Example 15: Styles of Control Structures**

 First style for control structures:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if (Start\_Point == ZERO)

 {

 ++Start\_Point;

 }

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Second style for control structures:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if (Start\_Point == ZERO) {

 ++Start\_Point;

 }

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***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.

### **3.6.4 Example 16: Comparison Written Explicitly**

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

/\* 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);

}

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**3.7 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*.

### **3.7.1 Example 17: Poor Examples of Loop Variables**

 First bad example of loop variables:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

for (Row = 0; Row < Dot; Row++) /\* Bad \*/

{ /\* increments test \*/

 Dot++; /\* condition in loop \*/

}

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

 Second bad example of loop variables:

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

for (Row = 0; Row < Dot; Row++) /\* Bad \*/

{ /\* Explicit change \*/

 Dot = 3; /\* to test condition \*/

}

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**4. Readability and Documentation**

**4.1 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).*

***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.

***Guideline***: *Upper case should be used for user* #define'*d* i*dentifiers and Macro names.*

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

* 1. The first character of each word in upper case lettering with subsequent characters in lower case.
	2. Each word separated by the underscore character.
	3. A combination of the above.

### **4.1.1 Example 18: Mixed Case Underscores**

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#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;

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**4.2 Comments**

***Standard***: *Comments shall not be used to disable executable statements.*

***Standard***: *The characters* /\* *shall introduce a comment, while the characters* \*/ *terminate it*.

***Standard***: *Comments shall not be nested.*

***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.

***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:

1. *Boxed comments*: Each line contains the characters /\* and the characters \*/ which are all vertically aligned with the other lines of the comment.
2. *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.
3. *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.

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

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

**5. Libraries and Memory**

**5.1 Common Libraries**

***Guideline****: Use code which already exists in a common library—there is no need to rewrite it.*

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.

**5.2 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.

**6. Interoperability**

**6.1 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 the guidelines listed below.

***Standard***: *Align each declaration type name*.

Avoid extremely long or continuation lines in a declaration statement by using multiple statements.

***Guideline***: *List several variables of a single type on a line alphabetically.*

Explicitly dimension all arrays using parameters as much as possible to specify array dimensions/sizes. We recognize that it is not always possible to do so.

***Guideline***: *Data types must match between languages.*

Though not all types have a match, here are some examples: (Fortran) real\*4 = (C) float and (Fortran) integer\*4 = (C) int

***Guideline***: *Always use lower case names for routines to avoid problems.*

C is case sensitive, Fortran is not and all routine names get converted to lowercase when compiled.

***Guideline***: *Fortran appends an* “\_” *character to all routine names after compilation*.

* 1. Add the “\_” character to your called routine name in C code (see Example 20 below).
	2. Add the “\_” character to your C routine name when called from Fortran (see Example 21 below).

\*\* NOTE: IBM needs to set compiler flag for “\_” \*\*

***Guideline***: *Avoid passing global data between languages, such as common blocks.*

***Guideline***: *Fortran passes arguments by reference, C passes by value or reference.*

Use the C operators “&” and “\*” accordingly when passing arguments.

***Guideline***: *Use the* “extern” *command in C to declare external Fortran routines.*

***Guideline***: *Use the* “external” *command in Fortran to declare external C routines.*

### **6.1.1 Example 19: 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);

}

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### **6.1.2 Example 20: 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

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**6.2 C/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.

***Standard***: *When calling a C function from C++ the function name will be mangled unless you turn it off using 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.

### **6.2.1 Example 21: C Syntax Wrapping**

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

extern "C" int my\_great\_function();

extern "C"

{

 int strncpy(...);

 int my\_great\_function();

};

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**6.2.2 Example 22: Creating a C Function in C++**

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

extern "C" void

a\_c\_function\_in\_cplusplus(int a)

{

}

\_\_cplusplus Preprocessor Directive

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

### **6.2.3 Example 23: Preprocessor Directive**

If you have code that must compile in a C and C++ environment then you must use the \_\_cplusplus preprocessor directive.

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#ifdef \_\_cplusplus

extern "C" some\_function();

#else

extern some\_function();

#endif

 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**7. Errors**

**7.1 Error Handling**

***Standard***:  *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..

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

***Standard****: Include the system error text for every system error message.*

***Standard***: *Check every call to* malloc *or* realloc *unless you know your versions of these calls do the right thing.*

It may be a good idea to create a wrapper for these calls, so you can do the right thing always and developers don't have to make memory checks everywhere.

**8. Items to Avoid in C**

Refer to the General Programming Standards for a more comprehensive list of security issues. In this document we augment that list with some items specific to C.

**8.1 Library functions to Avoid**

## ***Guideline****: These functions, if not used with care, are prone to buffer overflows.*

## Avoid or be very careful when using the following C library functions:

* gets
* strcpy
* strcat
* sprint
* scanf
* sscanf
* fscanf
* vfscanf
* vsprintf
* vscanf
* vsscanf
* streadd
* strecpy
* strtrns

The following website identifies risks associated with using these functions, provides information on alternative C library functions, and also illustrates safer ways to implement these functions if they are used http://www.ibm.com/developerworks/library/s-buffer-defend.html.

**8.2 Manage Interprocess Communication (IPC) Resources**

IPC resources are shared resources that must be managed properly. These include shared memory, message queues, and semaphores. These are extremely useful and powerful tools. The danger and power of these resources is that they can be created by a program and then persist after that program exits. They live within the system and are not owned by any particular process.

***Recommendation****:* *A program can create shared memory blocks with* shmget, *semaphores with* semget, *and message queues with* msgget.

However, always be sure to free these resources after the programs have finished or develop software that is smart enough to clean up after itself so these allocated resources do not linger. The ipcs and ipcrm system calls can be used to identify and free these resources. This concept can be found at: http://beej.us/guide/bgipc/output/html/singlepage/bgipc.html.

**8.3 Grandfathering**

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

### **8.3.1 COTS**

***Recommendation****: 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.

### **8.3.2 Reuse**

***Recommendation****:*  *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.*

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

***Recommendation****: 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.*

***Recommendation****: 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.*