Software Development Guidelines

Contents

WEBster Home Page

Authors:

Randall Hyde

Note: The authors of this paper acknowledge that many of the rules and ideas appearing in this document were taken from Steve McConnell's text "Code Complete" (Microsoft Press, ISBN 1-55615-484-4). This is an excellent text on personal software engineering and every programmer should obtain a copy. Additional material was taken from Steve Maguire's "Writing Solid Code," also from Microsoft Press (ISBN 1-55615-551-4). Another good book on introductory software engineering/software modeling is "Software Development in Pascal" by Sartaj Sahni (ISBN 0-942450-01-9). This document also borrows heavily from that text.

Software Development Guidelines

1 - Introduction

Contents - **Next section** *(2 - General Programming Guidelines)* >

1 - Introduction

The intent of this document is to create a guide for software source code quality. The guidelines appearing herein apply to anyone who creates, modifies, or reads software source code.

This document is not a description of a complete software process. A particular group will need to develop their own methodologies and procedures for the specification, design, implementation, testing, and deployment of their software systems. This document is simply a set of rules to follow during the implementation phase that will help produce a higher quality result.

This document addresses general and language-specific topics. The general concepts apply on any project regardless of any implementation details. The language specific topics apply to a project in addition to the general guidelines once a given programming language has been chosen for the project.

One restriction often found in corporations involving software engineering is the choice of programming language for a project. Many IS shops are "one-language" houses. A corporation, for example, may hire only COBOL programmers and insist that all programs, regardless of their nature, be written in COBOL. Other companies do not have this policy. This allows engineers the flexibility to choose the appropriate tool for the job; it also demands a certain amount of flexibility insofar as engineers must be capable of working with legacy code written in any of several different languages.

Such flexibility, without a central direction guiding development, can lead to chaos. This is particularly true when using languages that support rapid application development and prototyping like Visual BASIC, Powerbuilder, and Delphi. This document is intended to provide guidance during development so one developer can easily take over a project from another without facing "culture shock" upon reading the new program. Most professionals, who take a non-personal view of their programming tools, should find these guidelines non-obtrusive.

Many programmers take a near-religious view of machines and software tools. Alas, such views are often based on an individual's fear of having to learn something new rather than the actual applicability of a given system. Engineers should maintain an open mind with respect to evolving technologies. Doing something out of habit ("Because that's the way it's always been done," or "because that's the industry standard.") are insufficient reasons for continuing to use an inadequate methodology.

Human nature resists change. This document describes several procedures that represent a departure from the norm for most people. However, once you "unlearn" current habits and replace them with new habits, you will probably agree that this change is for the best. Once you develop new habits and expand your software development ideas, you will question how you ever accomplished anything without these new methodologies.

1.1 - What This Process Will Achieve

You can hope to achieve three goals by requiring a consistent source code style:

- Improve the productivity of existing programmers.
- Allow new programmers to become comfortable with existing source code in less time than would otherwise be necessary.
- Allow existing programmers to move around to different projects easily without having to adjust to the programming style in use by other groups.

Standardizing the "look and feel" of the source code will reduce the time to market, it will reduce the time spent correcting problems in the product line, and it will give engineers the flexibility to jump on and off projects without a large learning curve. This means that you will be able to spend less time on legacy products and jump into the more interesting task of designing and implementing future products.

1.2 - How These Guidelines Will Affect You

This coding standard is not intended to be down to the level of dotting i's and crossing t's. It is flexible enough to allow some breathing room while still achieving a visual standard in the way code is written. Nevertheless, we all have coding habits we've developed over the years that we will need to change. As noted above, human nature resists change. However, with the right attitude, perhaps approaching learning this new style with the same sense of curiosity or excitement you would have when learning a new language, you will find that the process of changing these old habits is straightforward.

Almost everyone will need to make some changes to their coding styles. Many people view this as an assault on their personality. After all, we all like to believe we are unique (especially software development types) and many of us tend to exhibit our personality in our programming style. Individuals often view any attempt to conform their programming style to some "standard" as a step in conforming their personality as well. However, this is really no different than expecting programmers to write their comments in English rather than, say, French or Spanish.

You should encourage engineers to express (the positive aspects of) their personalities in their code in a positive fashion. You should pride yourself` on the diversity of the engineering staff and encourage creativity and experience in software development. However, in a situation where you have a large number of software engineers and this number is growing everyday, certain standards will be necessary in order to ensure effortless communication between engineers via code. A reasonable software development standard will help promote this.

Adopting this new standard will improve your worth to the company as well as improve you professionally. You will quickly begin reaping benefits from this new process.

1.3 - What the Standard does not Cover

There are many issues with regard to programming style that these guidelines do not cover. They do not, for example, dictate a particular programming language. Engineers should have the experience and knowledge to choose an appropriate language for a given project based on technical and economic merits. In particular, this specification does not:

- Prevent you from trying something not explicitly covered by the software development standard.
- Dictate the use of, or forbid the use of, any particular software development tool (certain enterprise-wide tools are excepted, including configuration/version management tools and software defect tracking tools).
- Specify the choice of a particular programming language on a project.
- Specify the complete software design process.
- Deal with platform specific issues.
- Deal with project management issues.
- Deal with user documentation for software products.

Although this document does not address those issues, other document and initiatives may very well do so. You should consult with your manager for the specific liberties and responsibilities that you have.

1.4 - What the Style Guidelines do Cover

Software development occurs in different languages, environments, and on different machines. These style guidelines attempt to generalize programming style across these divergent systems. For example, assume you've written a program in two different languages. If those two languages share some approximately equivalent statements and semantics (e.g., BASIC and C), the two programs should look very similar. Obviously, some language pair choices (e.g., Pascal/assembly, BASIC/LISP, or C++/SETL) will look quite a bit different, but many elements in the pairs should be identical, including variable names, comments, etc.

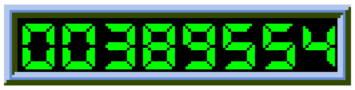
Although this specification attempts to be generic, there are some language-specific issues that any set of generic style guidelines must address. Language-specific issues appear in later sections. This standard attempts to address the following issues:

- Modularization
- Characteristics of high quality program units
- Data typing
- Names
- Abstract data types and objects
- Organizing control structures
- Program layout
- Comments and (program) documentation
- Coding for testability

This document will not address issues like how to design software, how to test software, how to debug software, etc. Different documents, particular to each department, will address those topics.

Contents - **Next section** *(2 - General Programming Guidelines) >*

Number of Web Site Hits since Dec 1, 1999:



Software Development Guidelines

2 - General Programming Guidelines

< Previous section *(1 - Introduction)* - **Contents** - **Next section** *(3 - C (and related C++) Specific Issues)* **>**

---

2 - General Programming Guidelines

This document contains three types of rules:

- Guidelines,
- Rules that were made to be broken, and
- Enforced rules that an engineer must never violate.

The description of the second rule above is, obviously, tongue-in-cheek. Such rules should always be followed unless there is a valid, defendable, reason for violating the rule. Violations of these rules should be rare and well documented (explaining the reason behind the violation). Guidelines are a less severe form of a rule that was made to be broken. As a general rule, you should always follow guidelines unless there are reasons for violating them. Guideline violations do not need to be documented, only be verbally defensible. The third category, enforced rules, should only be violated if everyone agrees to ammend this document to demote the enforced rule to a simple rule. This document will refer to these three types of rules as guidelines, rules, and enforced rules.

This standard deals with products developed using several different programming languages. The desire is to have a "look and feel" to the code that is (as much as possible) consistent across all programs, not simply across programs in a given language. That is, Visual BASIC, Pascal/Delphi, C/C++, Tcl, assembly language, shell scripts, Perl, Flex/Lex, Yacc/Bison, and other programs should all adhere (as much as possible) to the same standard. Obviously, differences in the languages will have a big impact on the applicability of these style guidelines. Nevertheless, many concepts are common to all these languages (e.g., the need for readable identifiers, meaningful comments, appropriate layout, etc.).

C/C++ programmers will feel the biggest impact of these guidelines. C/C++ programmers have developed a considerable set of completely arcane and poorly thought-out conventions over the years. Rarely will you see these conventions employed in programs written in other languages (e.g., can you truly explain why capitalizing all the characters in an identifier for constants and macros is the best way to point out that those objects are macros or constants? This convention seems to be unique to C/C++ and other C-derived languages [e.g. Java]).

The conventions appearing in this paper have been carefully researched and thought out. The principle author (Randy Hyde) has studied and taught programming language design for several years at UC Riverside. While this paper is malleable and subject to change, be aware that most of the concepts appearing in this style guide are quite defensible with respect to modern programming language design. They are not the result of a desire to make every language in the world look like the first language the authors learned (which, by the way, was FORTRAN and has very little influence on these guidelines).

The following subsections cover the following generic topics: characteristics of high quality program units, modularization data typing, names, organizing control structures, program layout, comments and (program) documentation, coding for testability, and communication. The following major section describes these subjects with respect to specific programming languages.

For more information on these subjects, you should check out "Code Complete" by Steve McConnell and "Writing Solid Code" by Steve Maguire, both from Microsoft Press. Although these texts are both full of contradictions and contain lots of bad advice, they are easily read and contain several diamonds amongst the gravel.

2.1 - Characteristics of High Quality Routines

A routine is a generic program unit, that is, a function, procedure, subroutine, iterator, block, or main program. The quality of the routines appearing in a program have a tremendous impact on the reliability and readability of that program. The following subsections describe some of the attributes of a high quality routine.

2.1.1 - Routine Cohesion

Routines exhibit the following kinds of cohesion (listed from good to bad):

- Functional or logical cohesion exists if the routine accomplishes exactly one (simple) task.
- Sequential or pipelined cohesion exists when a routine does several sequential operations that must be performed in a certain order with the data from one operation being fed to the next in a "filter-like" fashion.
- Global or communicational cohesion exists when a routine performs a set of operations that make use of a common set of data, but are otherwise unrelated.
- Temporal cohesion exists when a routine performs a set of operations that need to be done at the same time (though not necessarily in the same order). A typical initialization routine is an example of such code.
- Procedural cohesion exists when a routine performs a sequence of operations in a specific order, but the only thing that binds them together is the order in which they must be done. Unlike sequential cohesion, the operations do not share data.
- State cohesion occurs when several different (unrelated) operations appear in the same module and a state variable (e.g., a parameter) selects the operation to execute. Typically such routines contain a case (switch) or **if..elseif..elseif...** statement.
- No cohesion exists if the operations in a routine have no apparent relationship with one another.

The first three forms of cohesion above are generally acceptable in a program. The fourth (temporal) is probably okay, but you should rarely use it. The last three forms should almost never appear in a program. For some reasonable examples of routine cohesion, you should consult "Code Complete".

Guideline:

> **All routines should exhibit good cohesiveness. Functional cohesiveness is best, followed by sequential and global cohesiveness. Temporal cohesiveness is okay on occasion. You should avoid the other forms.**

2.1.2 - Routine Coupling

Coupling refers to the way that two routines communicate with one another. There are several criteria that define the level of coupling between two routines:

- Cardinality- the number of objects communicated between two routines. The fewer objects the better (i.e., fewer parameters).
- Intimacy- how "private" is the communication? Parameter lists are the most private form; private data fields in a class or object are next level; public data fields in a class or object are next, global variables

are even less intimate, and passing data in a file or database is the least intimate connection. Well-written routines exhibit a high degree of intimacy.

- Visibility- this is somewhat related to intimacy above. This refers to how visible the data is to the entire system that you pass between two routines. For example, passing data in a parameter list is direct and very visible (you always see the data the caller is passing in the call to the routine); passing data in global variables makes the transfer less visible (you could have set up the global variable long before the call to the routine). Another example is passing simple (scalar) variables rather than loading up a bunch of values into a structure/record and passing that structure/record to the callee.
- Flexibility- This refers to how easy it is to make the connection between two routines that may not have been originally intended to call one another. For example, suppose you pass a structure containing three fields into a function. If you want to call that function but you only have three data objects, not the structure, you would have to create a dummy structure, copy the three values into the field of that structure, and then call the routine. On the other hand, had you simply passed the three values as separate parameters, you could still pass in structures (by specifying each field) as well as call the routine with separate values.

A function is loosely coupled if it exhibits low cardinality, high intimacy, high visibility, and high flexibility. Often, these features are in conflict with one another (e.g., increasing the flexibility by breaking out the fields from a structures [a good thing] will also increase the cardinality [a bad thing]). It is the traditional goal of any engineer to choose the appropriate compromises for each individual circumstance; therefore, you will need to carefully balance each of the four attributes above.

A program that uses loose coupling generally contains fewer errors per KLOC (thousands of lines of code). Furthermore, routines that exhibit loose coupling are easier to reuse (both in the current and future projects). For more information on coupling, see the appropriate chapter in "Code Complete".

Guideline:

Coupling between routines in source code should be loose;

2.1.3 - Routine Size

Sometime in the 1960's, someone decided that programmers could only look at one page in a listing at a time, therefore routines should be a maximum of one page long (66 lines, at the time). In the 1970's, when interactive computing became popular, this was adjusted to 24 lines -- the size of a terminal screen. In fact, there is very little empirical evidence to suggest that small routine size is a good attribute. In fact, several studies on code containing artificial constraints on routine size indicate just the opposite -- shorter routines often contain more bugs per KLOC.

A routine that exhibits functional cohesiveness is the right size, almost regardless of the number of lines of code it contains. You shouldn't artificially break up a routine into two or more subroutines (e.g., sub_partI and sub_partII) just because you feel a routine is getting to be too long. First, verify that your routine exhibits strong cohesion and loose coupling. If this is the case, the routine is not too long. Do keep in mind, however, that a long routine is probably a good indication that it is performing several actions and, therefore, does not exhibit strong cohesion.

Of course, you can take this too far. Most studies on the subject indicate that routines in excess of 150-200 lines of code tend to contain more bugs and are more costly to fix than shorter routines. Note, by the way, that you do not count blank lines or lines containing only comments when counting the lines of code in a program.

Guideline:

> **Do not let artificial constraints affect the size of your routines. If a routine exceeds 150-200 lines of code, make sure the routine exhibits functional or sequential cohesion. Also look to see if there aren't some generic subsequences in your code that you can turn into stand alone routines.**

Rule:

> **Never shorten a routine by dividing it into n parts that you would always call in the appropriate sequence as a way of shortening the original routine.**

2.2 - Modularization

A module is a collection of objects that are logically related. Those objects may include constants, data types, variables, and program units (e.g., functions, procedures, etc.). Note that objects in a module need not be physically related. For example, it is quite possible to construct a module using several different source files. Likewise, it is quite possible to have several different modules in the same source file. However, the best modules are physically related as well as logically related; that is, all the objects associated with a module exist in a single source file (or directory if the source file would be too large) and nothing else is present.

Modules contain several different objects including constants, types, variables, and program units (routines). Modules shares many of the attributes with routines; this is not surprising since routines are the major component of a typical module. However, modules have some additional attributes of their own. The following sections describe the attributes of a well-written module.

2.2.1 - Module Attributes

A module is a generic term that describes a set of program related objects (routines as well as data and type objects) that are somehow coupled. Good modules share many of the same attributes as good routines as well as the ability to hide certain details from code outside the module.

Good modules exhibit strong cohesion. That is, a module should offer a (small) group of services that are logically related. For example, a "printer" module might provide all the services one would expect from a printer. The individual routines within the module would provide the individual services.

Good modules exhibit loose coupling. That is, there are only a few, well-defined (visible) interfaces between the module and the outside world. Most data is private, accessible only through accessor functions (see information hiding below). Furthermore, the interface should be flexible.

Good modules exhibit information hiding. Code outside the module should only have access to the module through a small set of public routines. All data should be private to that module.

Guideline:

> **A module should implement an abstract data type. All interface to the module should be through a well-defined set of operations.**

2.2.2 - Physical Organization of Modules

Many languages provide direct support for modules (e.g., packages in Ada, modules in Modula-2, and units in Delphi/Pascal). Some languages provide only indirect support for modules (e.g., a source file in C/C++). Others, like BASIC, don't really support modules, so you would have to simulate them by physically grouping objects together and exercising some discipline.

Insofar as the particular language you're using supports the concept of a module, embrace that implementation. Beyond that, here are a few rules that can help make modules easier to read and understand.

Rule:

> **Each module should completely reside in a single source file. If size considerations prevent this, then all the source files for a given module should reside in a subdirectory specifically designated for that module.**

Rule:

> **If a particular language processing system does not support modules of any kind, simulate those modules by physically grouping the objects in the source code. Be sure to access the module using only "approved" interfaces. Always check for inconsistencies when reviewing your code.**

Some people have the crazy idea that modularization means putting each function in a separate source file. Such physical modularization generally impairs the readability of a program more than it helps. Strive instead for logical modularization, that is, defining a module by its actions rather than by source code syntax (e.g., separating out functions).

This document does not address the decomposition of a problem into its modular components. Presumably, you can already handle that part of the task. There are a wide variety of texts on this subject if you feel week in this area.

2.3 - Data Typing, Declarations, Variables, and other Objects

Most languages' built-in data types are abstractions of the underlying machine organization and rarely does the language define the types in terms of exact machine representations. For example, an integer variable may be a 16-bit two's complement value on one machine, a 32-bit value on another, or even a 64-bit value. Clearly, a program written to expect 32 or 64 bit integers will malfunction on a machine (or compiler) that only supports 16-bit integers. The reverse can also be true.

One supposed advantage of a high level language is that it abstracts away the machine dependencies that exist in data types. In theory, an integer is an integer is an integer ... In practice, there are short integers, integers, and long integers. Common sizes include eight, sixteen, thirty-two, and even sixty-four bits, with more on the way. Unfortunately, the abstraction the high level language provides can destroy the ability to port a program from one machine to another.

Most modern high level language provide programmers with the ability to define new data types as isomorphisms (synonyms) of existing types. Using this facility, it is possible to define a data type module that provides precise definitions for most data types. For example, you could define the int16 and int32 data types that always use 16 or 32 bits, respectively. By doing so, you can easily guarantee that your programs can easily port between most systems (and their compilers) by simply changing the definition of the int16 and int32 types on the new machine. Consider the following C/C++ example:

On a 16-bit machine:

```
typedef int int16;
typedef long int32;
```

On a 32-bit machine:

```
typedef short int16;
typedef int int32;
```

Rule:

> **If a built-in type has different semantics on different architectures or in different compilers, always use a set of type definitions that let you easily change adjust the program to a different architecture. It is dangerous to assume a particular object uses a specific data format (e.g., two's complement binary or IEEE floating point). It is even worse to assume an object has a fixed number of bits. You should avoid using predefined types in a language.**

Guideline:

> **If the data type you are creating depends upon a specific format, use names like int8, int16, int32, int64, real32, real64, and real80 (that is, a type name with the number of bits appended) to denote your types. If the data type does not depend on a specific representation, use a descriptive name (see the next section on naming conventions). Try to avoid the use of types in a language that vary depdning on the underlying machine representation (alas, this is not always possible).**

Don't redefine existing types. This may seem like a contradiction to the guideline above, but it really isn't. This statement says that if you have an existing type that uses the name "integer" you should not create a new type named "integer." Doing so would only create confusion. Another programmer, reading your code, may confuse the old "integer" type every time s/he sees a variable of type integer. This applies to existing user types as well as predefined types.

Enforced Rule:

> **Never redefine an existing type.**

Declare all variables, even if the language processor allows implicit declarations. At one time there was a controversy as to whether it was better to have implicitly declared variables or force the user to explicitly declare all variables (e.g., the FORTRAN vs. ALGOL/Pascal crowd). When NASA and JPL lost a Venus probe due to an implicitly declared variable (that just happened to have the wrong type), the "explicitly declare" crowd won the argument. Fortunately, most modern languages require explicit declarations.

Enforced Rule:

> **Always explicitly declare all variables (and other identifiers) unless the language does not allow this.**

Some languages force you to declare all your variables at a given point in a program unit (e.g., Pascal); some languages are more flexible and let you declare variables anywhere in your program as long as you declare them before their first use; other languages do not require that you declare variables at all (see the above rule).

Since it is possible to declare symbols at different points in a program, different programmers have developed different conventions concern the position of their declarations. The two most popular conventions are the following:

- Declare all symbols at the beginning of the associated program unit (function, procedure, etc.).
- Declare all variables as close as possible to their use.

Logically, the second scheme above would seem to be the best. However, it has one major drawback - although names typically have only a single definition, the program may use them in several different locations. So although you can easily define a variable just prior to its first use, other uses may be hundreds of lines away. The advantage of declaring variables at the beginning of the program unit is that, no matter how far away it is, the programmer always knows where to look to find the variable declarations. If you embed the definition in the middle of the code nearest the first usage, someone reading the program may have to resort to a "linear search" in order to find the declaration.

Rule:

> **All variable, constant, and type definitions should occur at the very beginning of the program unit whose limits define the scope of the object.**

Unfortunately, not all name definitions are passive, some actually execute code. A instance of a class object in C++ is a good example. The definition of a class object calls the constructor for that class. The constructor may require the computation of some parameter values prior to the object's definition. This would prevent the placement of the definition at the beginning of the module. The solution is rather simple and well within the definition of a "Rule" within this guide:

Rule:

> **If you cannot define an object at the beginning of the program unit to which it belongs, then put a place-holder comment at the beginning of the block and define the variable as soon as possible within the program unit. You should place a comment near such a definition to remind the reader to update the comment at the beginning of the block if the actual definition ever changes.**

Some might argue that certain languages, like C++, provide excellent facilities for declaring otherwise anonymous variables with certain language constructs. For example, the "for (int i = 0; i < 10; ++i) ..." statement limits the scope of "i" to this for loop. However, the goal of these guidelines is to produce a standard that applies to all languages; making special exceptions for C++ (or some feature-laden language) will only lead to confusion. Besides, C++ lets you create new program units by using "{" and "}" (e.g., the compound statement). Those who absolutely desire to put their definitions as close to the for-loop as possible can always do something like the following:

```
        // Previous statements in this code...
                .
                .
                .
        {
                int i;
                for (i=start; i <= end; ++k) ...
        }
                .
                .
                .
        // Additional statements in this code.
```

Descriptive comments should always accompany a set of variable declarations. These comments should describe the purpose of the variables, provide complete English names for the variables if the names use any abbreviations (see the next section), and describe any constraints or assumptions on the use of these variables. The position of these comments should be immediately before the block or program unit that declares the variables (e.g., in the block of comments preceding a function definition). To improve readability and make it easy for a programmer to locate a particular name while manually scanning through a listing, you should place only one variable declaration per line so the reader can easily find the variable's name while scanning the left-hand side of the list. In languages where the type name precedes the variable name, it's a good idea to put the type name on one line and the variable name (indented) on the next line.

Rule:

> **Associated with any set of variable declarations will be a set of comments known as the "Data Dictionary." This data dictionary will describe the name and purpose for each variable. The Data Dictionary will also describe any constraints or assumptions on the use of the variables.**

Guideline:

> **Variable declarations should appear on separate lines. If desired, the type specification should appear on a separate line as well. Variable and type names should be aligned in columns and easy to find and read.**

Examples:

```
        (* Pascal *)
        var
                LineCnt,                { Number of lines, words, and   }
                WordCnt,                { and characters in a file.     }
                CharCnt:integer;

        (* Also Reasonable *)

        var
                LineCnt:integer;        { Number of lines, words, and   }
                WordCnt:integer;        { and characters in a file.     }
                CharCnt:integer;


        /* C/C++  */

        int
                LineCnt,                /* Number of lines, words, and  */
                WordCnt,                /* and characters in a file.    */
                CharCnt;

        /* Another C/C++ Version */

        int     LineCnt;        /* Number of lines, words, and          */
        int     WordCnt;        /* and characters in a file.            */
        float   CharCnt;
```

2.4 - Names

According to studies done at IBM, the use of high-quality identifiers in a program contributes more to the readability of that program than any other single factor, including high-quality comments. The quality of your identifiers can make or break your program; program with high-quality identifiers can be very easy to read, programs with poor quality identifiers will be very difficult to read. There are very few "tricks" to developing high-quality names; most of the rules are nothing more than plain old-fashion common sense. Unfortunately, programmers (especially C/C++ programmers) have developed many arcane naming conventions that ignore common sense. The biggest obstacle most programmers have to learning how to create good names is an unwillingness to abandon existing conventions. Yet their only defense when quizzed on why they adhere to (existing) bad conventions seems to be "because that's the way I've always done it and that's the way everybody else does it."

Naming conventions represent one area in Computer Science where there are far too many divergent views (program layout is the other principle area). The primary purpose of an object's name in a programming language is to describe the use and/or contents of that object. A secondary consideration may be to describe the type of the object. Programmers use different mechanisms to handle these objectives. Unfortunately, there are far too many "conventions" in place, it would be asking too much to expect any one programmer to follow several different standards. Therefore, this standard will apply across all languages as much as possible.

The vast majority of programmers know only one language - English. Some programmers know English as a second language and may not be familiar with a common non-English phrase that is not in their own language (e.g., rendezvous). Since English is the common language of most programmers, all identifiers should use easily recognizable English words and phrases.

Rule:

All identifiers that represent words or phrases must be English words or phrases.

2.4.1 - Alphabetic Case Considerations

A case-neutral identifier will work properly whether you compile it with a compiler that has case sensitive identifiers or case insensitive identifiers. In practice, this means that all uses of the identifiers must be spelled exactly the same way (including case) and that no other identifier exists whose only difference is the case of the letters in the identifier. For example, if you declare an identifier "ProfitsThisYear" in Pascal (a case-insensitive language), you could legally refer to this variable as "profitsThisYear" and "PROFITSTHISYEAR". However, this is not a case-neutral usage since a case sensitive language would treat these three identifiers as different names. Conversely, in case-sensitive languages like C/C++, it is possible to create two different identifiers with names like "PROFITS" and "profits" in the program. This is not case-neutral since attempting to use these two identifiers in a case insensitive language (like Pascal) would produce an error since the case-insensitive language would think they were the same name.

Enforced Rule:

All identifiers must be "case-neutral."

Different programmers (especially in different languages) use alphabetic case to denote different objects. For example, a common C/C++ coding convention is to use all upper case to denote a constant, macro, or type definition and to use all lower case to denote variable names or reserved words. Prolog programmers use an initial lower case alphabetic to denote a variable. Other comparable coding conventions exist. Unfortunately, there are so many different conventions that make use of alphabetic case, they are nearly worthless, hence the following rule:

Rule:

> **You should never use alphabetic case to denote the type, classification, or any other program-related attribute of an identifier (unless the language's syntax specifically requires this).**

There are going to be some obvious exceptions to the above rule, this document will cover those exceptions a little later. Alphabetic case does have one very useful purpose in identifiers - it is useful for separating words in a multi-word identifier; more on that subject in a moment.

To produce readable identifiers often requires a multi-word phrase. Natural languages typically use spaces to separate words; we can not, however, use this technique in identifiers. Unfortunatelywritingmultiwordidentifiers makesthemalmostimpossibletoreadifyoudonotdosomethingtodistiguishtheindividualwords (Unfortunately writing multiword identifiers makes them almost impossible to read if you do not do something to distinguish the individual words). There are a couple of good conventions in place to solve this problem. This standard's convention is to capitalize the first alphabetic character of each word in the middle of an identifier.

Rule:

> **Capitalize the first letter of interior words in all multi-word identifiers.**

Note that the rule above does not specify whether the first letter of an identifier is upper or lower case. Subject to the other rules governing case, you can elect to use upper or lower case for the first symbol, although you should be consistent throughout your program.

Lower case characters are easier to read than upper case. Identifiers written completely in upper case take almost twice as long to recognize and, therefore, impair the readability of a program. Yes, all upper case does make an identifier stand out. Such emphasis is rarely necessary in real programs. Yes, common C/C++ coding conventions dictate the use of all upper case identifiers. Forget them. They not only make your programs harder to read, they also violate the first rule above.

Rule:

> **Avoid using all upper case characters in an identifier.**

2.4.2 - Abbreviations

The primary purpose of an identifier is to describe the use of, or value associated with, that identifier. The best way to create an identifier for an object is to describe that object in English and then create a variable name from that description. Variable names should be meaningful, concise, and non-ambiguous to an average programmer fluent in the English language. Avoid short names. Some research has shown that programs using identifiers whose average length is 10-20 characters are generally easier to debug than programs with substantially shorter or longer identifiers.

Avoid abbreviations as much as possible. What may seem like a perfectly reasonable abbreviation to you may totally confound someone else. Consider the following variable names that have actually appeared in commercial software:

NoEmployees, NoAccounts, pend

The "NoEmployees" and "NoAccounts" variables seem to be boolean variables indicating the presence or absence of employees and accounts. In fact, this particular programmer was using the (perfectly reasonable in the real world) abbreviation of "number" to indicate the number of employees and the number of accounts. The "pend" name referred to a procedure's end rather than any pending operation.

Programmers often use abbreviations in two situations: they're poor typists and they want to reduce the typing effort, or a good descriptive name for an object is simply too long. The former case is an unacceptable reason for using abbreviations. The second case, especially if care is taken, may warrant the occasional use of an abbreviation.

Guideline:

> **Avoid all identifier abbreviations in your programs. When necessary, use standardized abbreviations or ask someone to review your abbreviations. Whenever you use abbreviations in your programs, create a "data dictionary" in the comments near the names' definition that provides a full name and description for your abbreviation.**

The variable names you create should be pronounceable. "NumFiles" is a much better identifier than "NmFls". The first can be spoken, the second you must generally spell out. Avoid homonyms and long names that are identical except for a few syllables. If you choose good names for your identifiers, you should be able to read a program listing over the telephone to a peer without overly confusing that person.

Rule:

> **All identifiers should be pronounceable (in English) without having to spell out more than one letter.**

2.4.3 - The Position of Components Within an Identifier

When scanning through a listing, most programmers only read the first few characters of an identifier. It is important, therefore, to place the most important information (that defines and makes this identifier unique) in the first few characters of the identifier. So, you should avoid creating several identifiers that all begin with the same phrase or sequence of characters since this will force the programmer to mentally process additional characters in the identifier while reading the listing. Since this slows the reader down, it makes the program harder to read.

Guideline:

> **Try to make most identifiers unique in the first few character positions of the identifier. This makes the program easier to read.**

Corollary:

> **Never use a numeric suffix to differentiate two names.**

Many C/C++ Programmers, especially Microsoft Windows programmers, have adopted a formal naming convention known as "Hungarian Notation." To quote Steve McConnell from Code Complete: "The term 'Hungarian' refers both to the fact that names that follow the convention look like words in a foreign language and to the fact that the creator of the convention, Charles Simonyi, is originally from Hungary." One of the first rules given concerning identifiers stated that all identifiers are to be English names. Do we really want to create "artificially foreign" identifiers? Hungarian notation actually violates another rule as well: names using the Hungarian notation generally have very common prefixes, thus making them harder to read.

Hungarian notation does have a few minor advantages, but the disadvantages far outweigh the advantages.

The following list from Code Complete and other sources describes what's wrong with Hungarian notation:

- Hungarian notation generally defines objects in terms of basic machine types rather than in terms of abstract data types.
- Hungarian notation combines meaning with representation. One of the primary purposes of high level language is to abstract representation away. For example, if you declare a variable to be of type integer, you shouldn't have to change the variable's name just because you changed its type to real.
- Hungarian notation encourages lazy, uninformative variable names. Indeed, it is common to find variable names in Windows programs that contain only type prefix characters, without an descriptive name attached.
- Hungarian notation prefixes the descriptive name with some type information, thus making it harder for the programming to find the descriptive portion of the name.

Guideline:

> **Avoid using Hungarian notation and any other formal naming convention that attaches low-level type information to the identifier.**

Although attaching machine type information to an identifier is generally a bad idea, a well thought-out name can successfully associate some high-level type information with the identifier, especially if the name implies the type or the type information appears as a suffix. For example, names like "PencilCount" and "BytesAvailable" suggest integer values. Likewise, names like "IsReady" and "Busy" indicate boolean values. "KeyCode" and "MiddleInitial" suggest character variables. A name like "StopWatchTime" probably indicates a real value. Likewise, "CustomerName" is probably a string variable. Unfortunately, it isn't always possible to choose a great name that describes both the content and type of an object; this is particularly true when the object is an instance (or definition of) some abstract data type. In such instances, some additional text can improve the identifier. Hungarian notation is a raw attempt at this that, unfortunately, fails for a variety of reasons.

A better solution is to use a suffix phrase to denote the type or class of an identifier. A common UNIX/C convention, for example, is to apply a "_t" suffix to denote a type name (e.g., size_t, key_t, etc.). This convention succeeds over Hungarian notation for several reasons including (1) the "type phrase" is a suffix and doesn't interfere with reading the name, (2) this particular convention specifies the class of the object (const, var, type, function, etc.) rather than a low level type, and (3) It certainly makes sense to change the identifier if it's classification changes.

Guideline:

> **If you want to differentiate identifiers that are constants, type definitions, and variable names, use the suffixes "_c", "_t", and "_v", respectively.**

Rule:

> **The classification suffix should not be the only component that differentiates two identifiers.**

Can we apply this suffix idea to variables and avoid the pitfalls? Sometimes. Consider a high level data type "button" corresponding to a button on a Visual BASIC or Delphi form. A variable name like "CancelButton" makes perfect sense. Likewise, labels appearing on a form could use names like "ETWWLabel" and "EditPageLabel". Note that these suffixes still suffer from the fact that a change in type will require that you change the variable's name. However, changes in high level types are far less common than changes in low-level types, so this shouldn't present a big problem.

2.4.4 - Names to Avoid

Avoid using symbols in an identifier that are easily mistaken for other symbols. This includes the sets {"1" (one), "I" (upper case "I"), and "l" (lower case "L")}, {"0" (zero) and "O" (upper case "O")}, {"2" (two) and "Z" (upper case "Z")}, {"5" (five) and "S" (upper case "S")}, and ("6" (six) and "G" (upper case "G")}.

Guideline:

> **Avoid using symbols in identifiers that are easily mistaken for other symbols (see the list above).**

Avoid misleading abbreviations and names. For example, FALSE shouldn't be an identifier that stands for "Failed As a Legitimate Software Engineer." Likewise, you shouldn't compute the amount of free memory available to a program and stuff it into the variable "Profits".

Rule:

> **Avoid misleading abbreviations and names.**

You should avoid names with similar meanings. For example, if you have two variables "InputLine" and "InputLn" that you use for two separate purposes, you will undoubtedly confuse the two when writing or reading the code. If you can swap the names of the two objects and the program still makes sense, you should rename those identifiers. Note that the names do not have to be similar, only their meanings. "InputLine" and "LineBuffer" are obviously different but you can still easily confuse them in a program.

Rule:

> **Do not use names with similar meanings for different objects in your programs.**

In a similar vein, you should avoid using two or more variables that have different meanings but similar names. For example, if you are writing a teacher's grading program you probably wouldn't want to use the name "NumStudents" to indicate the number of students in the class along with the variable "StudentNum" to hold an individual student's ID number. "NumStudents" and "StudentNum" are too similar.

Rule:

> **Do not use similar names that have different meanings.**

Avoid names that sound similar when read aloud, especially out of context. This would include names like "hard" and "heart", "Knew" and "new", etc. Remember the discussion in the section above on abbreviations, you should be able to discuss your problem listing over the telephone with a peer. Names that sound alike make such discussions difficult.

Guideline:

> **Avoid homonyms in identifiers.**

Avoid misspelled words in names and avoid names that are commonly misspelled. Most programmers are notoriously bad spellers (look at some of the comments in our own code!). Spelling words correctly is hard enough, remembering how to spell an identifier incorrectly is even more difficult. Likewise, if a word is often spelled incorrectly, requiring a programer to spell it correctly on each use is probably asking too much.

Guideline:

> **Avoid misspelled words and names that are often misspelled in identifiers.**

If you redefine the name of some library routine in your code, another program will surely confuse your name with the library's version. This is especially true when dealing with standard library routines and APIs.

Enforced Rule:

> **Do not reuse existing standard library routine names in your program unless you are specifically replacing that routine with one that has similar semantics (i.e., don't reuse the name for a different purpose).**

2.5 - Organizing Control Structures

Although the control structures found in most modern languages trace their roots back to Algol-60, there is a surprising number of subtle variations between the control structures found in common programming languages in use today. This paper will describe a mechanism to unify the control structures the various programming languages use in an attempt to make it possible for a Visual BASIC programmer to easily understand code written in Pascal or C++ as well as make it possible for C++ programmers to read BASIC and Pascal programs, etc.

Typical programming languages contain eight flow-of-control statements: two conditional selection statements (if..then..else and case/switch), four loops (while, repeat..until/do..while, for, and loop), a program unit invocation (i.e., procedure call), and a sequence. There are other less common control structures include processes/coroutines, foreach loops (iterators), and generators, but this paper will focus only on the more common control mechanisms.

Control structures typically come in two forms: those that act on a single statement as an operand and those that act on a sequence of statements. For example, the if..then statement in Pascal operates on a single statement:

if (expression) then Single_Statement;

Of course it is possible to apply Pascal's if statement to a list of statements, but that involves creating a compound statement using a begin..end pair. There are two problems with this type of statement. First of all, it introduces the problem of where you are supposed to put the begin and end in a well-formatted program. This is a very controversial issue with large numbers of programmers in different camps. Some feel an if with a compound statement should look like this:

```
        if (expression) then begin

                { Statement 1 }
                { Statement 2 }
                        .
                        .
                        .
                { Statement n }

        end;
```

Others feel it should look like this:

```
        if (expression) then
        begin

                { Statement 1 }
                { Statement 2 }
                        .
                        .
                        .
                { Statement n }

        end;
```

C/C++ programmers are even worse, there are no less than four common ways of putting the opening and closing braces around a compound statement after an "if".

The second problem with C/C++'s and Pascal's "if" statements is the ambiguity involved. Consider the following Pascal code:

```
        if (expression) then
            if (expression) then
                (* Statement *)

           else (* Statement *);
```

To which "if" does the "else" belong? Of course, you've always been taught that the else goes with the first un-elsed "if" looking back in the file (i.e., the second "if" statement above). What happens if you want it to go with the first one? What happens if there is a long compound statement after the second "if" above and the else is far removed from these two ifs? How easy is it to tell which if belongs to the else?

Modern programming languages (Modula-2, Ada, Visual BASIC, FORTRAN 90, etc.) avoid this problem altogether by using control structures that begin and end with a reserved word, for example, IF and ENDIF. The code above, in one of these languages would look something like:

```
        if (expression) then

                if (expression) then
                        { Statement list}
                endif;

        else
                { Statement list};
        endif;
```

Now there is no question that the else belongs to the first if above, not the second. Note that this form of the if statement allows you to attach a list of statements (between the if and else or if and endif) rather than a single or compound statement. Furthermore, it totally eliminates the religious argument concerning where to put the braces or the begin..end pair on the if.

The complete set of modern programming language constructs includes:

```
if..then..elseif..else..endif
select..case..default..endselect  (typical case/switch statement).
while..endwhile
repeat..until
loop..endloop
```

```
for..endfor
break
breakif
continue
```

Those who have had the opportunity to use these control structures for a considerable amount of time generally recognize their superiority over the Pascal/C/C++ variants. The biggest fault Pascal/C/C++ programmers tend to find with these structures (other than they are different) is that "Ada uses these structures and Ada is a 'yucky' language." Hardly a scientific assessment of the quality of these control constructs.

All programs should use these control structures where available and simulate them if they are not available. The exact simulation details will appear in language-specific sections of this document.

Rule: Programs written in a standard imperative language (e.g., C/C++, Pascal, Ada, Visual BASIC, Delphi, etc.) will use the modern versions of the standard control constructs. If the language does not directly support these control structures, the programmer will simulate them using rules appearing elsewhere in this document.

Rule:

> **If your code contains a chain of if..elseif..elseif.......elseif..... statements, do not use the final else clause to handle a remaining case. Only use the final else to catch an error condition. If you need to test for some value in an if..elseif..elseif.... chain, always test the value in an if or elseif statement.**

Most compilers implement multi-way selection statements (case/switch) using a jump table. This means that the order of the cases within the selection statement is usually irrelevant. Placing the statements in a particular order rarely improves performance. Since the order is usually irrelevant to the compiler, you should organize the cases so that they are easy to read. There are two common organizations that make sense: sorted (numerically or alphabetically) or by frequency (the most common cases first). Either organization is readable, sorting by frequency has the advantage of being faster if your compiler uses a brain-dead if..then.elseif..elseif... implementation of multi-way selection. One drawback to the second approach is that it is often difficult to predict which cases the program will execute most often.

Guideline:

> **When using multi-way selection statements (case/switch) sort the cases numerically (alphabetically) or by frequency of expected occurrence.**

There are three general categories of looping constructs available in common high-level languages- loops that test for termination at the beginning of the loop (e.g., while), loops that test for loop termination at the bottom of the loop (e.g., **repeat..until**), and those that test for loop termination in the middle of the loop (e.g., **loop..endloop**). It is possible simulate any one of these loops using any of the others. This is particularly trivial with the **loop..endloop** construct:

```
/* Test for loop termination at beginning of LOOP..ENDLOOP */

    loop
        breakif (x==y);
         .
         .
         .
    endloop;
```

```
/* Test for loop termination in the middle of LOOP..ENDLOOP */

    loop
         .
         .
         .
        breakif (x==y);
         .
         .
         .
    endloop;

/* Test for loop termination at the end of LOOP..ENDLOOP */

    loop
         .
         .
         .
        breakif (x==y);
    endloop;
```

Given the flexibility of the **loop..endloop** control structure, you might question why one would even burden a compiler with the other loop statements. However, using the appropriate looping structure makes a program far more readable, therefore, you should never use one type of loop when the situation demands another. If someone reading your code sees a **loop..endloop** construct, they may think it's okay to insert statements before or after the exit statement in the loop. If your algorithm truly depends on **while..do** or **repeat..until** semantics, the program may now malfunction.

Rule:

> **Always use the most appropriate type of loop (categorized by termination test position). Never force one type of loop to behave like another.**

Many languages provide a special case of the while loop that executes some number of times specified upon first encountering the loop (a definite loop rather than an indefinite loop). This is the "for" loop in most languages. Unfortunately, this iterative loop ranges from very simple (e.g., in Pascal) to extremely complex (e.g., Algol-68 and PL/I). The vast majority of the time a for loop sequences through a fixed range of value incrementing or decrementing the loop control variable by one. Therefore, most programmers automatically assume this is the way a for loop will operate until they take a closer look at the code. Since most programmers immediately expect this behavior, it makes sense to limit for loops to these semantics. If some other looping mechanism is desirable, you should use a while loop to implement it (since the for loop is just a special case of the while loop). There are other reasons behind this decision as well. Most compilers generate especially efficient code for standard for loops, while they tend to generate less than optimal code for "funny" versions of for loops. Hence there are efficiency considerations as well as readability reasons behind this choice.

Rule:

> **"FOR" loops should always use an ordinal loop control variable (e.g., integer, char, boolean, enumerated type) and should always increment or decrement the loop control variable by one.**

Most people expect the execution of a loop to begin with the first statement at the top of the loop, therefore,

Rule:

> **All loops should have one entry point. The program should enter the loop with the instruction at the top of the loop.**

Likewise, most people expect a loop to have a single exit point, especially if it's a while or **repeat..until** loop. They will rarely look closely inside a loop body to determine if there are "break" statements within the loop once they find one exit point. Therefore,

Guideline:

> **Loops with a single exit point are more easily understood.**

Whenever a programmer sees an empty loop, the first thought is that something is missing. Worse yet, in languages like Pascal or C/C++ where you don't have a terminating ENDloop statement, it's easy to think that the next statement in the program is the body of the loop (worse yet, it's easy to forget the semicolon that marks the end of the loop and actually make the next statement in the program the loop's body). Therefore,

Guideline:

> **Avoid empty loops. If testing the loop termination condition produces some side effect that is the whole purpose of the loop, move that side effect into the body of the loop. If a loop truly has an empty body, place a comment like "/* nothing */" or "{null statement}" within your code.**

Even if the loop body is not empty, you should avoid side effects in a loop termination expression. When someone else reads your code and sees a loop body, they may skim right over the loop termination expression and start reading the code in the body of the loop. If the (correct) execution of the loop body depends upon the side effect, the reader may become confused since s/he did not notice the side effect earlier. The presence of side effects (that is, having the loop termination expression compute some other value beyond whether the loop should terminate or repeat) indicates that you're probably using the wrong control structure. Consider the following while loop from "C" that is easily corrected:

```
while ( ( ch = getc(stdin)) != 'A')
{
    << statements >>
}
```

A better implementation of this code fragment would be to use a loop..endloop construct:

```
for(;;) /* C/C++'s infinite loop statement */
{
    ch = getc(stdin);
    if (ch != 'A') break;

    << statements >>
}
```

An even better solution to the above would be to use the newer high level language constructs. See the C/C++ language-specific section for more details.

Rule:

> **Avoid side-effects in the computation of the loop termination expression (others may not be expecting such side effects). Also see the guideline about empty loops.**

Like functions, loops should exhibit functional cohesion. That is, the loop should accomplish exactly one thing. It's very tempting to initialize two separate arrays in the same loop. You have to ask yourself, though, "what do you really accomplish by this?" You save about four machine instructions on each loop iteration, that's what. That rarely accounts for much. Furthermore, now the operations on those two arrays are tied together, you cannot change the size of one without changing the size of the other. Finally, someone reading your code has to remember two things the loop is doing rather than one.

Guideline:

> **Make each loop perform only one function.**

Programs are much easier to read if you read them from left to right, top to bottom (beginning to end). Programs that jump around quite a bit are much harder to read. Of course, the goto statement is well-known for its ability to scramble the logical flow of a program, but you can produce equally hard to read code using other, structured, statements in a language. For example, a deeply nested set of if statements, some with and some without else clauses, can be very difficult to follow because of the number of possible places the code can transfer depending upon the result of several different boolean expressions.

Rule:

> **Code, as much as possible, should read from top to bottom.**

Rule:

> **Related statements should be grouped together and separated from unrelated statements with whitespace or comments.**

Enforced Rule:

> **GOTOs, if they appear at all in a program, must be okayed by a peer review of at least two peers, both of whom agree the resulting code with a GOTO is easier to understand than equivalent code without a GOTO. GOTOs should only be used in exception processing statements or after exhausting several other attempts at writing clear code without the GOTO. Of course some code is actually easier to read with a GOTO statement than without, but it is easy to develop a mental block that would suggest the use of a GOTO when a clearer solution exists, hence the peer review.**

2.6 - Expressions

Few things look so similar between different languages yet act so different as arithmetic expressions. Between various languages the precedence of operators is different, the associativity of operators is different, even the operation computed is often different. It goes without saying that different languages often use the same symbol for different operations and, likewise, use different symbols for the same operation. This creates a problem with a coding standard if the intent is to allow a Visual BASIC programmer to easily read a program written in C/C++ or Pascal. Although there are many issues that a coding standard cannot practically resolve, some standards can improve the situation.

One of the big areas where programming languages differ is how they handle operator precedence. For example, in C/C++ the "<<" and ">>" (shift left and shift right) operators have lower precedence than addition and subtraction. In Borland Turbo Pascal and Delphi, the "SHL" and "SHR" operators have higher precedence than addition and subtraction. Likewise, in many languages the relational operators all have the same precedence while in others they do not. The overly simplistic solution is to take the "Beginning Programmer Textbook" attitude of accepting the (almost) universal precedence relationship between addition, subtraction, multiplication, and division and requiring parentheses everywhere else. While this is, perhaps, a good starting point it often falls short in practice because some expressions wind up with too many parentheses (impairing the readability) when the intent would have been clear without them.

As a general rule, the reader of a program should be able to make the following assumptions about the operator precedence within a program:

- Operands have the highest precedence. This includes functions, variables (scalar, array element, and record field), constants, dereferenced pointers, etc.
- Unary operators
- Multiplication, division, and remainder (mod)
- Addition and subtraction
- Relational operators (may not all be the same precedence)
- Logical operators (and, or, may not be the same precedence)

As long as two adjacent operators in an expression belong to two different classes above, you can skip using parentheses. You can assume that addition, subtraction, multiplication, remainder and division are left associative. Therefore, if there are two adjacent operators are addition and subtraction, or multiplication, remainder, or division, then you can skip the parentheses. In all other cases, you must supply parentheses to explicitly state the precedence.

Rule:

The assumable precedences are: [highest]: {operands} {unary operators} {*,/,mod} {+.-} {<, <=, =, <>, >, >=} {and, or}. Note that you can only assume left associativity for {*,/,mod} and {+,-}. Assume all other operators are non-associative and that you must use parentheses if they are next to one another in an expression. If you cannot assume the precedence according to the rule above, use parentheses to explicitly state the precedence.

Some language use short-circuit evaluation, some use full evaluation of expressions. If your program uses and depends upon short-circuit evaluation, you will comment this fact next to each expression that requires short-circuit evaluation.

Rule:

If an expression depends upon short-circuit evaluation to produce a correct answer, you must explicitly state this in a comment nearby.

In most languages it is possible to produce side effects within an expression. You can accomplish this, for example, by passing a parameter by reference to a function or if the function modifies global variables. Since most languages give the compiler writer leeway with respect to the order of evaluation of expressions, you should never use a variable whose value is modified as a side effect of a function or operator within that expression (e.g., in C/C++ consider the statement "Y = X + Y + ++X;"). Even if you're sure the result will be correct, such code would be very difficult to understand.

Guideline:

An expression should not produce any side effects.

There are some obvious exceptions to the rule above. The whole purpose of some operators and functions is to produce a side effect. Examples include the "++" and "--" operators in C/C++ and any of the various assignment operators. A stronger rule to allow for this might be

Rule:

A program should never use the value of a variable modified as a result of a side effect within that same expression.

Never execute an expression solely for the side effects it produces. Programmers generally expect the value of an expression to carry some significance; they feel there would be no need to compute the value of an expression if that value were of no importance. If all you need are the side effects, find some other way to achieve those side effects. Example: What does the following C statement do? (This came out of a real program on the net.)

```
        *s++ || *s++ || *s++ || *s++ || s++;
```

Rule:

> **Never execute an expression solely for the side effects it produces.**

There are some mechanical issues regarding expressions that can make them easier to read. The following rules and guidelines document these issues:

Guideline:

> **There should be no spaces between a unary operator (e.g., "-") and the object on which it operates.**

```
        -x      *p      !b      /* from C/C++ */
```

Guideline:

> **There should be at least one space on either side of a binary operator.**

```
        x = *p + a / b;
```

Guideline:

> **Operators that select a component of a larger object (e.g., "." for records/structures and "[ ]" for arrays) should be adjacent to the object(s) they operate upon.**

```
        recname.field                   recptr->field  ary[ i ]
```

Guideline:

> **Objects that separate items (e.g., "," and ";") should immediately follow the previous object. If a second object follows the separator, there should be a space between the separator and the second object.**

```
        proc( parm1, parm2, parm3);
        procedure PascalProc( i:integer; b:boolean );
```

Guideline:

> **Bracketing symbols (e.g., "(" and ")", "[" and "]", and "{" and "}") should have one space on the "open" end of the symbol, that is, to the right of "(", "[", and "[" and to the left of ")", "]", and "}".**

```
        x := f( x + 2 * a[ i, j ] );
```

Some languages (C/C++ and Algol-68 come to mind) have a tremendous number of operators. Some of them are quite arcane and have no counterpart in other languages (when was the last time you used ">>=" or "->*" ?). If an alternative is available, you should avoid using assignments within expressions and other lesser-used operators.

2.7 - Program Layout

After naming conventions and where to put braces (or begin..end), the other major argument programmers engage in is how to lay out a program, i.e., what are the indentations one should use in a well written program? Unfortunately, the ideal program layout is something that varies by language. The layout of an easy to read C/C++ program is considerably different than that of an assembly language, Prolog, or Bison/YACC program. As usual, this section will describe those conventions that generally apply to all programs. It will also discuss layouts of the standard control structures described earlier.

According to McConnell (Code Complete), research has shown that there is a strong correlation between program indentation and comprehensibility. Miaria et. al ("Program Indentation and Comprehension") concluded that indentation in the two to four character range was optimal even though many subjects felt that six-space indentation looked better. These results are probably due to the fact that the eye has to travel less distance to read indented code and therefore the reader's eyes suffer from less fatigue.

Guideline:

> **Indentation should be three to four spaces in an indented control structure with four spaces probably being the optimal value.**

Enforced Rule:

> **If you use tabs to indent your code, insert a comment at the very beginning of the program that states the number of positions for each tab stop. E.g., "/* This program is formatted using four character position tabstops. */"**

Steve McConnell, in Code Complete, mentions several objectives of good program layout:

- The layout should accurately reflect the logical structure of the code. Code Complete refers to this as the "Fundamental Theorem of Formatting." White space (blank lines and indentation) is the primary tool one can use to show the logical structure of a program.
- Consistently represent the logical structure of the code. Some common formatting conventions (e.g., those used by many C/C++ programmers) are full of inconsistencies. For example, why does the "{" go on the same line as an "if" but below "int main()" (or any other function declaration)? A good style applies consistently.
- Improve readability. If the indentation scheme makes a program harder to read, why waste time with it? As pointed out earlier, some schemes make the program look pretty but, in fact, make it harder to read (see the example about 2-4 vs. 6 position indentation, above).
- Withstand modifications. A good indentation scheme shouldn't force a programmer to modify several lines of code in order to affect a small change to one line. For example, many programmers put a begin..end block (or "{".."}" block) after an if statement even if there is only one statement associated with the if. This allows the programmer to easily add new statements to the then-clause of the **if** statement without having to add additional syntactical elements later.

The principle tool for creating good layout is whitespace (or the lack thereof, that is, grouping objects). The following paragraphs summarize McConnell's finding on the subject:

- Grouping: Related statements should be grouped together. Statements that logically belong together should contain no arbitrary interleaving whitespace (blank lines or unnecessary indentation).
- Blank lines: Blank lines should separate declarations from the start of code, logically related statements

from unrelated statements, and blocks of comments from blocks of code.

- Alignment: Align objects that belong together. Examples include type names in a variable declaration section, assignment operators in a sequence of related assignment statements, and columns of initialized data.
- Indentation: Indenting statements inside block statements improves readability, see the comments and rules earlier in this section.

Rule:

At least one blank line must separate a comment on a line by itself from a line of code following or preceding the comment.

This style guide uses the "Pure Blocks" layout form suggested by McConnell. This is the obvious layout scheme to use when your language supports modern structured statements like if..then..elseif..else..endif. Since this standard requires the emulation of the modern block structured statements, the Pure Blocks layout is appropriate.

Rule:

The standard layout scheme for this coding standard is the Pure Block format. For languages that do not support modern structured control statements, this coding standard specifies an emulation of these statements that allows the use of the Pure Block layout format.

In theory, a line of source code can be arbitrarily long. In practice, there are several practical limitations on source code lines. Paramount is the amount of text that will fit on a given terminal display device (we don't all have 21" high resolution monitors!) and what can be printed on a typical sheet of paper. If this isn't enough to suggest an 80 character limit on source lines, McConnell suggests that longer lines are harder to read (remember, people tend to look at only the left side of the page while skimming through a listing).

Enforced Rule:

Source code lines will not exceed 80 characters in length.

If a statement approaches the maximum limit of 80 characters, it should be broken up at a reasonable point and split across two lines. If the line is a control statement that involves a particularly long logical expression, the expression should be broken up at a logical point (e.g., at the point of a low-precedence operator outside any parentheses) and the remainder of the expression placed underneath the first part of the expression. E.g.,

```
if
(
    ( ( x + y * z) < ( ComputeProfits(1980,1990) / 1.0775 ) ) &&
    ( ValueOfStock[ ThisYear ] >= ValueOfStock[ LastYear ] )
)

        << statements >>

endif;
```

Many statements (e.g., IF, WHILE, FOR, and function or procedure calls) contain a keyword followed by a parenthesis. If the expression appearing between the parentheses is too long to fit on one line, consider putting the opening and closing parentheses in the same column as the first character of the start of the statement and indenting the remaining expression elements. The example above demonstrates this for the "IF" statement. The following examples demonstrate this technique for other statements:

```
    while
    (
        ( NumberOfIterations < MaxCount ) &&
        ( i <= NumberOfIterations )
    )

        << Statements to execute >>

    endwhile;

    fprintf
    (
        stderr,
        "Error in module %s at line #%d, encountered illegal value\n",
        ModuleName,
        LineNumber
    );
```

Guideline:

> **For statements that are too long to fit on one physical 80-column line, you should break the statement into two (or more) lines at points in the statement that will have the least impact on the readability of the statement. This situation usually occurs immediately after low-precedence operators or after commas.**

For block statements there should always be a blank line between the line containing an **if, elseif, else, endif, while, endwhile, repeat, until,** etc., and the lines they enclose. This clearly differentiates statements within a block from a possible continuation of the expression associated with the enclosing statement. It also helps clearly show the logical format of the code. Example:

```
    if ( ( x = y ) and PassingValue( x, y ) ) then

        Output( 'This is done' );

    endif;
```

Rule:

> **Always put a blank line between any block statement and the statement(s) it encloses.**

If a procedure, function, or other program unit has a particularly long actual or formal parameter list, each parameter should be placed on a separate line. The following (C/C++) examples demonstrate a function declaration and call using this technique:

```
    int
    MyFunction
    (
        int    NumberOfDataPoints,
        float  X1Root,
        float  X2Root,
        float  &YIntercept
    );


    x = MyFunction
        (
            GetNumberOfPoints(RootArray),
            RootArray[ 0 ],
            RootArray[ 1 ],
            Solution
        );
```

Rule:

If an actual or formal parameter list is too long to fit a function call or definition on a single line, then place each parameter on a separate line and align them so they are easy to read.

2.8 - Comments and (program) Documentation

Almost everyone agrees that a program should have good comments. Unfortunately, few people agree on the definition of a good comment. Some people, in frustration, feel that minimal comments are the best. Others feel that every line should have two or three comments attached to it. Everyone else wishes they had good comments in their program but never seem to find the time to put them in.

It is rather difficult to characterize a "good comment." In fact, it's much easier to give examples of bad comments than it is to discuss good comments. The following list describes some of the worst possible comments you can put in a program (from worst up to barely tolerable):

- The absolute worst comment you can put into a program is an incorrect comment. Consider the following Pascal statement:

```
A := 10;  { Set 'A' to 11 }
```

- It is amazing how many programmers will automatically assume the comment is correct and try to figure out how this code manages to set the variable "A" to the value 11 when the code so obviously sets it to 10.
- The second worst comment you can place in a program is a comment that explains what a statement is doing. The typical example is something like "A := 10; { Set 'A' to 10 }". Unlike the previous example, this comment is correct. But it is still worse than no comment at all because it is redundant and forces the reader to spend additional time reading the code (reading time is directly proportional to reading difficulty). This also makes it harder to maintain since slight changes to the code (e.g., "A := 9") requires modifications to the comment that would not otherwise be required.
- The third worst comment in a program is an irrelevant one. Telling a joke, for example, may seem cute, but it does little to improve the readability of a program; indeed, it offers a distraction that breaks concentration.
- The fourth worst comment is no comment at all.
- The fifth worst comment is a comment that is obsolete or out of date (though not incorrect). For example, comments at the beginning of the file may describe the current version of a module and who last worked on it. If the last programmer to modify the file did not update the comments, the comments

are now out of date.

Steve McConnell provides a long list of suggestions for high-quality code. These suggestions include:

- Use commenting styles that don't break down or discourage modification. Essentially, he's saying pick a commenting style that isn't so much work people refuse to use it. He gives an example of a block of comments surrounded by asterisks as being hard to maintain. This is a poor example since modern text editors will automatically "outline" the comments for you. Nevertheless, the basic idea is sound.
- Comment as you go along. If you put commenting off until the last moment, then it seems like another task in the software development process and management is likely to discourage the completion of the commenting task in hopes of meeting new deadlines.
- Avoid self-indulgent comments. Also, you should avoid sexist, profane, or other insulting remarks in your comments. Always remember, someone else will eventually read your code.
- Avoid putting comments on the same physical line as the statement they describe. Such comments are very hard to maintain since there is very little room. McConnell suggests that endline comments are okay for variable declarations. For some this might be true but many variable declarations may require considerable explanation that simply won't fit at the end of a line. One exception to this rule is "maintenance notes." Comments that refer to a defect tracking entry in the defect database are okay (note that the CodeWright text editor provides a much better solution for this -- buttons that can bring up an external file). Endline comments are also useful for marking the end of a control structure (e.g., "end{if};").
- Write comments that describe blocks of statements rather than individual statements. Comments covering single statements tend to discuss the mechanics of that statement rather than discussing what the program is doing.
- Focus paragraph comments on the why rather than the how. Code should explain what the program is doing and why the programmer chose to do it that way rather than explain what each individual statement is doing.
- Use comments to prepare the reader for what is to follow. Someone reading the comments should be able to have a good idea of what the following code does without actually looking at the code. Note that this rule also suggests that comments should always precede the code to which they apply.
- Make every comment count. If the reader wastes time reading a comment of little value, the program is harder to read; period.
- Document surprises and tricky code. Of course, the best solution is not to have any tricky code. In practice, you can't always achieve this goal. When you do need to restore to some tricky code, make sure you fully document what you've done.
- Avoid abbreviations. While there may be an argument for abbreviating identifiers that appear in a program, no way does this apply to comments.
- Keep comments close to the code they describe. The prologue to a program unit should give its name, describe the parameters, and provide a short description of the program. It should not go into details about the operation of the module itself. Internal comments should to that.
- Comments should explain the parameters to a function, assertions about these parameters, whether they are input, output, or in/out parameters.
- Comments should describe a routine's limitations, assumptions, and any side effects.

Rule:

All comments will be high-quality comments that describe the actions of the surrounding code in a concise manner

Enforced Rule:

> **All comments will be up to date. If a programmer makes changes to the code, that programmer is responsible for updating the internal comments and any external documentation affected by those changes.**

2.9 - Unfinished Code

Often it is the case that a programmer will write a section of code that (partially) accomplishes some task but needs further work to complete a feature set, make it more robust, or remove some known defect in the code. It is common for such programmers to place comments into the code like "This needs more work," "Kludge ahead," etc. The problem with these comments is that they are often forgotten. It isn't until the code fails in the field that the section of code associated with these comments is found and their problems corrected.

Ideally, one should never have to put such code into a program. Of course, ideally, programs never have any defects in them, either. Since such code inevitably finds its way into a program, it's best to have a policy in place to deal with it, hence this section.

Unfinished code comes in four general categories: non-functional code, partially functioning code, suspect code, and code in need of enhancement. Non-functional code might be a stub or driver that needs to be replaced in the future with actual code or some code that has severe enough defects that it is useless except for some small special cases. This code is really bad, fortunately its severity prevents you from ignoring it. It is unlikely anyone would miss such a poorly constructed piece of code in early testing prior to release.

Partially functioning code is, perhaps, the biggest problem. This code works well enough to pass some simple tests yet contains serious defects that should be corrected. Moreover, these defects are known. Software often contains a large number of unknown defects; it's a shame to let some (prior) known defects ship with the product simply because a programmer forgot about a defect or couldn't find the defect later.

Suspect code is exactly that- code that is suspicious. The programmer may not be aware of a quantifiable problem but may suspect that a problem exists. Such code will need a later review in order to verify whether it is correct.

The fourth category, code in need of enhancement, is the least serious. For example, to expedite a release, a programmer might choose to use a simple algorithm rather than a complex, faster algorithm. S/he could make a comment in the code like "This linear search should be replaced by a hash table lookup in a future version of the software." Although it might not be absolutely necessary to correct such a problem, it would be nice to know about such problems so they can be dealt with in the future.

The fifth category, documentation, refers to changes made to software that will affect the corresponding documentation (user guide, design document, etc.). The documentation department can search for these defects to bring existing documentation in line with the current code.

This standard defines a mechanism for dealing with these five classes of problems. Any occurrence of unfinished code will be preceded by a comment that takes one of the following forms (where "@" denotes the standard comment delimiters in a given language and "_" denotes a single space):

```
@_#defect#severe_@
@_#defect#functional_@
@_#defect#suspect_@
@_#defect#enhancement_@
@_#defect#documentation_@
```

It is important to use all lower case and verify the correct spelling so it is easy to find these comments using a text editor search or a tool like grep. Obviously, a separate comment explaining the situation must follow these comments in the source code.

Examples in various languages:

Pascal/Delphi:

```
(* #defect#severe *)
{ #defect#enhancement }
(* #defect#functional *)
{ #defect#suspect }
{ #defect#documentation }
```

C:

```
/* #defect#severe */
/* #defect#suspect */
/* #defect#documentation */

C++:
/* #defect#functional */
// #defect#enhancement //

BASIC:
' #defect#functional '

Assembly (80x86):
; #defect#suspect ;

Ada:
-- #defect#enhancement --
-- #defect#documentation --
```

Notice the use of delimiters on both sides even if the language, technically, doesn't require them (C++. BASIC, assembly, and Ada).

Enforced Rule:

> **If a module contains some defects that cannot be immediately removed because of time or other constraints, the program will insert a standardized comment before the code so that it is easy to locate such problems in the future. The four standardized comments are "@_#defect#severe_@, "@_#defect#functional_@", "@_#defect#suspect_@", "@_#defect#enhancement_@", and "@_#defect#documentation_@" where "@" denotes the comment delimiter and "_" denotes a**

> **single space. The spelling and spacing should be exact so it is easy to search for these strings in the source tree.**

2.10 - Cross References in Code to Other Documents

In many instances a section of code might be intrinsically tied to some other document. For example, you might refer the reader to the user document or the design document within your comments in a program. This document proposes a standard way to do this so that it is relatively easy to locate cross references appearing in source code. The technique is similar to that for defect reporting, except the comments take the form:

```
                @  text #link#location text @
```

The "@" represents the comment delimiters. "Text" is optional and represents arbitrary text (although it is really intended for embedding html commands to provide hyperlinks to the specified document). "Location" describes the document and section where the associated information can be found.

```
Examples:
C/C++:

/* #link#User's Guide Section 3.1 */
// #link#Program Design Document, Page 5 //

Pascal:

(* #link#Funcs.pas module, "xyz" function *)
{ <A HREF="DesignDoc.html#xyzfunc"> #link#xyzfunc </a> }
```

Guideline:

> **If a module contains some cross references to other documents, there should be a comment that takes the form "@ text #link#location text @" that provides the reference to that other document. In this comment, the "@" represents the language's comment delimeter(s), "text" represents some optional text (typically reserved for html tags), and "location" is some descriptive text that describes the document (and a position in that document) related to the current section of code in the program.**

< Previous section *(1 - Introduction)* - **Contents** - **Next section** *(3 - C (and related C++) Specific Issues) >*

Software Development Guidelines

3 - C (and related C++) Specific Issues

< Previous section *(2 - General Programming Guidelines)* - **Contents** - **Next section** *(4 - Pascal/Delphi Specific Formatting Issues) >*

3 - C (and related C++) Specific Issues

This coding standard will probably offer the biggest challenge to C/C++ programmers since conventional C/C++ programming practices are pervasive and unusual (compared to other high level languages). This document recommends some unusual solutions that will improve the C/C++ languages by modernizing their control structures. The fact that C/C++ is flexible enough to allow such an extension speaks highly of the language. On the downside, these changes mean that current C/C++ programmers will have to adopt to a slightly different programming paradigm in order to meet the standard.

The first step is to move the C/C++ language into the 21st century with respect to control structures. With respect to programming language design, C is a very old language. C++ is certainly newer, but still carries around a lot of old baggage for the sake of upwards compatibility. One primary drawback to C/C++ is that it doesn't directly support modern control structures like **if..then..elseif..else..endif**, while..endwhile, **repeat..until, loop..endloop, for..endfor, context..endcontext,** and a modern **switch**. Fortunately, C/C++ does provide macro facilities through which we can construct versions of all these statements. Adopting these macros into your C/C++ programs may seem like an annoyance at first, but they offer many major advantages over existing C/C++ statements; plus they eliminate the religious argument about the placement of "{" and "}" after a statement.

The following macros (from the "ratc.h" header file) provide the constructions for each of these statements. See the "ratc.h" file for details on how to implement these constructs.

Rule:

> **All C/C++ programs will use the control structures found in the "ratc.h" (RATional C) header file in place of the traditional C/C++ control structures.**

3.1 - Repeat..Until Statement

The **repeat..until** statement in C/C++ takes the following form:

```
    _repeat

        << statements >>

    _until ( boolean_expression )
```

There must be one blank line between the "_repeat" and the first statement in the body of the loop. Likewise, there must be one blank line between the last statement in the body of the loop and the "_until" clause. You should indent the loop body statements two to four characters.

This loop executes the body of the loop at least once and then tests the value of the expression. If the expression evaluates to false the loop repeats. The loop terminates when the expression evaluates to true. Some people complain that the loop termination condition should be false in order to match the termination condition of the while loop. There is very little empirical evidence (I am aware of) to suggest that this affects the readability of "test at the bottom" loops one way or the other.

3.2 - The Loop..Endloop Statement

The cryptic "for(;;)" statement has been the traditional way to create an infinite loop in C/C++ since the earliest version of the K&R reference. The "_loop" and "_endloop" macros in the "ratc.h" header file encapsulate this strange structure to make it more readable.

A typical _loop.._endloop statement takes the following form:

```
    _loop

        << statements >>

    _endloop
```

There should be at least one blank line between the "_loop" and the first statement of the loop body. Likewise, there should be at least one blank line between the last statement of the loop body and the "_endloop" clause. You should indent the loop statements two to four characters.

The _loop.._endloop statement creates an infinite loop. A typical use of the _loop.._endloop statement is to create loops whose termination test occurs somewhere in the middle of the loop body. You can use C/C++'s existing break and continue statements to exit the loop or jump to the top of the loop. You can also use the "_breakif" statement described next.

3.3 - The Breakif Statement

The "_breakif" macro combines a test for loop termination with the C/C++ "break" statement. This statement takes the form:

```
        _breakif ( expression );
```

If the expression evaluates false, the program ignores this statement. If the expression evaluates to true, then control transfers to the first statement beyond the immediately enclosing loop construct.

Using the "_breakif" statement is slightly superior to the if statement above because it is easier to determine the purpose of the instruction since the reader doesn't have to scan past the expression to see the "break" statement.

The "_breakif" statement is especially useful for terminating a "_loop.._endloop" iteration. This is the typical statement one would use to terminate loop execution in the middle of a loop.

3.4 - The While Statement

The "_while" and "_endwhile" macros provide an implementation of a modern while loop. The syntax of this statement takes the following form:

```
    _while ( expression )

        << statements >>

    _endwhile
```

The statements that comprise the loop body should be indented 2-4 spaces. Also, there should be a blank line between the "_while" and the first statement of the loop body, there should also be a blank line between the last statement in the loop body and the "_endwhile".

3.5 - The For..Endfor and Downto..Endfor Loops

These two statements implement the standard versions of the for loop that iterate from a smaller value to a larger value by one and iterate from a larger value down to a smaller value by minus one. These two loops take the following forms:

```
    _for( var, start, stop )

        << statements >>

    _endfor;

    _downto( var, start, stop )

        << statements >>

    _endfor;
```

There should be a blank line between the _for, _downto, and _endfor clauses and any statements within the loop body. The loop body statements should be indented 2-4 characters within the loop.

Note: These variants of the "for" loop severely restrict the capabilities of C/C++'s for loop. This is good! If you need a fancier type of for loop, construct it using a while loop, do not revert back to C/C++'s "out of control" for loop. Doing so makes your programs harder to read.

3.6 - If..Elseif..Else..Endif Statement

The "ratc.h" macro file includes a set of macros to add a sophisticated **if..elseif..else..endif** statement to the C/C++ programming languages. This statement takes the following form:

```
    _if ( expression )

        << then statements >>
    _elseif ( expression )  /* optional, may be repeated */

        << elseif statements >>

    _else               /* optional */

        << else statements >>

    _endif
```

The "_elseif" clause is optional; the program may have any number of "_elseif" clauses. The "_else" section is also optional, although the statement may contain only one "_else" clause.

There should be a blank line between the statement blocks in the code above (i.e., << then statements >>, << elseif statements >>, and << else statements >>) and the corresponding delimiting characters. The following code sequences provide some examples of the "_if.._elseif.._else.._endif" statement:

```
/* Note, this is an example only, there are obviously better ways  */
/* to do the following                                             */

    _if
    (
        (( ch >= 'A' ) and ( ch <= 'Z' )) ||
        (( ch >= 'a' ) and ( ch <= 'z'))
    )

        ch = ( ch xor 0x20h );    /* swap case */

    _elseif (( ch >= '0' ) and ( ch <= 9 ))

        value = ch & 0xF;

    _else

        PrintError();

    _endif


    /* The _elseif and _else sections are optional, as the */
    /* next example shows.                                 */

    _if ( isAlpha(ch) )

        ch = ( ch xor 0x20 );

    _endif
```

3.7 - The Switch..EndSw Statement

C/C++'s switch statement is one of the least structured statements in the language. A typical example of a "_switch.._endsw" statement might be:

```
    _switch ( ch )

      _case 'A':

        << A statements >>

      _endcase

      _case 'B':

        << B statements >>

      _endcase

      _case 'Q':

        << Q statements >>

      _endcase

      _default:

        << Default statements >>

    _endsw
```

The first thing to note is that there are no "break" statements in the code above. The "_endcase" macro quietly inserts a break for you.

Consider the following C/C++ switch statement:

```
    switch ( ch )
    {
      case 'A':
        DoAStuff();
        break;

      case 'B':

        DoBStuff();
        break;

      case 'Q':
      case 'X':

        return ( 0 );

      default:

        PrintError();

    }
```

This example demonstrates the one situation where, stylistically, it is okay for one case to drop down into another. To simulate this with the "_switch" statement you would use the following code:

```
    _switch ( ch )

      _case 'A':

        DoAStuff();

      _endcase

      _case 'B':

        DoBStuff();

      _endcase

      _case 'Q':
      _case 'X':

        return ( 0 );

      _endcase          /* Some compilers may complain about this  */
                        /* because the code is unreachable.        */

      default:

        PrintError();

    _endsw
```

The comment in the code describes a minor problem that may occur. Some compilers will warn you if there is unreachable code in your program. The "break" statement the "_endcase" emits will be unreachable, hence the warning. Short of leaving off the "_endcase" (probably a worse style violation than accepting a warning from the compiler), there is little you can do about this.

3.8 - The _context.._endcontext, _leave, and _return Statements

There are a few occasions when a goto is warranted in a C/C++ program. Typically, this occurs when breaking out of several loops or when handling an exceptional condition that requires transfer to the end of a function. The _context.._endcontext block provides a structured way to handle this situation. The "_context" and "_endcontext" clauses surround a block of statements you want to exit under some condition. Once you define a context with these statements, you can exit the context with the _leave and _return statements. The syntax for the _context.._endcontext statement is

```
_context( Unique_Label )

    << Statements within the context >>

_endcontext( Unique_Label )
```

The labels above should be the same label, but should otherwise be unique in the current function (technically, the compiler ignores the label in the _context clause, but for readability you should always ensure that the two labels match). You should indent all statements between the _context and _endcontext clauses four character positions.

The _context and _endcontext clauses normally have no effect on program execution. That is, in the absence of an _leave or _return statement within the context, the program will executes the statements sequentially as though the _context statement were not present. However, should the program execute an _leave(Unique_Label) statement within the context, control immediately transfers to the first statement after the corresponding _endcontext(Unique_Label) clause.

The _return statement uses the following syntax:

```
_return( expression, Unique_label );
```

This statement assigns the value of "expression" to a variable named "Result". You must define the variable "Result" within the current function. Presumably, this it has the same type as the function return result. A typical use of the "_return" statement would be to transfer control to the end of a function to allow clean-up before actually returning from the function. For example, consider the following code:

```
int SomeFunc()
{
    int Result;
    char *Pointer;

    _context( SF_context );

        Pointer = malloc( 256 );
            .
            .
            .
        _if( x == y )
```

```
            _return( x+2, SF_context );

        _endif
          .
          .
          .
    _endcontext( SF_context );
    free( Pointer );
    return Result;
}
```

As you can see, the use of _return allows you to set up the function return result and still jump to clean-up code (in this case, simply freeing up the storage allocated for "Pointer") before returning from the function. Of course, if you use the _return statement, you must make sure that you assign the function's return value to the "Result" variable before attempting to return from the function. Note that most C/C++ compiler optimizers will remove this actual assignment from the code, so there is rarely a performance penalty when using this technique.

An interesting use of the _context.._endcontext statement is that it lets you check to see if you've terminated a loop via the loop termination condition or via a break (_leave) statement. Consider the following while loop:

```
_context( whl_cntxt )

    _while( expression )
            .
            .
            .
        _if( expression2 )

            _leave( whl_cntxt );

        _endif
            .
            .
            .
    _endwhile

    << statements that execute if expression is false >>

_endcontext( whl_cntxt );
```

Note that the "<<..>>" marked statements above do not execute if the code breaks out of the loop via the _leave statement. This fact is quite useful on many occasions.

Warning: The _context.._endcontext statement is really nothing more than a specialized form of a GOTO statement. Although it is structured and, therefore, safer to use than a straight goto, you can still obfuscate your code if you use too many _context.._endcontext statements within a function. This is especially true if you have several of them nested or there is a large number of statements between the _context and the corresponding _endcontext. Should this happen, you should consider converting the statements in the _context.._endcontext statement to a function (if appropriate to do so).

3.9 - Operators

Dennis Ritchie designed the C programming language around 1971 when most programming was still done on a 10 cps teletype machine. As such, he developed a terse language that minimized the number of keystrokes one had to type to enter a program. Unfortunately, this terseness led to the creation of similar operators with similar names that are easily confused; furthermore, the lexemes chosen were completely different from other those that other languages employ, adding to the confusion. C++ made this problem worse by introducing function overloading.

The "ratc.h" macro file contains a few macro definitions to overcome some of C/C++'s deficiencies. These macros include the following:

```
#define and &&
#define or  ||
#define not !
#define ifx(x,t,f) ((x) ? (t) : (f))
```

There are many other possible macros that could be written. They were not included for the following reasons:

- SHL ("<<") and SHR (">>") would seem to be possible candidates. However, inclusion of these operators could introduce subtle bugs in C++ code. Consider the following C++ statement:

```
                cout <<  2 shl 8;
```

Normally, one would expect this to print 256; however, it prints "28" instead. The reason is because this code expands to:

```
                cout << 2 << 8;
```

Since the "<<" operator is left associative, C++ evaluates the "cout << 2" expression first. This produces an "OSTREAM << 8" result that prints the "8". Because of the possible confusion in C++ programs, defining SHL and SHR seems like a bad idea.

- Replacing "&x" (address of x) with something like "adrs(x)" would also seem to make sense considering the number of places C/C++ already uses the "&" symbol. However, taking the address of an object is an extremely common operation in C programs, so it's questionable how welcome such a change would be to most programmers.

The other C/C++ operators are either reasonable or have no counterparts in other languages.

3.10 - Modules in C/C++

Modules in C/C++ physically consist of a ".c" source file[1] and a corresponding ".h" source file. C/C++ files associate one of four namespace attributes with identifiers within the source code: local names, static names, global names, or class/structure names (extern names are the same as global names). The proper use of these namespace attributes lets C/C++ programmers implement information hiding and abstract data types. This section will describe how to use these features in C. C++ also offers the availability of namespaces, this document will not consider namespaces here. Furthermore, C++ also offers direct support for abstract data types via classes. This section will not deal with that aspect of information hiding.

Local names are any object identifiers declared within a function in a C/C++ program (this section will not address locals appearing within compound statements in a function). Such names are not available outside the function and typically hold temporary results of interest only to the function enclosing them. The proper use of local variables is one of the first programming rules beginning programmers are taught, there is little need to expand on that discussion here.

Function and variable names declared outside of any function can be either static or global. By default, all such identifiers are global. The C/C++ keyword static precedes any static declarations. The scope of static objects is limited to the current source file; that is, you cannot reference a static object from another module. Global objects, on the other hand, are public and their names are available (via the linker) to other modules. Unfortunately, C/C++'s default is backwards. As a general rule an object should only be public if the programmer explicitly requests it to be so. Good programming style dictates information hiding - that is, publishing only those names and data structures that other modules need to use and keeping the remaining names private. To accomplish this, you must precede all private objects with the static storage class specifier.

Rule:

> **All private objects (that is, variable and function names that should remain local to a given source file) must have the keyword "static" preceding them in a C/C++ source file. If the "static" keyword is not present, the reader must be able to assume that the object is a public object that can be used by code in other modules.**

Leaving off the keyword "static" is only half the work necessary to make a name available in another module. The other half of the work is defining that object in the other module using an "extern" storage class specifier. To avoid maintenance problems, the only acceptable way to incorporate such external objects is via a "header" or ".h" file. If a module with the name xyz.c exports any names, then there will be a corresponding xyz.h file that contains the external declarations and any public constants and type declarations. Under no circumstances should a programmer insert "extern" definitions directly into a ".c" file; programs using this technique are very hard to maintain.

Enforced Rule:

> **All intermodule communication in C/C++ programs must take place through header files (".h" files). All extern directives, public class definitions, public type definitions, and public constants must appear in the header file that all interested modules will include.**

ANSI C and C++ support the notion of a function prototype. A prototype provides a mechanism whereby the compiler can perform stronger type checking on functions that a module calls before their declaration. Prototypes, if organized properly, can also improve the readability of a module. Proper organization consists of nothing more than placing all prototypes together near the beginning of the module. The best organization is order the prototypes in the order that the functions appear in the source module. This makes it easier for a programmer to manually locate a particular function in a listing by first locating its prototype at the beginning of the module and noting the functions around it.

Enforced Rule:

> **All functions (public or private) appearing in a source module will have an associated prototype. The prototypes for all functions will appear near the beginning of the source file (typically after the include and define directives and any other type definitions also appearing there).**

Guideline:

The order of the prototypes should match the order of the functions appearing in the source module.

Rule:

Functions that are also public (non-static) should appear near the beginning of the source file.

3.11 - Coding for Testability in C/C++

C/C++ provides two facilities that make it easy to write self-testing code: assertions and conditional compilation. An assertion is simply a macro that has a boolean expression as its parameter. The assert macro evaluates this expression and aborts the program with an error message if the expression is false. The conditional compilation directives let you include or exclude code based on various conditions. This lets you put in special debugging and testing statements that you can easily remove by changing a single global value.

3.11.1 - The Assert Macro

The assert macro expands one of two ways, normally it expands to some code that tests the expression and aborts the program if the expression is false. If the NDEBUG symbol is defined prior to the "#include <assert.h>" statement, then asserts (effectively) expand to a no-operation. Normally, the NDEBUG symbol should not be defined; this will produce a debugging/testing version of the software. When compiling a production version of the software, this symbol must be defined to ensure that you don't ship any debugging code embedded in a production system.

Guideline:

For internal use, all compiles should expand all assertions to abort the program if the assertion turns out to be false.

Rule:

The makefile for a given project must offer a "standard/debug" compilation and a "production" compilation option. The "production" compilation option should define the macro symbol NDEBUG for every C source file it processes.

The assert macro displays the filename containing the assertion, the line number of the assertion, and the text of the expression whenever the program aborts. For example, if the statement "assert(x == y);" is sitting at line number 25 in the file "xyz.c" and "x==y" evaluates false, the C run-time system will print (among other things): "x==y file: xyz.c line 25." Although this is useful information, some additional information might be helpful. This is especially true if an expression like "x==y" occurs in several different assertion statements. Assertions don't normally provide the ability to attach a string to the output message, but there is a trick you can use to print a string literal along with the (text of the) expression. When an assertion fails, the C run-time system prints everything between the parentheses in the assertion statement. Unfortunately, the C preprocessor removes comments before assert gets a look at the parameter list, otherwise we'd be able to stick a comment in the assertion and print it. A second solution is to put a string in the assertion parameter list. One way to do this is to use an assertion call like the following:

```
assert( ( "Assertion Message", x == y ) );
```

This particular statement, if the assertion fails, prints '("Assertion Message" , x == y)' along with the values for the line number and filename. There is nothing magic going on here; the comma operator tells C to ignore the first operand and use the value of the second ("x == y"). The parentheses are necessary around the whole bit so the C compiler does not confuse the "x == y" clause with a second parameter.

Since you can easily remove them by defining the symbol NDEBUG, assertions are cheap insurance and you should use them liberally to check input parameters to any functions you write, to check return values from functions you call, and to handle the default ("can't occur") case in **if..then..elseif..elseif..else..endif** and **switch..endswitch** statements.

Rule:

> **You should use assertions throughout your code to check degenerate and "impossible" conditions. You should also use assertions to check the sanity of parameters input to a function.**

Some people might argue against assertions claiming that they clutter up the listing. However, an editor like Codewright can remove lines containing assertion statements if this proves to be a problem. The benefit of having assertions in your code far outweighs the disadvantages.

3.11.2 - The RatC _affirm and _claim Macros

RatC provides two macros that offer a "softer" implementation of assert: _affirm and _claim. These two macros test an expression and print a string if the expression is false (just like assert). Unlike assert, however, they do not abort the program if the expression is false; they simply print their strings to the standard error device and then execution continues.

The _claim macro takes a single parameter -- the expression to test. If the expression evaluates true, then _claim does nothing. However, if the expression evaluates false, then the _claim macro will print the line number and the name of the file along with the text corresponding to the expression.

The _affirm macro takes two parameters -- and expression and a literal string constant. If the expression evaluates to true, the _affirm macro does nothing. If the expression evaluates false, the _affirm macro prints the filename, the line number of the _affirm statement, and the string literal.

Like the assert statement, all of the _claim and _affirm calls in your program will disappear if you define the NDEBUG symbol. Therefore, you can easily remove this code from your program for production purposes.

3.11.3 - A Convenient Way to Test a Function Return Result With Assert.

Suppose you have a function f(x) that returns zero if it is successful and any other value (typically an error code) if it is unsuccessful. You can easily attach an assertion to a call to f as follows:

```
                f( x ) _passert( "f(x) failed" );
```

This is functionally equivalent to the following C code:

```
    _if ( f( x ) )

        assert(!*"f(x) failed");
    _endif;
```

Why should we prefer the "_passert" version over the "_if" version? There are a couple of reasons. First of all, keep in mind that assertions typically check for exceptional conditions (meaning they rarely occur). Consider the malloc function; it returns a pointer to an allocated object on the heap if it succeeds, it returns NULL if a (rarely occurring) out of memory condition exists. Consider the following calls to malloc:

```
    ptr = (char *) malloc( NumberOfBytes );

vs.

    _if ( ptr = (char *) malloc( NumberOfBytes ) )

        fprintf( stderr, "Memory allocation failure\n");
        abort();

    _endif;
```

Which sequence above do you feel is easiest to read? Obviously, the first one it easier to read; it is also a lot easier to write and maintain. Of course, there is one major drawback -- it doesn't check the error return code. The vast majority of the time the first call to malloc above will work properly. However, once in a great while the program might actually run out of memory and then it would probably crash since it doesn't consider the out of memory condition. Obviously, this is unacceptable in commercial quality code.

To work out a solution, consider why the first statement above is more readable. One might argue (correctly) that the first version is more readable than the second because is it shorter. However, it's easy to make the second version much shorter by using a macro. It wouldn't be hard to write the second version as follows:

```
    asrt( ptr = (char *) malloc( NumberOfBytes ), "Malloc failure");
```

This still isn't as readable as the first version above. The primary reason is because the real work (allocating storage for a string and storing its address into ptr) occurs in the middle of this statement rather than at the beginning of the statement. A program will be easier to read if the real work of each statement occurs at the beginning of the statement. This is why the "_if (f(x)) ..." statement earlier isn't as easy to read as the "f(x) _passert..." version. The structure of the "_if" version implies that there is an important test that is necessary to the logic of the program. This is true in an absolute sense, but in a relative sense we would like to push error checking into the background and concentrate on the main computations the program performs. Having to worry about degenerate cases all the time simply confuses the reader.

The "ratc.h" header file defines two macros: "_passert" (positive assertion) and "_nassert" (negative assert). Both expect some boolean expression to precede the macro. If the expression evaluates to false, "_passert" does nothing. Likewise, if the expression's value is true then "_nassert" does nothing. If the expression evaluates to the opposite value, then these two functions abort the program printing the single string literal value passed to them.

The important thing to keep in mind about these operators is that they expect an arithmetic expression immediately preceding the macro invocation. These operators have the same precedence as the C "?" ternary operator.

Rule:

> **If a function can succeed or fail in addition to returning some value, the function should return the failure status as the function result and return the other value through a reference parameter. This allows you to use the _assert and _nassert macros to check the return status of the function.**

3.11.4 - Special Note for C++ Users

If you glance back at the descriptions in the last section, you'll notice that it refers to "C" not "C/C++." That's because in many instances, C++ provides a better solution: exception handling. By using the try, catch, and throw statements in C++ you can raise an exception inside your function and handle it at some point other than immediately upon return. Since exceptions should rarely occur, this is much better way to handle degenerate cases in the program.

Rule:

> **If your language provides exception handling capabilities, use them rather than manufacturing your own tests for exceptional conditions.**

3.11.5 - Using Conditional Compilation

Assertions provide a good check against a disastrous condition. Should an assertion fail, the program stops with a message that indicates the source of the problem. Sometimes, however, you might want to insert some debugging code that logs some information to a file (or the standard error output device) without stopping the execution of the program. Assertions, since they terminate program execution, are unsuitable for this. A better solution is to use the #ifdef and #ifndef directives to include or not include debugging code.

Rule:

> **All debugging code must disappear if the symbol "NDEBUG" is defined. That is, you must surround all debugging code with "#ifndef NDEBUG" and a corresponding #endif.**

The single symbol "NDEBUG" eliminates all assertions in a program. Therefore, it makes sense to say that this single symbol should also eliminate all debugging code if it is defined. This allows whoever builds a production copy of the C code to feel they have confidently eliminated all debugging code by defining a single symbol during compilation.

Unlike assertions, you may not want to have all debugging statements active at one time. For example, if a particular module has five sets of debugging statements in it, you may only be interested in one set or the other. With the NDEBUG symbol, it's all or nothing; either all the debug statements are present or none of them are present. The solution is to associate a name with the current debug block (or a set of debug blocks) and use a nested #ifdef statement to activate or deactivate the debugging code. Consider the following code:

```
#ifndef NDEBUG    /* To turn off debugging for production code */
#ifdef DebugCalc

    fprintf(debugLogFile,"Appropriate Logging Output\n");

#endif
#endif
```

Normally, you should indent #ifdef and #ifndef statements just like any statement in C/C++. That would suggest that the style above is incorrect. However, since these statements will always occur in pairs, an exception is quite justifiable in this particular case.

It is tempting to create a macro that will emit the #ifndef and #ifdef statements above. Avoid this temptation! The Codewright editor on the PC has a nifty feature that automatically removes #ifdef/#ifndef code that wouldn't normally be compiled. Unfortunately, Codewright only looks for #ifdef and #ifndef; it isn't smart enough to expand macros when searching for these statements. A major problem with #ifdef and #ifndef is that they tend to clutter up your code. Codewright's ability to selective display sections of code (and not display others, like the #ifdef code) is a very handy feature.

Guideline:

> **Don't avoid the use of #ifdef and #ifndef statements in your program because you are worried about making your program harder to read. Tools exist to remove these #ifdef and #ifndef statements thus eliminating the clutter.**

Debugging statements programmers put into their programs generally fall into three categories:

- One-time blocks of code that help you track down an eliminate a single bug (and, obviously, never appear in production code).
- Debugging statements that log current values in a module for later perusal to see if a section of code is getting and computing reasonable values. This code never appears in the production version of the program.
- Debugging statements that log values or provide information on demand in a production version of a program. A field engineer, for example, might use such code to test the program at a customer's site.

The first category of statements above (one-time blocks) normally shouldn't exist - you should really be using a debugger to track down these types of problems. However, not all debuggers are flexible enough to handle every situation and a small block of debugging code can save the day. These debugging statements are the most dangerous to insert into a program. Often, the programmer sees these statements as temporary additions to the code that s/he will remove momentarily. Unfortunately, such statements invariably wind up in production code because the programmer, in haste, forgot to remove the debugging statements and failed to protect them with a #ifndef NDEBUG statement.

Enforced Rule:

> **All "temporary" debugging statements you add to a program, no matter how temporary they seem, must be protected with "#ifndef NDEBUG" and "#endif" statements. This is the only line of defense against forgetting to remove the code before compiling a production version of the program.**

The second category of debug statements above will typically print parameters passed into a function, print computations during the execution of a function, print return results from a called function, and print results that the current function returns. Programmers often insert such statements into their programs as temporary measures to track down an elusive defect in the software. However, such statements should be coded in as permanent fixtures in a program. After all, once you've tracked down a current problem with a routine, there is no guarantee that there won't be future problems with that routine. If you leave the logging code in place, you will not have to rewrite it the next time you want to trace a routine.

Code in this second category is the type of code you should protect with two #ifdef or #ifndef statements. The outside directive should be a "#ifndef NDEBUG" statement that removes debugging code from the program for a normal (production) release. The second "#ifdef SomeSymbol" determines whether the compiler should process this particular block of debugging code. The symbol name should either (1) Be a general name that applies to several different blocks of debugging code or (2) be a specific name consisting of the function name followed by a description of what the code is logging.

Example 1: an example of some generic debugging code. The "EntryExit" symbol, if defined throughout the current source code module, prints a short message to the debug logging file on entry and exit from every function in this module.

```
int
MyFunction( int i )
{
    #ifndef NDEBUG
    #ifdef EntryExit

        fprintf( debugLogFile, "Entering MyFunction\n" );

    #endif
    #endif
        .
        .   << Statements that implement MyFunction >>
        .
    #ifndef NDEBUG
    #ifdef EntryExit

        fprintf( debugLogFile, "Exiting MyFunction\n" );

    #endif
    #endif

    return WhateverValueWasComputed;
}


int
YourFunction( int i )
{
    #ifndef NDEBUG
    #ifdef EntryExit

        fprintf( debugLogFile, "Entering YourFunction\n" );

    #endif
    #endif
        .
        .   << Statements that implement YourFunction >>
        .
    #ifndef NDEBUG
    #ifdef EntryExit

        fprintf( debugLogFile, "Exiting YourFunction\n" );
    #endif
    #endif

    return WhateverValueWasComputed;
}
```

By defining or undefining a single label ("EntryExit" in this case) you can quickly turn on or off the logging within a particular source module.

Often, you will want to explicitly turn on or off a single set of debugging statements. This is really no different than the situation above except that you can use a more specific name for the symbol that controls the code expansion. A good name generally involves the function name and the type of debugging action that occurs. The following code provides an example of such usage:

```
int
Recursive(int i)
{
    #ifndef NDEBUG
    #ifdef debugRecursive

        static int count=0;
        ++count;
        fprintf( debugLogFile, "Entry to recursive, count=%d i=%d\n",
               count, i );

    #endif
    #endif

    if ( ! --i ) Recursive( i );

    #ifndef NDEBUG
    #ifdef  debugRecursive

        --count;

    #endif
    #endif
}
```

You should define all the debug symbols for a source module at the very beginning of the source module. If you are not currently using a particular symbol, simply comment out its definition, do not remove the definition from the module. Every such symbol (commented out or present) should have some associated comments that describe the purpose of the symbol. Example:

```
<< Module header comments and include files >>

/*
```

```
** Debugging Symbols:
**
** EntryExit-  Defining this symbol causes many of the routines in this
**             module to log a brief message whenever you call them and
**             they return to the caller.
**
** debugRecursive- Displays the parameter and current recursive depth
**                 for a call to the "Recursive" function.
*/
/* #define EntryExit */
#define debugRecursive

        .
        .
        .
```

3.12 - Handling Error Return Values

Many standard library functions in C/C++ attempt to return two values in a single function result: the expected function result or an error indication. The "malloc" routine is a good example as is "getc". While this may seem to make an efficient use of a limited resource (i.e., the single function return result), combining two objects into a single return result is very bad programming style. It encourages programmers to ignore one or the other of the values since it is rarely obvious that a function is returning two values. For example, probably over 75% of the calls to "malloc" in typical C/C++ programs go unchecked for an out of memory condition.

Rule:

> **Functions should never attempt to return two (or more) values through a single parameter or function return result. If a function truly needs to return two different values, return them in separate locations (e.g., through pass by reference parameters).**

Although it is easy enough to control functions you write, a problem arises when calling standard library or third party library routines. The solution is to write wrapper functions that check input and output values for such library functions. By convention, many programmers have traditionally written special versions malloc, free, realloc, and other such calls to produce "safe" versions of these routines. This paper will suggest an extension of this feature and apply it to any library routine that can fail, not just the memory allocation routines.

For a given runtime library routine xyz(a, b, ...) you will substitute the call xyz_safe(&result, a, b, ...). The xyz_safe function returns a non-zero error code if an error occurs (typically the value one if there is only one type of error), it returns zero if the function was successful. This function returns the (non-error) function return result in the result parameter. A typical call to xyz_safe would look like this:

```
    _if ( xyz_safe(&result, a, b ) )

        fprintf(stderr,"function XYZ returned an error\n");
        fprintf(logfile,"error XYZ=%d\n" errno);
        exit(1);

    _endif;
    /* Down here "Result" contains whatever legal value xyz returned */
```

Another solution is to use the "_passert" macro to test the return result:

```
                 xyz_safe(&result, a, b) _passert("XYZ Failure");
```

Writing a wrapper function is very easy. Consider the "malloc" function. It normally returns NULL if a memory allocation error occurs, it returns a valid pointer into the heap if there is no error. The "malloc_safe" function takes the following form:

```
int malloc_safe(void **Result, size_t size)
{
    *Result = malloc(size);
    return Result == NULL;
}
```

This version of malloc is more portable, is easier to use (assuming you always test the malloc return result for an error), and is easier to read and understand (remember, functions should never return two different values in the same function return result; malloc_safe corrects this problem).

3.13 - Comments in a C/C++ Source File

Comments should take one of four basic forms in a C/C++ source file: module headers, program unit headers, multi-line comments, and single-line/endline comments. Module headers describe the contents of a source file. Program unit headers describe functions and global objects. Multi-line comments appear within a function or other program unit and describe the statements that immediately follow. Single-line and endline comments are single line comments appearing throughout the source file. Typically they describe a source statement or declaration appearing on the same line or immediately after the comment.

3.13.1 - Module Header Comments.

A module header is a block of comments at the beginning of a source file. Module headers should look like the following:

```
/*************************************************************************/
/*  Copyright(c) Information Management Associates, Inc. 1990            */
/*          All Rights Reserved                                          */
/*          An Unpublished Work                                          */
/*                                                                       */
/*  This is a Proprietary program product material and is the            */
/*  property of  INFORMATION  MANAGEMENT  ASSOCIATES, INC. No            */
/*  sale, reproduction or other  use of this program  product            */
/*  is authorized  except as  granted by the  fully  executed            */
/*  INFORMATION MANAGEMENT ASSOCIATES,  INC.  product license            */
/*  or by  the  separate  written  agreement  and approval of         */
/*                                                                    */
/*      INFORMATION   MANAGEMENT   ASSOCIATES,  INC.                  */
/*      6527 Main Street                                              */
/*      Trumbul, CT 94010                                             */
/*                                                                    */
/*  Author: XXXX YYYY                                                 */
/*  Revision History:                                                 */
/*    << date and explanation of each change >>                       */
/********************************************************************/
/*      str.c   Version 22.1
**      Checked in  1/24/96 at 12:39:03
*/      Retrieved   4/22/96 at 18:16:48
/********************************************************************/
/*  Purpose:                                                          */
/*  --------                                                          */
/*  This file contains the code to generate a string value            */
/*   containing a specified number of occurrences of a                 */
/*   specified string                                                 */
/********************************************************************/
```

3.13.2 - Function Header Comments

A function header should be a box of comments containing:

- Subroutine name and short description
- The purpose of the subroutine, its primary job, and any side-effects.
- A description of the parameters, their types and possible values, parameter passing mechanism, and whether they are in, out, or in/out parameters.
- A description of the return value (if any).
- Any assertions or other debugging tests available while in debugging mode.
- Data Dictionary- A description of the local variables, their types, and possible values.

Example:

```
/******************************************************************/
/*                                                                */
/* Name: Match                                                    */
/*                                                                */
/* Purpose:                                                       */
/* This function matches an input string against a specified pattern. */
/* It returns true if the pattern can match the string, it returns    */
/* false if this is not possible.                                 */
/*                                                                */
/* Parameters:                                                    */
/*                                                                */
/* char *StringToMatch; (input-only).                             */
/* This is the input string to match.                                     */
/*                                                                        */
/* Pat *Pattern; (input-only).                                            */
/* This is the pattern to apply to the string.                            */
/*                                                                        */
/* Returns:                                                               */
/* True (1) if the pattern can match this string, false otherwise.        */
/*                                                                        */
/* Assertions:                                                            */
/* Neither StringToMatch nor Pat are NULL.                                */
/* StringToMatch points at a zero terminated character string,            */
/* Pattern points at a valid "Pat" data structure.                        */
/*                                                                        */
/* Variables:                                                             */
/* int Index -  This is the current index into the string.                */
/* Pat *SubPat- This is a pointer to the current subpattern               */
/*              we are matching.                                          */
/*                                                                        */
/* Written by:                                                            */
/* Randall Hyde 3/30/95  Version 1.0                                      */
/* Randall Hyde 7/02/95  Improved performance of recursive                */
/*                       matching function.                               */
/*                                                                        */
/**************************************************************************/
```

3.13.3 - Multi-line Comments

Multi-line comments are several lines of comments appearing in the middle of a function that describes the statements immediately following. These comments should be less obtrusive than function and module headers. Hence, you shouldn't enclose them in a box of asterisks. To avoid destroying the indentation of the program, you should indent all comments in a function as though they were a statement. Multi-line comments should immediately precede the code to which they apply. There should be at least one blank line between the comment and the code it describes. There should be at least two blank lines between the preceding code and the start of the comment. Physically, multi-line comments should look like the following:

```
/*
** If the string has matched the current subpattern, try
** matching the string against the "next" subpattern.
**
** On the other hand, if the string did not match the
** current subpattern, try matching the string against
** the "alternate" subpattern.
**
** Return true if the current subpattern matches
** and "next" is NULL or the "next" subpattern also matches.
**
** Return true if the current subpattern fails to match but
** the "alternate" subpattern matches.
**
** Return false if neither the current subpattern nor the
** "alternate" subpattern matches.
*/
```

Alternately, C++ programmers can use comments like the following:

```
//
// If the string has matched the current subpattern, try
// matching the string against the "next" subpattern.
//
// On the other hand, if the string did not match the
// current subpattern, try matching the string against
// the "alternate" subpattern.
//
// Return true if the current subpattern matches
// and "next" is NULL or the "next" subpattern also matches.
//
// Return true if the current subpattern fails to match but
// the "alternate" subpattern matches.
//
// Return false if neither the current subpattern nor the
// "alternate" subpattern matches.
//
```

3.13.4 - Single Line / Endline Comments

Single line comments are exactly that- a short description that applies to a statement or a declaration. Single line comments sit on a single line by themselves immediately before the statement to which they apply (although a blank line always goes between the comment and the following code). Endline comments appear on the same line as the statement to which they apply. These comments generally look like one of the following:

```


C Programs:

    /* After sorting, the median element is at index n/2 */

    sort( data, NumItems );

        .
        .
        .

    Median = data[ NumItems / 2 ];  /* Data *must* be sorted! */


C++ Programs:
    // After sorting, the median element is at index n/2

    sort( data, NumItems );

        .
        .
        .

    Median = data[ NumItems / 2 ];  // Data *must* be sorted!
```

3.14 - C++ Specific Features and Guidelines

(This section still needs to be written).

---

[1] For C++, extensions like ".cc", ".cpp", and ".cxx" are common as well. [back]

< Previous section *(2 - General Programming Guidelines)* - **Contents** - **Next section** *(4 - Pascal/Delphi Specific Formatting Issues) >*

Software Development Guidelines

4 - Pascal/Delphi Specific Formatting Issues

< Previous section *(3 - C (and related C++) Specific Issues)* - **Contents** - **Next section** *(5 - Visual BASIC Specific Formatting Issues) >*

4 - Pascal/Delphi Specific Formatting Issues

This section describes style guidelines for Borland's Turbo Pascal and Delphi products. Many of the guidelines appearing in this section are generic enough to apply to any Pascal source file, however, no attempt is made to differentiate Borland Pascal/Delphi from generic Pascal since 90% of today's Pascal programming is done with a Borland product. A special section near the end of this section discusses Delphi-specific issues.

4.1 - Control Constructs in Pascal

Like C/C++, Pascal does not provide the modern if..elseif..else.endif, while..endwhile, etc., control structures (except, of course, for repeat..until). Unlike C/C++, Pascal does not provide a macro mechanism that lets us create our own versions of these statements. Therefore, we will simulate these statements rather than create them outright.

To emulate the modern control structures, we will always associate a begin..end block with each Pascal statement (except repeat..until). The proper indentation style is to place the "begin" on the same line as the statement and line up the corresponding "end" in the same column as the statement. The following paragraphs provide the rules for the if..elseif..else..endif, while..endwhile, for..endfor, and loop..endloop statements.

To create an if..endif statement, you simply attach a "begin" and "end{if}" sequence to the standard Pascal if statement. Note the use of the Pascal comment to denote the end of an if sequence. You should indent the statement(s) to execute four character positions within the if..end block:

```
if ( expression ) then begin

    << Statement(s) to execute >>

end{if};
```

An if..then..else statement is similar. Note that the "{if}" comment only follows the last "end" in the if statement:

```
if ( expression ) then begin

    << Statement(s) to execute if "expression" is true >>

end
else begin

    << Statement(s) to execute if "expression" is false >>

end{if};
```

An if..then.elseif statement takes the following form:

```
if ( expression ) then begin

    << Statement(s) to execute if "expression" is true >>

end
else if ( expression2 ) then begin

    << Statement(s) to execute if "expression2" is true >>

end{if};
```

Finally, here's an example of an if..then..elseif..else..endif statement:

```
if ( expression ) then begin

    << Statement(s) to execute if "expression" is true >>

end
else if ( expression2 ) then begin

    << Statement(s) to execute if "expression2" is true >>

end
else begin

    << Statement(s) to execute if both expressions are false >>
```

```
end{if};
```

Of course it is perfectly reasonable to insert as many "else if" sections as you require into the sequence above.

You should treat the "end/else if" and "end/else begin" sequences as a single clause that should be kept together. In particular, you should avoid separating these clauses by a large number of comments or blank lines. Doing so would give the impression that the previous clause has ended when this is not the case. You should place all comments that apply to the following block within that block (and indent those comments four spaces to line up with the other statements in the block).

To implement a while..endwhile statement, you need only attach a begin..end block to the existing while statement. As for the if statement, a "{while}" comment should immediately follow the closing "end" clause to mark the end of the statement. You should indent the statements inside the while..endwhile block two character positions. The following example demonstrates this statement:

```
while( expression ) do begin

    << Statement(s) to repeat while "expression" is true >>

end{while};
```

Like the while statement, we can emulate a modern for..endfor loop in Pascal by simply attaching a begin..end block to the Pascal "for" statement. As usual, we will follow the "end" clause with a "{for}" comment. You should indent statements within the emulated for..endfor statement four character positions. The following example demonstrates how to do this:

```
for LoopVar := Start to Stop do begin

    << Statement(s) to execute specified number of times >>

end{for};
```

Pascal does not provide a built-in mechanism for creating infinite loops. The convention among Pascal programmers is to use a "while(true) do" loop and rely upon the compiler to optimize away the expression. This standard adopts that convention with a minor twist -- rather than just use the constant "true", you will define a constant "loop" whose value is true and use that constant, without surrounding parenthesis on a separate line. This gives the illusion of having a loop statement:

```
while
loop do begin

    << Statement(s) to execute forever. >>

end{loop};
```

Note the use of the "{loop}" rather than "{while}" to mark the end of this statement. Of course, for this trick to work properly, you must have the following statement appearing in a visible (preferably global) "const" section of your Pascal program:

```
loop = true;
```

Borland's Pascal compilers provide "break" and "continue" statements (Borland calls these "procedures" for some reason). You may use the "break" statement to exit an infinite loop. For the purposes of testing for loop termination in the middle of a loop..endloop statement, you can use a Pascal if statement like the following:

```
{break}if( expression ) then break;
```

Note that for this special case you do not have to attach a begin..end statement since we're really simulating the breakif statement here. If necessary, you may use this same structure with the continue statement (although it is rarely necessary).

Unfortunately, Pascal does not provide an easy way to simulate the _context.._endcontext structure. The only plausible way is to use the GOTO statement. An appropriate way to do this is to use a comment to mark the beginning and end of the context as follows:

```
function SomeFunc:integer;
label SF_cntx;
begin

    {context SF_cntx}
            .
            .
            .
        if( expression ) then goto {leave} SF_cntx;
            .
            .
            .
    {end context} SF_cntx:
```

Note that when simulating an _leave, like when simulating a _breakif, you do not need to use the begin..end block. Also note the use of the comment between the GOTO and the corresponding label. Of course, Standard Pascal requires numeric rather than symbol statement labels; fortunately, Borland's Pascal and Delphi compilers allow the use of symbolic labels.

Unfortunately, there is no easy way to simulate the "_return" statement with a single statement in Pascal. The only option is to use two statements. The first should assign a value to the "Result" variable (predeclared for you in Delphi!) or to the function return name and then jump to the end of the function.

Given the unstructured nature of the context..endcontext emulation in Pascal, you should avoid its use whenever possible. For those using the Delphi language, the try..finally and try..except blocks provide a nicer and more readable alternative (although you should be careful about using exception handling statements in normal flow-of-control situations). Keep in mind that this context..endcontext emulation still uses GOTO statements and, therefore, the guideline concerning the use of GOTOs within a program applies.

The Pascal repeat..until statement already takes the form for a modern control structure. Therefore, there is no need to mess with it's syntax.

4.2 - Semicolons in a Pascal Program

Pascal uses semicolons to separate, rather than terminate, statements. Therefore, semicolons are not necessary after statements that immediately appear before certain clauses like "end" or "until" in a Pascal program. Of course, it is syntactically legal to insert a semicolon after these statements because Pascal allows the null statement (the empty string) to appear anywhere a statement is legal. Therefore, semicolons immediately before one of these ending clauses are optional. Experienced programmers know that it is a good idea to always put the optional semicolons into a program; doing so makes the program easier to modify since one doesn't have to remember to put the semicolon in when adding new statements to the program.

Rule:

> **Use a semicolon in a Pascal program wherever it is optional.**

4.3 - Modules in Pascal/Delphi

Borland's Pascal and Delphi compilers use the defacto standard for Pascal modules used by UCSD Pascal- units. Pascal units support the concept of information hiding via a public interface section and a private implementation section. Pascal units provide a powerful facility for creating maintainable programs.

Function and variable names declared outside of any function or procedure can be either private or public. Those identifiers appears between the interface reserved word and the implementation reserved word are public. Those appearing after the implementation reserved word are private to the module. The scope of private objects is limited to the current source file; that is, you cannot directly reference a private object from another module. Note, however, that all functions and procedures within the current module may access private objects. Public objects, on the other hand, are available (via the linker) to other modules.

Rule:

> **All private objects (that is, variable, function, and procedure names that should remain local to a given source file) must appear in the implementation section in a Pascal/Delphi source file. Variables appearing in the interface section are public objects that other modules can use.**

Borland Pascal and Delphi require function headers/definitions in the interface section for all functions the module exports. This document will hereafter refer to such definitions as prototypes. A prototype provides a mechanism whereby the compiler can perform stronger type checking on functions that a module calls before their declaration. A second type of prototype is the Pascal forward declaration. One would typically use a forward declaration when mutual recursion occurs or when you want to physically reorder procedures and functions within a source file for purposes of readability rather than calling sequence. However, many Pascal programmers have discovered that placing a set of forward declarations at the beginning of a source module (in the implementation section, for private functions) makes it easier to work on the programs (since the function/procedure prototypes appear all in one spot).

Guideline:

> **All functions (public or private) appearing in a source module will have an associated prototype or forward declaration. Public prototypes must appear at the appropriate point in the interface section (i.e., after the const, type, and var sections), private prototypes and forward declarations must appear at the appropriate spot in the implementation section (i.e., after const, type, and var sections, but before the first real program unit bodies).**

Guideline:

> **The order of the Pascal prototypes and forward declarations should match the order of the functions/procedures appearing in the source module.**

Rule:

> **Pascal functions and procedures that are also public should appear near the beginning of the source file.**

4.4 - Coding for Testability in Delphi/Pascal

Borland's Pascal compilers provide a conditional compilation facility that is similar to C/C++'s conditional compilation. In addition, Borland's Delphi language provides exception handling via try..except and try..finally. Unlike C/C++, Borland's Pascal and Delphi compilers will automatically emit code to perform range checking, out of memory checking, and other consistency checks. Together, these three techniques provide facilities for inserting code into a program to help test the correct operation of that code.

Enforced Rule:

> **By default, Borland's Pascal and Delphi compilers have most of the optional run-time checks disabled. During software development you should enable all these checks. Turn them off when shipping production code.**

4.5 - Conditional Compilation in Delphi/Pascal

The Pascal/Delphi conditional compilation directives are very similar to those found in C/C++. By convention, Borland's engineers embedded these directives into comments that begin with the lexeme "(*$ directive *)" or "{$ directive }". Since the "{$ directive }" is less obtrusive, it makes a better choice for containing the conditional compilation directives.

Guideline:

> **You should use the "{$ " and "}" symbols to surround a Delphi/Pascal compiler directive since the result is easier to read than the same directive surrounded by "(*$" and "*)".**

The following is a list of the more useful conditional compilation directives:

- {$define symbol }
- {$ifdef symbol }
- {$ifndef symbol }
- {$else}
- {$endif}

You can easily surround a section of test code using the {$ifdef} (or {$ifndef}) directive and activate or deactivate this code by defining or undefining the appropriate symbol. The NDEBUG symbol is special. If defined, your program must not compile any test/debugging code. This allows whoever builds a production copy of the Pascal code to feel they have confidently eliminated all debugging code by defining a single symbol during compilation.

Rule:

> **All Delphi/Pascal debugging code must disappear if the symbol "NDEBUG" is defined. That is, you must surround all debugging code with "{$ifndef NDEBUG }" and a corresponding "{$endif}".**

Although you may eliminate all test/debugging statements using a single definition (NDEBUG), you may not want to have all debugging statements active at one time (i.e., if you have not defined NDEBUG). For example, if a particular module has five sets of debugging statements in it, you may only be interested in one set or the other. With the NDEBUG symbol, it's all or nothing; either all the debug statements are present or none of them are present. The solution is to associate a name with the current debug block (or a set of debug blocks) and use a nested {$ifdef} directive to activate or deactivate the debugging code. Consider the following code:

```
{$ifndef NDEBUG    To turn off debugging for production code. }
{$ifdef DebugCalc}

    writeln(debugLogFile,'Appropriate Logging Output');
{$endif}
{$endif}
```

Normally, you should indent {$ifdef} and {$ifndef} statements just like any statement in Delphi/Pascal. That would suggest that the style above is incorrect. However, since these statements will always occur in pairs, an exception is quite justifiable in this particular case.

Debugging statements programmers put into their programs generally fall into three categories:

- One-time blocks of code that help you track down an eliminate a single bug (and, obviously, never appear in production code).
- Debugging statements that log current values in a module for later perusal to see if a section of code is getting and computing reasonable values. This code never appears in the production version of the program.
- Debugging statements that log values or provide information on demand in a production version of a program. A field engineer, for example, might use such code to test the program at a customer's site.

The first category of statements above (one-time blocks) normally shouldn't exist - you should really be using a debugger to track down these types of problems. However, not all debuggers are flexible enough to handle every situation and a small block of debugging code can save the day. These debugging statements are the most dangerous to insert into a program. Often, the programmer sees these statements as temporary additions to the code that s/he will remove momentarily. Unfortunately, such statements invariably wind up in production code because the programmer, in haste, forgot to remove the debugging statements and failed to protect them with a {$ifndef NDEBUG} directive.

Enforced Rule:

> **All "temporary" debugging statements you add to a Pascal/Delphi program, no matter how temporary they seem, must be protected with "{$ifndef NDEBUG}" and "{$endif}" directives. This is the only line of defense against forgetting to remove the code before compiling a production version of the program.**

The second category of debug statements above will typically print parameters passed into a function, print computations during the execution of a function, print return results from a called function, and print results that the current function returns. Programmers often insert such statements into their programs as temporary measures to track down an elusive defect in the software. However, such statements should be coded in as permanent fixtures in a program. After all, once you've tracked down a current problem with a routine, there is no guarantee that there won't be future problems with that routine. If you leave the logging code in place, you will not have to rewrite it the next time you want to trace a routine.

Code in this second category is the type of code you should protect with two {$ifdef} or {$ifndef} directives. The outside directive should be a "{$ifndef NDEBUG}" statement that removes debugging code from the program for a normal (production) release. The second "{$ifdef SomeSymbol}" directive determines whether the compiler should process this particular block of debugging code. The symbol name should either (1) Be a general name that applies to several different blocks of debugging code or (2) be a specific name consisting of the function name followed by a description of what the code is logging.

Example 1: an example of some generic debugging code. The "EntryExit" symbol, if defined throughout the current source code module, prints a short message to the debug logging file on entry and exit from every function in this module.

```
function MyFunction( i:integer ):integer;
begin

    {$ifndef NDEBUG}
    {$ifdef EntryExit}
```

```
        WriteLn( debugLogFile, 'Entering MyFunction' );

    {$endif}
    {$endif}
        .
        .   << Statements that implement MyFunction >>
        .
    {$ifndef NDEBUG}
    {$ifdef EntryExit}

        WriteLn( debugLogFile, 'Exiting MyFunction' );

    {$endif}
    {$endif}

end{MyFunction};


function YourFunction( i:integer ):integer;
begin

    {$ifndef NDEBUG}
    {$ifdef EntryExit}

        WriteLn( debugLogFile, 'Entering YourFunction' );

    {$endif}
    {$endif}
        .
        .   << Statements that implement YourFunction >>
        .
    {$ifndef NDEBUG}
    {$ifdef EntryExit}

        WriteLn( debugLogFile, 'Exiting YourFunction' );

    {$endif}
    {$endif}

end{YourFunction};
```

By defining or undefining a single label ("EntryExit" in this case) you can quickly turn on or off the logging within a particular source module.

Often, you will want to explicitly turn on or off a single set of debugging statements. This is really no different than the situation above except that you can use a more specific name for the symbol that controls the code expansion. A good name generally involves the function name and the type of debugging action that occurs. The following code provides an example of such usage:

```
(*
** Note: Presumably, "count" integer variable initialized to
**       zero by some module initialization code.
*)

procedure Recursive( i:integer );
begin

    {$ifndef NDEBUG}
    {$ifdef debugRecursive}

        inc( count );
        WriteLn( debugLogFile, 'Entry to recursive, count=', count, ' I=', i );

    {$endif}
    {$endif}

    dec( i );
    if ( i <> 0 ) then begin

        Recursive( i );

    end{if};

    {$ifndef NDEBUG}
    {$ifdef  debugRecursive }

        dec( count );

    {$endif}
    {$endif}

end{Recursive};
```

You should define all the debug symbols for a source module at the very beginning of the source module. If you are not currently using a particular symbol, simply replace the "$" in the directive with another symbol like "*", do not remove the definition from the module. Every such symbol (commented out or present) should have some associated comments that describe the purpose of the symbol. Example:

```
<< Module header comments and include files >>

(*
** Debugging Symbols:
**
** EntryExit-  Defining this symbol causes many of the routines in this
**             module to log a brief message whenever you call them and
**             they return to the caller.
**
** debugRecursive- Displays the parameter and current recursive depth
**                 for a call to the "Recursive" function.
*)
```

```
{*define EntryExit }   (* This symbol is commented out! *)
{$define debugRecursive}


        .
        .
        .
```

4.6 - Handling Error Return Values

A few Pascal functions may need to return an error status as well as a function return result. Some programmers have developed a habit of returning an illegal value to denote an error, an otherwise legal value denotes success. Unfortunately, this is a rather bad programming practice, especially in Delphi (that provides exception handling facilities). For Borland Pascal users, where exception handling statements are not available, you should return the error status (a boolean value) as the function result and return any other function "result" via a reference parameter.

Rule:

> **Functions in Delphi code should never attempt to return two (or more) values through a single parameter or function return result. If a function truly needs to return two different values, return them in separate locations (e.g., through pass by reference parameters). If one of the return values denotes an error condition, use Delphi's exception handling facilities to raise an exception.**

4.7 - Comments in a Delphi/Pascal Source File

Comments should take one of four basic forms in a Delphi/Pascal source file: module headers, program unit headers, multi-line comments, and single-line/endline comments. Module headers describe the contents of a source file. Program unit headers describe functions and global objects. Multi-line comments appear within a function or other program unit and describe the statements that immediately follow. Single-line and endline comments are single line comments appearing throughout the source file. Typically they describe a source statement or declaration appearing on the same line or immediately after the comment.

4.7.1 - Module Header Comments.

A module header is a block of comments at the beginning of a source file. Module headers should look like the following:

```
(*************************************************************************)
(*  Copyright(c) Information Management Associates, Inc. 1990           *)
(*          All Rights Reserved                                          *)
(*          An Unpublished Work                                          *)
(*                                                                       *)
(*  This is a Proprietary program product material and is the           *)
(*  property of  INFORMATION  MANAGEMENT  ASSOCIATES, INC. No           *)
(*  sale, reproduction or other  use of this program  product           *)
(*  is authorized  except as  granted by the  fully  executed           *)
(*  INFORMATION MANAGEMENT ASSOCIATES,  INC.  product license           *)
(*  or by  the  separate  written  agreement  and approval of           *)
(*                                                                       *)
(*      INFORMATION   MANAGEMENT   ASSOCIATES,  INC.                      *)
(*      6527 Main Street                                                 *)
(*      Trumbul, CT 94010                                                *)
(*                                                                       *)
(*  Author: XXXX YYYY                                                    *)
(*  Revision History:                                                    *)
(*    << date and explanation of each change >>                          *)
(************************************************************************)
(*      str.pas   Version 22.1
**      Checked in  1/24/96 at 12:39:03
*)      Retrieved   4/22/96 at 18:16:48
(************************************************************************)
(*  Purpose:                                                             *)
(*  --------                                                             *)
(*  This file contains the code to generate a string value              *)
(*   containing a specified number of occurrences of a                    *)
(*   specified string                                                    *)
(************************************************************************)
```

4.7.2 - Function Header Comments

A function header should be a box of comments (a "flowerbox") containing:

- Subroutine name and short description
- The purpose of the subroutine, its primary job, and any side-effects.
- A description of the parameters, their types and possible values, parameter passing mechanism, and whether they are in, out, or in/out parameters.
- A description of the return value (if any).
- Any assertions or other debugging tests available while in debugging mode.
- Data Dictionary- A description of the local variables, their types, and possible values.

Example:

```
(**********************************************************************)
(*                                                                     *)
(* Name: Match                                                         *)
(*                                                                     *)
(* Purpose:                                                            *)
(* This function matches an input string against a specified pattern. *)
(* It returns true if the pattern can match the string, it returns    *)
(* false if this is not possible.                                      *)
(*                                                                     *)
(* Parameters:                                                         *)
(*                                                                     *)
(* StringToMatch:PChar (input-only)                                    *)
(* This is the input string to match.                                  *)
(*                                                                     *)
(* Pattern: Pat; (input-only).                                         *)
(* This is the pattern to apply to the string.                         *)
(*                                                                     *)
(* Returns:                                                            *)
(* True if the pattern can match this string, false otherwise.        *)
(*                                                                    *)
(* Assertions:                                                        *)
(* Neither StringToMatch nor Pat are NIL.                             *)
(* StringToMatch points at a zero terminated character string,        *)
(* Pattern points at a valid "Pat" data structure.                    *)
(*                                                                    *)
(* Variables:                                                         *)
(* Index:integer -  This is the current index into the string.        *)
(* SubPat:Pat - This is a pointer to the current subpattern           *)
(*              we are matching.                                      *)
(*                                                                    *)
(* Written by:                                                        *)
(* Randall Hyde 3/30/95  Version 1.0                                  *)
(* Randall Hyde 7/02/95  Improved performance of recursive            *)
(*                       matching function.                           *)
(*                                                                    *)
(**********************************************************************)
```

4.7.3 - Multi-line Comments

Multi-line comments are several lines of comments appearing in the middle of a function that describes the statements immediately following. These comments should be less obtrusive than function and module headers. Hence, you shouldn't enclose them in a box of asterisks. To avoid destroying the indentation of the program, you should indent all comments in a function as though they were a statement. Multi-line comments should immediately precede the code to which they apply. There should be at least one blank line between the comment and the code it describes. There should be at least two blank lines between the preceding code and the start of the comment. Physically, multi-line comments should look like the following:

```
(*
** If the string has matched the current subpattern, try
** matching the string against the "next" subpattern.
**
** On the other hand, if the string did not match the
** current subpattern, try matching the string against
** the "alternate" subpattern.
**
** Return true if the current subpattern matches
** and "next" is NIL or the "next" subpattern also matches.
**
** Return true if the current subpattern fails to match but
** the "alternate" subpattern matches.
**
** Return false if neither the current subpattern nor the
** "alternate" subpattern matches.
*)
```

4.7.4 - Single Line / Endline Comments

Single line comments are exactly that- a short description that applies to a statement or a declaration. Single line comments sit on a single line by themselves immediately before the statement to which they apply (although a blank line always goes between the comment and the following code). Endline comments appear on the same line as the statement to which they apply. These comments generally look like one of the following:

```
Pascal/Delphi Programs:

    (* After sorting, the median element is at index n/2 *)

    sort( data, NumItems );

        .
        .
        .

    Median = data[ NumItems / 2 ];  { Data *must* be sorted! }
```

Note the use of the "(*" and "*)" delimiters for single line comments and the "{" and "}" delimiters for endline comments. The "{" and "}" delimiters are less obtrusive and, therefore, are better candidates for endline comments since they produce less of a distraction. Single line comments, on the other hand, should stand out more; hence the use of the "(*" and "*)" symbols for single line comments.

Guideline:

> **Use the "{" and "}" delimiters for endline comments (comments appearing at the end of a line that contains some other statement). Use the "(*" and "*)" delimiters for single line comments (a single comment appearing on a line by itself).**

4.8 - Delphi Specific Issues

The layout of a typical Delphi program is often somewhat arbitrary because Delphi appends event handling procedures to the end of a source code module as the programmer adds objects to a form. Since the order that a programmer writes event handlers is sufficiently random, it is often difficult to find corresponding routines in a Delphi source listing. Most Delphi programmers depend on the code browser to help them locate the code associated with a particular item on a form. Alas, when one does have to consult the source file, the result is less than acceptable. Fortunately, Delphi does not prevent you from rearranging procedures and functions within a source module. Therefore, you can organize the source code rationally rather than randomly.

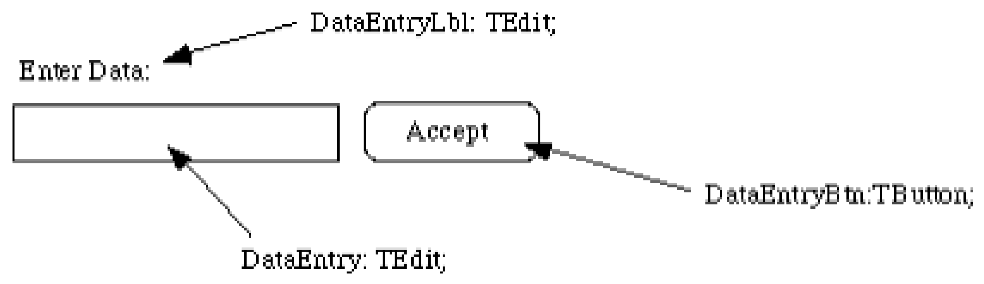
One good source file organization for a Delphi unit is to place all the procedures and functions (that are not associated with some class) at the beginning of an implementation section. Following the stand-alone procedures and functions in a unit come the methods (procedures and functions) associated with each class appearing in the unit. The methods associated with a single class should appear adjacent to one another in the source file. Furthermore, these groups of methods should appear in the same order as the class definitions in the source file except for the class associated with a form. The methods for items associated with the form should appear last in the source file (this is done because Delphi automatically appends new event handlers for components on the form to the end of the source file).

Guideline:

> **You should organize Delphi unit source files with the procedures and function that are not attached to a particular class at the beginning of the implementation section. Methods associated with local class**

objects should follow these procedures and functions. Finally, methods associated with event handlers for components on a form should appear at the end of the source file.

A minor problem with naming occurs in Delphi programs that place components on forms. Each component must appear as a direct field within the form's class definition[1]. Often there are several related components on a form:



Ideally one would like to encapsulate these related items into a class or record data structure and refer to them by names like DataEntry.edit, DataEntry.Btn, and DataEntry.Lbl. Unfortunately, Delphi does not let you (easily) do this. However, we can simulate this, to a degree, by simply naming the objects as though they were record or class objects and substitute an underscore for the period. This produces names like "DataEntry_Btn" and "DataEntry_Lbl". The following table suggests different suffixes for the common components found in Delphi:

Delphi Type Suffixes

Component Type Name	**Typical Suffix**
TMainMenu	menu
TPopupMenu	popup
TLabel	lbl
TEdit	(no suffix)
TMemo	memo
TButton	btn
TCheckBox	chk
TRadioButton	rbtn
TListBox	list
TComboBox	cmbo
TScrollBar	scrl
TGroupBox	box
TRadioGroup	grp
TPanel	pan
TBitBtn	btn
TSpeedBtn	btn
TMaskEdit	msk
TStringGrid	grid
TDrawGrid	dgrid
TShape	shp
TBevel	bvl
TScrollBox	scrl
TTabControl	tab
TPageControl	pag

TTreeView	tree
TListView	view
TImageList	ilst
TRichEdit	rtf
TStatusBar	sts
TImage	img

This is but a partial list of suggestions. Note that not all the names are unique; those that share the same suffix would rarely appear together. For those components not appearing in the table above, choose an abbreviation in the style of these suffixes and use it consistently.

Guideline:

> **When naming Delphi components you place on a form, append a suffix string that consists of an underscore followed by a standard string that denotes the type of that component (e.g., "_lbl" for TLabel objects).**

Guideline:

> **If two or more components are related (e.g., a TLabel object that describes the type of input for a TEdit object) then use the same name with different suffixes (e.g., Month and Month_lbl). Note that this is an exception to the rule that identifiers should not have more than a few characters in common as their prefix characters; it also violates the rule that two identifiers should not be the same except for a type suffix. This exception exists to help overcome the limitation that a component cannot be a field of a record or class except a TForm object.**

---

[1] Actually, this isn't quite true. If you create the component dynamically this restriction does not apply. However, creating all your component dynamically is a lot of work and doing so destroys one of the major reasons for using Delphi in the first place. [back]

< Previous section *(3 - C (and related C++) Specific Issues)* - **Contents** - **Next section** *(5 - Visual BASIC Specific Formatting Issues)* **>**

Software Development Guidelines

5 - Visual BASIC Specific Formatting Issues

< **Previous section** *(4 - Pascal/Delphi Specific Formatting Issues)* - **Contents** - **Next section** *(6 - Lex/Flex and Yacc/Bison Specific Formatting Issues)* >

5 - Visual BASIC Specific Formatting Issues

< **Previous section** *(4 - Pascal/Delphi Specific Formatting Issues)* - **Contents** - **Next section** *(6 - Lex/Flex and Yacc/Bison Specific Formatting Issues)* >

Software Development Guidelines

6 - Lex/Flex and Yacc/Bison Specific Formatting Issues

< Previous section *(5 - Visual BASIC Specific Formatting Issues)* - **Contents** - **Next section** *(7 - Other Languages (Formatting Issues))* >

6 - Lex/Flex and Yacc/Bison Specific Formatting Issues

The Lex/Flex and Yacc/Bison languages process regular expressions and context free grammars (respectively). Whenever an input string matches a pattern specified in a regular expression or context free grammar, the program executes a corresponding "semantic rule" (some C/C++ code). Clearly, the rules specified in this document for C/C++ code applies to those statements. This section will deal with the explicit Lex/Flex and Yacc/Bison code and ignore the C/C++ code that is also a part of a typical Lex/Flex or Yacc/Bison program.

Lex/Flex and Yacc/Bison programs have a very similar structure. The structure of programs in both languages is

```
        ...Definition Section...
%%
        ...Rules Section...
%%
        ...Auxiliary Code/User Section...
```

The definition section contains language specific statements and C/C++ code. There are two ways to insert C/C++ code into the definition section: (1) if a line begins with a whitespace character, the Lex/Flex or Yacc/Bison processor copies the entire line directly to the output C/C++ file; (2) the Lex/Flex and Yacc/Bison translators copy all lines of text between a line beginning with "%{" and a line beginning with "%}" to the C/C++ output file. C/C++ statements appearing outside the "%{" and "%}" lines are easy to confuse with other Lex/Flex or Yacc/Bison definitions, therefore

Rule:

> **All C/C++ statements appearing in the definitions section of a Lex/Flex or Yacc/Bison program should appear at the beginning of the sections bracketed by a pair of lines containing the "%{" and "%}" tokens.**

The translator copies C/C++ code from the definition section to the output C/C++ source file. Technically, you can place any C/C++ code you desire into this portion of the program. By convention, however, most programmers only place global type, constant, and variable declarations and definitions or function prototypes into this portion of the program. You should only define those C/C++ objects that the rules section uses in this section of the source file; you should declare and define all other C/C++ objects in the auxiliary code section.

Rule:

> **The definition section of a Flex/Lex or Yacc/Bison program should contain only those C/C++ definitions and declarations necessary for the rules section of the program. You should place all other C/C++ code in the auxiliary code section of the program.**

The rules section of a Lex/Flex or Yacc/Bison program contains the "real" program. Since these two sections vary considerably between the two languages, we will consider the formatting issues related to the rules section in later sections of this document.

The Lex/Flex and Yacc/Bison translators simply copy all statements appearing in the auxiliary code section to the output C/C++ program. Since this section must contain C/C++ code, see the section in this document covering C/C++ specific issues for the correct style.

6.1 - Lex/Flex Specific Issues

This section documents style and formatting issues specific to the Lex/Flex (Flex, hereafter) language. Keep in mind that a large portion of a typical Flex program is C/C++ code and most of the C/C++ specific guidelines apply to that code.

6.1.1 - The Flex Definitions Section

The definitions section in a Flex program typically contains four types of items: comments, C/C++ code, definitions (substitutions), and starting symbol definitions.

Comments are really just a special case of C/C++ code; for portability reasons, any line containing a comment should contain whitespace at the beginning of a line or appear within a "%{" .. "%}" block. Generally, comments appearing within a "%}" .. "%}" block should document the C/C++ code associated with that block, comments outside the block should apply to the Flex specific code they immediately proceed[1].

For some strange reason, programmers who normally do an excellent job of commenting their C/C++ programs will often write Flex programs with few, or no, comments. Although Flex is a very high level language, programs written in this language still need a large number of comments. This is especially true when defining complex regular expressions. Don't assume the reader can easily make sense of a Flex program just because it's short.

Rule:

> **The Flex and C/C++ statements in a Flex program must contain a liberal number of comments describing the code. Do not assume that the regular expression is "self documenting."**

As discussed earlier, the only C/C++ code that should appear in a Flex definitions section should be global definitions, function prototypes, and include statements that define objects used by the C/C++ code appearing in the rules section of the program. You should place all other C/C++ code in the auxiliary section.

Definitions are, essentially, simple macros that let you substitute meaningful names for regular expressions. In a sense, Flex definitions are like "consts" in a C/C++ program. Unfortunately, the widespread use of definitions in a Flex program is not common. Programmers who would never dream of embedding "magic" literal values into the middle of a C/C++ program don't even think twice about using pure regular expressions throughout a Flex program, even if that regular expression appears several times. If a regular expression describes some pattern whose purpose you can easily articulate, you should create a definition for it (examples: identifier, intconst, floatconst, etc.).

Guideline:

> **Use Flex definitions to give meaningful names to common or oft-used regular expressions within a program. Once you create a definition, be sure to use the definition's name as appropriate throughout the Flex program.**

Rule:

> **Always precede each Flex definition with a comment that describes the pattern matched by the corresponding regular expression. Also provide several example strings that the regular expression matches. Don't assume that the definition's name completely documents the corresponding regular expression.**

A Flex definitions section may also contain a set of start conditions. These statements in a Flex program generally begin with a "%s" or "%x" in column one followed by a list of C/C++ identifiers. These symbols identify blocks of code that Flex processes in a context-sensitive fashion (see the Flex documentation for more details). These identifiers, like all identifiers in a program, should be meaningful and describe the context in which the program uses the corresponding rules. For example, the following Flex code processes "C" style comments in a program:

```
%x ProcessComments
%%
```

```
      .
      .
      .
"/*"  {

        BEGIN ProcessComments;

    }
      .
      .
      .
<ProcessComments>.    {}
<ProcessComments>"*/" {

        BEGIN 0;

    }
```

Rule:

> **The identifiers you specify for Flex "starting conditions" should be meaningful; they should describe the context handled by the starting condition.**

Traditionally, (as exemplified above), Flex programmers have used the "BEGIN 0;" statement to revert back to the standard set of regular expressions appearing in the rules section. A better solution is to create a C/C++ constant using the "#define" directive to give this a more meaningful name, e.g.,

```
%{
#define StandardREs
}

%x ProcessComments
%%
      .
      .
      .
"/*"  {

        BEGIN ProcessComments;

    }
      .
      .
      .
<ProcessComments>.    {}
<ProcessComments>"*/" {

        BEGIN StandardREs;
    }
```

Guideline:

> **Define the macro "StandardREs" to be replaced by the literal constant "0". Then use "BEGIN StandardREs;" rather than "BEGIN 0;" to turn off context sensitivity in a Flex program.**

Each starting condition statement ("%s" or "%x") should define the symbols associated with a common context. The symbols for different contexts belong on separate lines. For example, you should put symbols like "ProcessComments" and "EndComments" together in the same statement if these symbols describe contexts used to process comments; however, something like "StringConstant" really belongs in a separate statement since it has nothing to do with processing comments:

```
%x ProcessComments  EndComments
%x StringConstant
```

The use of starting conditions to handle context sensitivity within a Flex program indicates a fair amount of complexity within a program. Therefore, there should always be a set of comments preceding the definition of a set of starting conditions explaining their purpose within a Flex program:

```
 /*
 ** ProcessComments and EndComments labels precede those regular
 ** expressions that process (and throw away) the characters
 ** appearing between the "/*" and "*/" lexemes in a C/C++ program.
 */

%x ProcessComments  EndComments

 /*
 ** The "StringConstant" label precedes those regular expressions that
 ** process all the characters in a C/C++ string constant.
 */

%x StringConstant
```

Rule:

> **When defining starting condition labels in a Flex program, use meaningful names that describe the nature of the context-sensitive set of regular expressions. Group related starting symbols into the same statement; separate unrelated names into separate starting condition statements. Always preface each starting condition statement with a set of comments explaining the purpose of the context sensitive section.**

The definition section of a Flex program should take the following form:

- Header comments explaining the nature of the program.
- C/C++ header file includes and global definitions.
- Definitions for named regular expressions.

- Starting condition definitions.

Organizing the file in this fashion makes it easier to locate specific items in a large Flex program.

Guideline:

> **The definitions section of a Flex program should begin with a set of header comments, followed by C/C++ include statements and global definitions, then the definitions for named regular expression, and ending with the starting condition symbol definitions.**

6.1.2 - The Flex Rules Section

The Flex rules section consists of a set of statements that take the following form:

```
regular_expression  action
```

At least one (non-quoted) whitespace character separates the regular expression from the action. The action can be nothing, a single C/C++ statement, or a sequence of C/C++ statements enclosed within a pair of braces. To produce consistent looking programs, this standard requires that you enclose all statements (even the null statement) within a set of braces. For null actions, the Flex/Lex statement may take the following form:

```
regular_expression  {}
```

For Flex actions that require one or more C/C++ statements, the Flex statement should take the following form:

```
                   <--> At least four spaces  (typically one tab stop)
regular_expression     {

        << C/C++ statements go here >>

    }
<-->  Four spaces
<------> Eight Spaces
```

The regular expression (as per Flex rules) must begin in column one. You must follow this with a tab or at least four spaces (whichever produces the most white space) and end with a brace. A single blank line must separate the line containing the regular expression and the first C/C++ statement. A blank line also separates the last C/C++ statement and a single line containing the closing brace (that you must indent four character positions).

If you want to assign the same semantic action to several regular expressions using the alternation symbol ("|"), you would use a sequence like the following:

```
                   <--> At least four spaces  (typically one tab stop)
regular_expression1    |
regular_expression2    |
    .
    .
    .
regular_expressionN    {

        << C/C++ statements go here >>

    }
<-->  Four spaces
<------> Eight Spaces
```

As best you can, you should attempt to line up the "|" symbols in the same column (this may not be practical since the regular expressions may vary considerably in length).

A block of comments should precede each statement in the rules section. If it serves no other purpose, the comment will, at least, separate the start of a regular expression from the end of the previous regular expression in the rules section. If a statement contains a non-trivial regular expression[2], the comment will explain the types of strings the regular expression matches and provide several sample strings. Unless the corresponding action is also trivial (e.g., simply returning a token value), the comments should also give a basic description of the action associated with the regular expression (this does not excuse you from commenting the C/C++ code within the action, if appropriate).

Rule:

> **Every non-trivial regular expression appearing in the rules section of a Flex program should have a block of comments immediately preceding the statement. The comments should describe the type of patters the regular expression matches, provide several examples of strings the regular expression matches, and describe the actions taken if the lexer matches a string with this regular expression.**

You should always attempt to group logically related regular expression together in a Flex program. For example, one would normally expect to find the regular expressions for a set of reserved words adjacent to one another, likewise, one would expect the regular expressions for character and string constants or integer and floating point constants to be near one another. Organizing Flex programs in this fashion makes them easier to read.

Guideline:

> **Organize your Flex rules so that logically related regular expressions are adjacent to one another in the source file..**

6.2 - Yacc/Bison Specific Issues

This section documents style and formatting issues specific to the Yacc/Bison language. Keep in mind that a large portion of a typical Yacc/Bison program is C/C++ code and most of the C/C++ specific guidelines apply to that code.

6.2.1 - The Yacc/Bison Definitions Section

A Yacc/Bison (Yacc, hereafter) definitions section contains the following (optional) items:

- Comments.
- A literal block of C/C++ code.
- A "%union" definition.
- A "%start" statement.
- A set of "%token" definitions.
- A set of "%type" definitions.
- A set of "%left" definitions.
- A set of "%right" definitions.
- A set of "%nonassoc" definitions.

The definitions section should begin with a set of header comments (see the C/C++ section for details). Immediately following this should be any embedded C/C++ statements (typically include directives, global declarations, and function prototypes). This literal block of C/C++ code should not contain any functions, those belong in the auxiliary section at the end of the Yacc program.

Typically the "%union" sequence should follow the embedded C/C++ code. If the declarations in the "%union" section do not depend on any of the C/C++ statements (e.g., typedefs or include files), then the "%union" section should probably precede the literal block of C/C++ code. You should include an entry in the "%union" structure that takes the following form:

```
%union {
          .
          .
          .
        char nil;
    }
```

The purpose of the nil variable is to create a Yacc "type" that corresponds to a "void" function return result. You will see the purpose for this shortly.

The list of nonterminal symbol definitions (i.e., the "%type" definitions) should appear next in the source file. These definitions should all be grouped together. To make the code easier to read, you should specify the attribute type for all non-terminal symbols. If a particular non-terminal does not have an attribute associated with it, specify the "nil" type (defined above).

Rule:

> **Define all non-terminal symbols in a Yacc/Bison program in a %type definition (do not rely upon the implicit declaration that Yacc provides). Always associate an attribute type with that non-terminal. Use the dummy type "nil" (defined in the "%union" section) for those non-terminals that do not have a specific attribute value.**

Terminal symbol declarations (using "%token") should follow the non-terminal declarations. You should declare all terminal symbols other than single ASCII characters in this section. Be sure to define an attribute type ("nil" if the terminal symbol doesn't have an attribute type) for each terminal symbol. There should be only one token definition per "%token" statement. You should let Yacc assign the token value to this symbol.

Rule:

> **Define all terminal symbols in a Yacc/Bison program in a %token definition (ASCII character terminal symbols are an exception). Always associate an attribute type with that terminal symbol. Use the dummy type "nil" (defined in the "%union" section) for those non-terminals that do not have a specific attribute value.**

Guideline:

> **Let Yacc/Bison choose the token value associated with terminal symbols.**

Following the %token definitions, you should place any necessary %left, %right, and %nonassoc declarations. Use these declarations only to specify the precedence and associativity of your terminal symbols. Since the average person will be looking in the "%token" section for type declarations, you should not use these statements to define terminal symbols; doing so makes your program harder to read.

Rule:

> **Use the Yacc/Bison %left, %right, and %nonassoc statements to specify precedence and associativity only. Do not use these statements to define terminal symbols or attach a type to a terminal symbol.**

The Yacc definition section may also contain a "%start" statement that specifies the starting symbol for the context free grammar. If the "%start" statement is not present, Yacc uses the first production in the rules section as the starting production. Since most Yacc programmers use the default (and, therefore, expect others to use the default), you should not use a %start statement in your program. Instead, specify the starting production as the first production in the rules section.

Rule:

> **The starting production should be the first production in the rules section of a Yacc/Bison program. Do not specify the starting symbol by using the "%start" statement.**

6.2.2 - The Yacc/Bison Rules Section

The Yacc rules section consists of a syntax directed definition (a syntax directed definition is a grammar where productions may have an associated semantic rule). Each production in a Yacc syntax directed definition takes the following form:

```
NonTermSym : Right_Hand_Side { Semantic_Action };
```

The NonTermSym item is the name (declared in the "%type" section) of the non-terminal that this production defines. The Right_Hand_Side is a sequence of terminal and non-terminal symbols[3]. The semantic action is a sequence of zero or more C/C++ statements surrounded by braces. Note that you may combine several productions using the "|" symbol. For example, you may combine the following two productions:

```
A : 'a' B C { ++ACnt; };
```

```
A : 'b' B C { ++BCnt; };
```

into the single Yacc statement:

```
A : 'a' B C { ++ACnt; }
  | 'b' B C { ++BCnt; };
```

Note, however, that this still represents two separate productions. This is just a convenient shorthand for the former two statements.

The first thing to consider is how to format these productions. A typical Yacc production should take the following form:

```
NonTerm :
    Grammar Symbols For The Right Hand Side
    {

        << C/C++ Statements that implement the semantic action >>

    }
<--> Four character positions
<------> Eight character positions.
```

If you wish to merge two or more productions into a single production, the formatting should look like the following:

```
NonTerm :
    Grammar Symbols For The Right Hand Side
    {
        << C/C++ Statements that implement the semantic action >>
    }

|   Grammar Symbols for Right Hand Side #2
    {
        << C/C++ Statements that implement the semantic action >>
    }
<--> Four character positions
<------> Eight character positions.
```

Note that the left hand side (i.e., the non-terminal you are defining) sits on a separate line from the right hand side of the production. If necessary, the right hand side may be spread across several lines although you should indent each new line an appropriate amount.

Before each production in the Yacc rules section, you must place a block of comments explaining the purpose of the production, describe the type of strings it maches, provide some sample strings it matches, and give a basic description of the semantic action that takes place when a match occurs. The comments should also describe any attribute values (if any) returned through the non-terminal whenever a match occurs. These comments are especially important because syntax directed definitions are not easy to read; none but the most trivial could be considered "self-documenting."

Rule:

> **Every production in a Yacc/Bison program shall be preceded by a block of comments that describes the type of strings it matches, gives some sample strings it matches, discusses the attribute return value(s) for the production, and describes any semantic actions that take place when the parser successfully reduces a production.**

Semantic actions (the C/C++ code) should only appear at the end of a production. Indeed, the parsing technique that Yacc-generated parsers employ only execute a semantic rule when a reduction occurs. However, if you want to embed semantic actions in the middle of a production, it is easy to do so by simply creating a new production whose right hand side is the empty string (except for the semantic action). For example, you can easily translate the following production containing an embedded semantic action to a (nearly) equivalent set of productions that do not have embedded semantic actions:

```
A : B { semantic action #1 } C { semantic action #2 };
```

becomes:

```
A : B D C { semantic action #2 };
D : { semantic action #1 };
```

Indeed, this is such a simple translation that Yacc will actually do it for you automatically. While this translation, in theory, produces an equivalent syntax directed translation, there are a few Yacc restrictions that create some unusual results. First, any use of the "$$" token within an embedded semantic rule sets the attribute value for that embedded semantic action, not for the whole production as one might expect; therefore, this behaves exactly as though the embedded action appeared in a separate production. However, if you use tokens like $1, $2, $3, etc., in an embedded semantic action, this refers to the attribute values in the production containing the embedded semantic action, not the attributes in the dummy production. Several bugs occur in Yacc programs because of the inconsistent nature of the way Yacc handles the attributes in embedded semantic actions.

Another problem with embedded semantic actions is that they force the parse to use a given production whenever the match encounters an embedded semantic action during parsing. As a result, grammars that contain a large number of embedded semantic actions often wind up with reduce/reduce conflicts or shift/reduce conflicts. For this reason, and the reasons mentioned above, you should not use any embedded semantic actions in a Yacc grammar.

Enforced Rule:

> **Do not insert any embedded sematic actions into the right hand side of a production in a Yacc/Bison program. Instead, create a separate production whose right hand side is only the semantic action and insert the associated non-terminal for that production in place of the embedded semantic action.**

---

[1] This is the one exception to the rule that all C/C++ code must appear within the "%{" .. "%}" block. C/C++ comments (the only type of comments that may appear in a Flex program) may appear outside the C/C++ block to document the Flex code. [back]

[2] We will define non-trivial to mean anything other than a simple character string. [back]

[3] Yacc also allows embedded semantic actions in the Right_Hand_Side; you will shortly see that this standard does not allow you to embed rules within a production. [back]

< Previous section *(5 - Visual BASIC Specific Formatting Issues)* - **Contents** - **Next section** *(7 - Other Languages (Formatting Issues))* **>**

Software Development Guidelines

7 - Other Languages (Formatting Issues)

< Previous section *(6 - Lex/Flex and Yacc/Bison Specific Formatting Issues)* - **Contents** - **Next section** *(8 - Appendices)* **>**

7 - Other Languages (Formatting Issues)

< Previous section *(6 - Lex/Flex and Yacc/Bison Specific Formatting Issues)* - **Contents** - **Next section** *(8 - Appendices)* **>**

Software Development Guidelines

8 - Appendices

< Previous section *(7 - Other Languages (Formatting Issues))* - **Contents** - **Next section** *(9 - Glossary)* >

8 - Appendices

8.1 - Appendix A: Guidelines

Guideline: All routines should exhibit good cohesiveness. Functional cohesiveness is best, followed by sequential and global cohesiveness. Temporal cohesiveness is okay on occasion. You should avoid the other forms.

Guideline: Coupling between routines should be loose;

Guideline: Do not let artificial constraints affect the size of your routines. If a routine exceeds 150-200 lines of code, make sure the routine exhibits functional or sequential cohesion. Also look to see if there aren't some generic subsequences in your code that you can turn into stand alone routines.

Guideline: A module should implement an abstract data type. All interface to the module should be through a well-defined set of operations.

Guideline: If the data type you are creating depends upon a specific format, use names like int8, int16, int32, int64, real32, real64, and real80 (that is, a type name with the number of bits appended) to denote your types. If the data type does not depend on a specific representation, use a descriptive name (see the next section on naming conventions). Try to avoid the use of types in a program that rely on the underlying machine representation (alas, this is not always possible).

Guideline: Variable declarations should appear on separate lines. If desired, the type specification should appear on a separate line as well. Variable and type names should be aligned in columns and easy to find and read.

Guideline: Avoid all identifier abbreviations in your programs. When necessary, use standardized abbreviations or ask someone to review your abbreviations. Whenever you use abbreviations in your programs, create a "data dictionary" in the comments near the names' definition that provides a full name and description for your abbreviation.

Guideline: Try to make most identifiers unique in the first few character positions of the identifier. This makes the program easier to read.

Corollary: Never use a numeric suffix to differentiate two names.

Guideline: Avoid using Hungarian notation and any other formal naming convention that attaches low-level type information to the identifier.

Guideline: If you want to differentiate identifiers that are constants, type definitions, and variable names, use the suffixes "_c", "_t", and "_v", respectively.

Guideline: Avoid using symbols in identifiers that are easily mistaken for other symbols .

Guideline: Avoid homonyms in identifiers.

Guideline: Avoid misspelled words and names that are often misspelled in identifiers.

Guideline: When using multi-way selection statements (case/switch) sort the cases numerically (alphabetically) or by frequency of expected occurrence.

Guideline: Loops with a single exit point are more easily understood.

Guideline: Avoid empty loops. If testing the loop termination condition produces some side effect that is the whole purpose of the loop, move that side effect into the body of the loop. If a loop truly has an empty body, place a comment like "/* nothing */" or "{null statement}" within your code.

Guideline: Make each loop perform only one function.

Guideline: An expression should not produce any side effects.

Guideline: There should be no spaces between a unary operator (e.g., "-") and the object on which it operates.

Guideline: There should be at least one space on either side of a binary operator.

Guideline: Operators that select a component of a larger object (e.g., "." for records/structures and "[ ]" for arrays) should be adjacent to the object(s) they operate upon.

Guideline: Objects that separate items (e.g., "," and ";") should immediately follow the previous object. If a second object follows the separator, there should be a space between the separator and the second object.

Guideline: Bracketing symbols (e.g., "(" and ")", "[" and "]", and "{" and "}") should have one space on the "open" end of the symbol, that is, to the right of "(", "[", and "[" and to the left of ")", "]", and "}".

Guideline: Indentation should be three to four spaces in an indented control structure with four spaces probably being the optimal value.

Guideline: For statements that are too long to fit on one physical 80-column line, you should break the statement into two (or more) lines at points in the statement that will have the least impact on the readability of the statement. This situation usually occurs immediately after low-precedence operators or after commas.

Guideline: If a module contains some cross references to other documents, there should be a comment that takes the form "@ text #link#location text @" that provides the reference to that other document. In this comment, the "@" represents the language's comment delimeter(s), "text" represents some optional text (typically reserved for html tags), and "location" is some descriptive text that describes the document (and a position in that document) related to the current section of code in the program.

Guideline: The order of the prototypes in a C/C++ program should match the order of the functions appearing in the source module.

Guideline: For internal use, all compiles should expand all assertions to abort the program if the assertion turns out to be false.

Guideline: Don't avoid the use of #ifdef and #ifndef statements in your program because you are worried about making your program harder to read. Tools exist to remove these #ifdef and #ifndef statements thus eliminating the clutter.

Guideline: Avoid fancy formatting that will not transfer well to HTML.

Guideline: All functions (public or private) appearing in a source module will have an associated prototype or forward declaration. Public prototypes must appear at the appropriate point in the interface section (i.e., after the const, type, and var sections), private prototypes and forward declarations must appear at the appropriate spot in the implementation section (i.e., after const, type, and var sections, but before the first real program unit bodies).

Guideline: The order of the Pascal prototypes and forward declarations should match the order of the functions/procedures appearing in the source module.

Guideline: You should use the "{$ " and "}" symbols to surround a Delphi/Pascal compiler directive since the result is easier to read than the same directive surrounded by "(*$" and "*)".

Guideline: Use the "{" and "}" delimiters for Delphi/Pascal endline comments (comments appearing at the end of a line that contains some other statement). Use the "(*" and "*)" delimiters for single line comments (a single comment appearing on a line by itself).

Guideline: You should organize Delphi unit source files with the procedures and function that are not attached to a particular class at the beginning of the implementation section. Methods associated with local class objects should follow these procedures and functions. Finally, methods associated with event handlers for components on a form should appear at the end of the source file.

Guideline: When naming Delphi components you place on a form, append a suffix string that consists of an underscore followed by a standard string that denotes the type of that component (e.g., "_lbl" for TLabel objects).

Guideline: If two or more components are related (e.g., a TLabel object that describes the type of input for a
TEdit object) then use the same name with different suffixes (e.g., Month and Month_lbl). Note that this is an exception to the rule that identifiers should not have more than a few characters in common as their prefix characters; it also violates the rule that two identifiers should not be the same except for a type suffix. This exception exists to help overcome the limitation that a component cannot be a field of a record or class except a TForm object.

Guideline: Use Lex/Flex definitions to give meaningful names to common or oft-used regular expressions within a program. Once you create a definition, be sure to use the definition's name as appropriate throughout the Lex/Flex program.

Guideline: Define the macro "StandardREs" to be replaced by the literal constant "0". Then use "BEGIN StandardREs;" rather than "BEGIN 0;" to turn off context sensitivity in a Lex/Flex program.

Guideline: The definitions section of a Lex/Flex program should begin with a set of header comments, followed by C/C++ include statements and global definitions, then the definitions for named regular expression, and ending with the starting condition symbol definitions.

Guideline: Organize your Lex/Flex rules so that logically related regular expressions are adjacent to one another in the source file..

Guideline: Let Yacc/Bison choose the token value associated with terminal symbols.

8.2 - Appendix B: Rules

Rule: Never shorten a routine by dividing it into n parts that you would always call in the appropriate sequence as a way of shortening the original routine.

Rule: Each module should completely reside in a single source file. If size considerations prevent this, then all the source files for a given module should reside in a subdirectory specifically designated for that module.

Rule: If a particular language processing system does not support modules of any kind, simulate those modules by physically grouping the objects in the source code. Be sure to access the module using only "approved" interfaces. Always check for inconsistencies when reviewing your code.

Rule: If a built-in type has different semantics on different architectures or in different compilers, always use a set of type definitions that let you easily change adjust the program to a different architecture. It is dangerous to assume a particular object uses a specific data format (e.g., two's complement binary or IEEE floating point). It is even worse to assume an object has a fixed number of bits. You should avoid using predefined types in a language.

Rule: All variable, constant, and type definitions should occur at the very beginning of the program unit whose limits define the scope of the object.

Rule: If you cannot define an object at the beginning of the program unit to which it belongs, then put a place-holder comment at the beginning of the block and define the variable as soon as possible within the program unit. You should place a comment near such a definition to remind the reader to update the comment at the beginning of the block if the actual definition ever changes.

Rule: Associated with any set of variable declarations will be a set of comments known as the "Data Dictionary." This data dictionary will describe the name and purpose for each variable. The Data Dictionary will also describe any constraints or assumptions on the use of the variables.

Rule: All identifiers that represent words or phrases must be English words or phrases.

Rule: You should never use alphabetic case to denote the type, classification, or any other program-related attribute of an identifier (unless the language's syntax specifically requires this).

Rule: Capitalize the first letter of interior words in all multi-word identifiers.

Rule: Avoid using all upper case characters in an identifier.

Rule: All identifiers should be pronounceable (in English) without having to spell out more than one letter.

Rule: The classification suffix should not be the only component that differentiates two identifiers.

Rule: Avoid misleading abbreviations and names.

Rule: Do not use names with similar meanings for different objects in your programs.

Rule: Do not use similar names that have different meanings.

Rule: Programs written in a standard imperative language (e.g., C/C++, Pascal, Ada, Visual BASIC, Delphi, etc.) will use the modern versions of the standard control constructs. If the language does not directly support these control structures, the programmer will simulate them using rules appearing elsewhere in this document.

Rule: If your code contains a chain of if..elseif..elseif.......elseif..... statements, do not use the final else clause to handle a remaining case. Only use the final else to catch an error condition. If you need to test for some value in an if..elseif..elseif.... chain, always test the value in an if or elseif statement.

Rule: Always use the most appropriate type of loop (categorized by termination test position). Never force one type of loop to behave like another.

Rule: "FOR" loops should always use an ordinal loop control variable (e.g., integer, char, boolean, enumerated type) and should always increment or decrement the loop control variable by one.

Rule: All loops should have one entry point. The program should enter the loop with the instruction at the top of the loop.

Rule: Avoid side-effects in the computation of the loop termination expression (others may not be expecting such side effects). Also see the guideline about empty loops.

Rule: Code, as much as possible, should read from top to bottom.

Rule: Related statements should be grouped together and separated from unrelated statements with whitespace or comments.

Rule: The assumable precedences are: [highest]: {operands} {unary operators} {*,/,mod} {+.-} {<, <=, =, <>, >, >=} {and, or}. Note that you can only assume left associativity for {*,/,mod} and {+,-}. Assume all other operators are non-associative and that you must use parentheses if they are next to one another in an expression. If you cannot assume the precedence according to the rule above, use parentheses to explicitly state the precedence.

Rule: If an expression depends upon short-circuit evaluation to produce a correct answer, you must explicitly state this in a comment nearby.

Rule: A program should never use the value of a variable modified as a result of a side effect within that same expression.

Rule: Never execute an expression solely for the side effects it produces.

Rule: At least one blank line must separate a comment on a line by itself from a line of code following or preceding the comment.

Rule: The standard layout scheme is the Pure Block format. For languages that do not support modern structured control statements, this coding standard specifies an emulation of these statements that allows the use of the Pure Block layout format.

Rule: Always put a blank line between any block statement and the statement(s) it encloses.

Rule: If an actual or formal parameter list is too long to fit a function call or definition on a single line, then place each parameter on a separate line and align them so they are easy to read.

Rule: All comments will be high-quality comments.

Rule: All C/C++ programs will use the control structures found in the "ratc.h" header file in place of the traditional C/C++ control structures.

Rule: All private objects (that is, variable and function names that should remain local to a given source file) must have the keyword "static" preceding them in a C/C++ source file.

Rule: C/C++ Functions that are also public (non-static) should appear near the beginning of the source file.

Rule: The makefile for a given project must offer a "standard/debug" compilation and a "production" compilation option. The "production" compilation option should define the macro symbol NDEBUG for every C source file it processes.

Rule: You should use assertions throughout your code to check degenerate and "impossible" conditions. You should also use assertions to check the sanity of parameters input to a function.

Rule: If a function can succeed or fail in addition to returning some value, the function should return the failure status as the function result and return the other value through a reference parameter. This allows you to use the _assert and _nassert macros to check the return status of the function.

Rule: If your language provides exception handling capabilities, use them rather than manufacturing your own tests for exceptional conditions.

Rule: All debugging code must disappear if the symbol "NDEBUG" is defined. That is, you must surround all debugging code with "#ifndef NDEBUG" and a corresponding #endif.

Rule: Functions should never attempt to return two (or more) values through a single parameter or function return result. If a function truly needs to return two different values, return them in separate locations (e.g., through pass by reference parameters).

Rule: Use a semicolon in a Pascal program wherever it is optional.

Rule: All private objects (that is, variable, function, and procedure names that should remain local to a given source file) must appear in the implementation section in a Pascal/Delphi source file. Variables appearing in the interface section are public objects that other modules can be use.

Rule: Pascal functions and procedures that are also public should appear near the beginning of the source file.

Rule: Functions in Delphi code should never attempt to return two (or more) values through a single parameter or function return result. If a function truly needs to return two different values, return them in separate locations (e.g., through pass by reference parameters). If one of the return values denotes an error condition, use Delphi's exception handling facilities to raise an exception.

Rule: All C/C++ statements appearing in the definitions section of a Lex/Flex or Yacc/Bison program should appear at the beginning of the sections bracketed by a pair of lines containing the "%{" and "%}"

tokens.

Rule: The definition section of a Flex/Lex or Yacc/Bison program should contain only those C/C++ definitions and declarations necessary for the rules section of the program. You should place all other C/C++ code in the auxiliary code section of the program.

Rule: The Lex/Flex and C/C++ statements in a Lex/Flex program must contain a liberal number of comments describing the code. Do not assume that the regular expression is "self documenting."

Rule: Always precede each Lex/Flex definition with a comment that describes the pattern matched by the corresponding regular expression. Also provide several example strings that the regular expression matches. Don't assume that the definition's name completely documents the corresponding regular expression.

Rule: The identifiers you specify for Lex/Flex "starting conditions" should be meaningful; they should describe the context handled by the starting condition.

Rule: When defining starting condition labels in a Lex/Flex program, use meaningful names that describe the nature of the context-sensitive set of regular expressions. Group related starting symbols into the same statement; separate unrelated names into separate starting condition statements. Always preface each starting condition statement with a set of comments explaining the purpose of the context sensitive section.

Rule: Every non-trivial regular expression appearing in the rules section of a Lex/Flex program should have a block of comments immediately preceding the statement. The comments should describe the type of patters the regular expression matches, provide several examples of strings the regular expression matches, and describe the actions taken if the lexer matches a string with this regular expression.

Rule: Define all non-terminal symbols in a Yacc/Bison program in a %type definition. Always associate an attribute type with that non-terminal. Use the dummy type "nil" (defined in the "%union" section) for those non-terminals that do not have a specific attribute value.

Rule: Define all terminal symbols in a Yacc/Bison program in a %token definition (ASCII character terminal symbols are an exception). Always associate an attribute type with that terminal symbol. Use the dummy type "nil" (defined in the "%union" section) for those non-terminals that do not have a specific attribute value.

Rule: Use the Yacc/Bison %left, %right, and %nonassoc statements to specify precedence and associativity only. Do not use these statements to define terminal symbols or attach a type to a terminal symbol.

Rule: The starting production should be the first production in the rules section of a Yacc/Bison program. Do not specify the starting symbol by using the "%start" statement.

Rule: Every production in a Yacc/Bison program shall be preceded by a block of comments that describes the type of strings it matches, gives some sample strings it matches, discusses the attribute return value(s) for the production, and describes any semantic actions that take place when the parser successfully reduces a production.

8.3 - Appendix C: Enforced Rules

Enforced Rule: Never redefine an existing type.

Enforced Rule: Always explicitly declare all variables (and other identifiers) unless the language does not allow this.

Enforced Rule: All identifiers must be "case-neutral."

Enforced Rule: Do not reuse existing standard library routine names in your program unless you are specifically replacing that routine with one that has similar semantics (i.e., don't reuse the name for a different purpose).

Enforced Rule: GOTOs, if they appear at all in a program, must be okayed by a peer review of at least two peers, both of whom agree the resulting code with a GOTO is easier to understand than equivalent code without a GOTO. GOTOs should only be used in exception processing statements or after exhausting several other attempts at writing clear code without the GOTO. Some code is actually easier to read with a GOTO statement than without, but it is easy to develop a mental block that would suggest the use of a GOTO when a clearer solution exists, hence the peer review.

Enforced Rule: If you use tabs to indent your code, insert a comment at the very beginning of the program that states the number of positions for each tab stop. E.g., "/* This program is formatted using four character position tabstops. */"

Enforced Rule: Source code lines will not exceed 80 characters in length.

Enforced Rule: All comments will be up to date. If a programmer makes changes to the code, that programmer is responsible for updating the internal comments and any external documentation affected by those changes.

Enforced Rule: If a modules contains some defects that cannot be immediately removed because of time or other constraints, the program will insert a standardized comment before the code so that it is easy to locate such problems in the future. The four standardized comments are "@_#defect#severe_@, "@_#defect#functional_@", "@_#defect#suspect @", and "@_#defect#enhancement_@" where "@" denotes the comment delimiter and "_" denotes a single space. The spelling and spacing should be exact so it is easy to search for these strings in the source tree.

Enforced Rule: All intermodule communication in C/C++ programs must take place through header files (".h" files). All extern directives, public class definitions, public type definitions, and public constants must appear in the header file that all interested modules will include.

Enforced Rule: All C/C++ functions (public or private) appearing in a source module will have an associated prototype. The prototypes for all functions will appear near the beginning of the source file (typically after the include and define directives and any other type definitions also appearing there).

Enforced Rule: All "temporary" debugging statements you add to a program, no matter how temporary they seem, must be protected with "#ifndef NDEBUG" and "#endif" statements. This is the only line of defense against forgetting to remove the code before compiling a production version of the program.

Enforced Rule: By default, Borland's Pascal and Delphi compilers have most of the optional run-time checks disabled. During software development you should enable all these checks. Turn them off when shipping production code.

Enforced Rule: All "temporary" debugging statements you add to a Pascal/Delphi program, no matter how temporary they seem, must be protected with "{$ifndef NDEBUG}" and "{$endif}" directives. This is the only line of defense against forgetting to remove the code before compiling a production version of the program.

Enforced Rule: Do not insert any embedded sematic actions into the right hand side of a production in a Yacc/Bison program. Instead, create a separate production whose right hand side is only the semantic action and insert the associated non-terminal for that production in place of the embedded semantic action.

< Previous section *(7 - Other Languages (Formatting Issues))* - **Contents** - **Next section** *(9 - Glossary)* >

Software Development Guidelines

9 - Glossary

< Previous section *(8 - Appendices)* - **Contents**

9 - Glossary

Big Bang Testing: A testing strategy wherein you design and code the entire program before testing. Suitable only for very small programs.

Big Bang

Integration

Testing: As you develop each module, you test it. This localizes any errors to that one module (so they are easy to find and correct). After writing and testing all modules, you integrate them into the program and test the entire product. Assuming you've removed all the bugs within the modules themselves, the only bugs remaining are in the interfaces between the modules.

Black box test

Data Generation: Generating test data for a program using the functional specification alone (no source code).

Case-Neutral: An identifier is "case-neutral" if you can compile a program using it with two different compilers -one that is case sensitive (with respect to identifiers) and one that ignores the case of alphabetic characters in an identifier. This means that you must spell the identifier exactly the way you declared it (with respect to case) and that no two (different) identifiers exist whose spelling differs only with respect to the case of the alphabetic symbols.

Debugging: The process of locating and removing defects in a software system. Not to be confused with testing (a different activity).

Driver: An empty function that calls functions under test. Generally it contains just enough code to set up parameters and globals prior calling the function.

Identifier Class: The object an identifier specifies can be classified as a constant, type definition, variable, function, procedure, iterator, program identifier, etc.

Incremental

Testing: A testing strategy in which you develop a module, test it, integrate it into the system (such as it exists at that point), and then test the system. This is an excellent testing strategy for very large systems.

LOC: Lines of code.

KLOC: Thousands of lines of code.

Program Unit: A function, procedure, iterator, subroutine, process, main program, or other comparable object. Sometimes includes a compound statement (block). One salient, defining feature of a program unit is that it limits the scope of objects (e.g., variables) defined within the unit.

RatC: RATional C. This is a macro package that adds modern control structures to the C and C++ programming languages.

Regression

Testing: Regression testing is the process of always running the same sequence of tests on a program unit every time the program unit changes. This verifies (within the bounds of the test) that the new code works and it doesn't break any old code, either.

Scaffolding: Any code written to help test a program that isn't actually part of the program under test. Stubs and drivers are examples of scaffolding code.

Stub: An empty function that replaces a function that is yet to be written. Generally, a stub provides absolute minimal functionality, it accomplishes just enough to test the calling code with just a few special cases. Also see Driver.

Testing: The process of supplying data to a program with the hope of encountering defects in the software. Generally involves test data generation, experimentation, and analysis phases.

Waterfall Model: A software development model in which work flows from one step to the next. See the text of this document for details.

White box test

Data Generation: Generating test data for a program by studying the source code and choosing sets of input values to feed the program that will achieve certain goals like executing each statement in the program at least once.

< Previous section *(8 - Appendices)* - **Contents**