AIPS++ Documentation System

DRAFT Version

Darrell Schiebel

Last modified: $Date: 2003/07/16 03:48:51 $ By: $Author: aips2adm $

1 Introduction

This document discusses the C++ source documentation system for the AIPS++ coding effort. This system is needed to provide a means of extracting documentation for source code. One of the important objectives of any documentation standard is to ensure that the source code and the documentation remain consistent. For this, it is important for as much information to be extracted from the actual source code as possible, and for the comments to be located as close as possible to the code to which they pertain as possible. In addition to these concerns, the syntax of this system is as simple as possible.

Documentation standards are essentially arbitrary. Because neither the syntax nor style is predetermined. So, as with any language, the choice is arbitrary. The only constraint is consistency (wherever possible).

Terseness and readability are competing concerns. On the one hand it is nice to have terse keystrokes like emacs key bindings to denote the different documentation keywords, but is also important to have the documentation understandable to the reader of un-processed documentation.

With these goals in mind, we have developed a keyword scheme for denoting different pieces of the documentation. This method is similar to the approach taken in "genman" by Bob Mastors, and the troff macros. We have, however, tried to closely tie the comments to the source in hopes that the extractor can get much of the basic information from the actual source. Also, the syntax is more expressive than "genman" to allow for the specification of nodes in a hyper-text system.

The hyper-text metaphor was chosen because it allows for scope, encapsulation, and detail hiding which are important vehicles for reducing the complexity of software documentation. Information about particular member functions can be hidden, and only displayed when information is required for that function. This is not the only way the information can be accessed. It can be printed as demonstrated by the GNU *texinfo* system, and it could just as easily be stored in a database system.

2 Documentation

The language for documentation which is presented here has a very simple syntax, but it allows for documentation scope which is used to specify comment nodes and the interdependency of comment nodes.

2.1 Comment Scope

The concept of scope will be important in this discussion. Each globally visible, in terms of **C++**, symbol will have a comment scope. This scope extends from the end of the globally visible symbol definition backwards through the symbol to be previous symbol definition. The *name* of the scope becomes simply the name of the symbol being defined. The use of the definition instead of the declaration implies that the include files will be the primary source of documentation.

The document scope defines the logical comment units. This concept will be important in various ways such as referencing elements of other comment units. Within a given comment scope, there may be other sub-scopes. For example, the comments for each of the members of a class have a comment scope, the members of an enumeration have a comment scope, and the members of structures each have a comment scope. Thus we have:

```
   class x {

   };
|
+---String's comment scope.
|
|  class String {
|    public:
||
|+---String::~String's comment scope.
||
|      ~String();
|
   };
etc.
```

So there is a set of global comment scopes which consists of the comment scopes of all of the globally visibly definitions, and within these scopes there are sub comment scopes, etc. This hierarchy forms the basis for the documentation system proposed in this paper.

2.2 Comment Nodes

For our purposes, comment nodes are the informational portion of the documentation extracted from the source code. There will be nodes which describe each of the libraries, classes, functions, enums, etc. There will also be links connecting related nodes. In the end, the comment system will be a directed graph of nodes and links.

2.2.1 Automatically Created Nodes and Links

Some nodes and links are automatically created by the comment extraction system. A node is created for each class, enum, and any other externally visible **C++** symbol. Links are created between derived classes and parent classes and between class member variables and the documentation for the given class. So for example, if we had the following classes:

```
class x {};
class y : public x {};

class A {};
class B : public A {};
class C : public B {
  private:
    y myy;};
```

The following node interconnections would be created by the system:

```
                   A
                   *               Where links denoted with a ``*'' are
                   |               bidirectional.
                   V
                   B      x
                   *      *
                   |      |
                   V      V
                   C----->y
```

These links are created because they represent fundamental relationships within program entities, and these relationships should be reflected with in the documentation system.

2.2.2 Documenter Created Nodes and Links

The person creating the documentation is free to create nodes at any point. The keyword `.SEC` introduces a new node into the hyper-text system. Its format is:

```
.SEC ( <node name> ) <node label> {
<text for node>
}
```

Thus a node is specified with the name `<node name>` and the label to reference the node is `<node label>`. The contents of the node is simply entered between the braces, `<text for node>`. This creates a reference where the `.SEC` command occurred, and a node with the appropriate text. This node is created in the current global comment scope.

So if we wanted to create a hyper text link to a node about the problems of using `strlen`. We might do something like the following:

```
.SEC(Warnings) Precautions concerning the use of strlen. {
  One should NEVER pass null pointers into the function strlen.
}
```

We would have a node, `Warning`, in the resulting documentation which would look like:

```
     * Precautions concerning the use of strlen.
```

and when this node is entered the node containing the text "`One should NEVER ...`" will be displayed.

2.2.3 Reference

Often the developer may wish to reference a node in more than one place, at the creation point and elsewhere. In this case, the reference keyword is used:

```
.REF( <node name>[!<scope name>] ) [ <node label> ]
```

This specifies that a reference to the node, <node name>, should be inserted at this point with the label <node label>. The portion "!<scope name>" is an optional scope specifier which can be used to indicate the scope of the node which is being referenced. If no scope specifier is given, the current comment scope is assumed. The "<node label>" portion is also optional, and it permits different labels to refer to the same comment node. When no label is specified, the original label for the referenced node, from .SEC, is used for user created nodes. For automatically created nodes, the summary string which is specified by the .SUMMARY keyword is used if available, otherwise the name of the node is used.

So, for example, if we wanted to insert a reference to the node created above. We would simply insert the following:

```
.REF(Warnings) strlen warnings
```

Thus another reference to the Warning node will be created.

2.2.4 Groups

Sometimes it may be helpful to specify a general or default comment for a group of code symbols. In this way, the person doing the documentation can group a set of source code symbols together and indicate that the comment applies to all of them. The syntax for a group is:

```
.GROUP (<node name>) [
   <source code lines>
]
```

In this case, the *group* follows the same comment scope rules as a regular *symbol* from the source code. If there are instances within the group where a comment node would be created, but where no comments exist from which to create the node, a reference will be created and it will refer to the node for the .GROUP. If a symbol within the group has a comment to put into a node, a node will be created with the comment, and the *more general* .GROUP comment node will be referenced at the beginning of the comment created for the symbol. So for example:

```
class Complex {
  public:
    // This is one of a set of accessor functions, .COD{getr} and .COD{geti},
    // which simply return the .EMP{real} and .EMP{imaginary} portions of
    // the real number.
    // .GROUP(complex accessors) [
    //This member function is special.
    float getr(){return r;}
    float geti(){return i;}
    // ]
```

Results in the node hierarchy which follows:

```
                                Where links denoted with a ``*'' are
      Complex                   bidirectional.
       *     *
       |     |
       V     V
      getr   complex accessors (in lew of a node for geti)
       *
       |
       V
complex accessors
```

Thus the `.GROUP` provides a way of specifying a general overview or default comment for a group of symbols. Nesting of `.GROUP` constructs is permitted, but unfortunately the semantics are currently not completely thought out *(so this could change).*

2.2.5 Libraries

Because there are possibly a huge number of root comment nodes, those which are not pointed to by a link from another node, it is important to have a method for grouping these nodes into logical sets. Libraries provide such a grouping in the same way that class libraries provide a grouping for **C++** classes. In fact, the libraries specified for comment nodes will probably correspond to **C++** class libraries. This is one way to provide structure to an otherwise flat collection of comment nodes. The format is simply:

```
.LIB(<library name>)
```

associates the current global comment node with the library `<library name>`.

2.3 Summary

For any comment scope, it is possible to provide a one line summary of the definition to which the comment scope belongs. The format of this statement is simply:

```
.SUMMARY <one line summary>
```

This will allow for generation of indexes, one line summary on *man* pages, etc.

2.4 *Special* Commands

There may be some need for specialized commands for providing special format for characters. For example, it may be important to add a special command to render the existential operator for *pre* and *post* conditions[1] or other mathematical functions for specifying a given software component. As this documentation system expands, these commands will be added as necessary.

Two of these *special* command will be supported initially. There are so many commands which could fit into this category that if they all were implemented one would end up with a language of complexity similar to TEX. The command to denote emphasis is:

```
.EMP{ <emphasized text string> }
```

It will provide emphasis for the specified string. The command to indicate that the string is taken from the source code is:

[1] See the AIPS++ note number 104 for a discussion of *pre* and *post* conditions.

```
.COD{ <source code string> }
```

these two commands were chosen because they are very central to the idea of documentation in general and source code documentation in particular.

In addition, there are two *escape* commands. Which force special or no processing of a comment string. The escape character is the backslash, "\". It specifies that the next character is to be taken as a literal. Thus to enter ".SEC" in a comment it would be "`\.SEC`". A pound sign, "`#`", indicates that the rest of the line should be ignored. It provides a way of inserting comments that will *not* make it to the final comment node.

3 Example

This chapter provides an example of the usage of this system. It lays out an entire header file:

```
#if !defined(COMPLEX_H_)
#define COMPLEX_H_

#include <iostream.h>
#include <string.h>

//.SUMMARY Simple complex number class.
//.LIB(math)
//
//This class implements a simple complex number class. It is relatively
//simple and provides only for the simple arithmetic operations on
//complex numbers.
//  .SEC (Complex Problems) The problems with using this class. {
//      This class is a very simple class the number of methods needs
//      to be expanded and the class needs to be made more general. }
class Complex {
  public:
    // This addition operator is implemented as a .COD{friend} function
    // to allow casts to occur on the right and left hand sides of the
    // operator.
    friend Complex operator+(const Complex &,const Complex &);

    // This is one of a set of accessor functions, .COD{getr} and .COD{geti},
    // which simply return the .EMP{real} and .EMP{imaginary} portions of
    // the real number.
    // .GROUP(complex accessors) [
    float getr(){return r;}
    float geti(){return i;}
    // ]

    // The subtraction operator is implemented as a member function
    // to demonstrate the problems with casting of LHS arguments in
    // member functions, i.e. .COD{b = 1 + a} where b and a are objects
    // of type complex.
    Complex operator-(const Complex &b){ return(Complex(r - b.r, i - b.i));};

    // The constructor will take 0-2 arguments, and it simply creates a new
    // Complex object from the two float parameters. The default value
    // is zero for both the real and imaginary components.
    Complex(float x=0, float y=0) : r(x), i(y), id(0){};
  private:
    float r,i;              //# internal data
//# operator=(Complex &);   //dis-allow the bit copy of Complex objects
};
```

```
#endif
```

Note that there is nothing to prevent the use of **C** comments. For example, the first part of this might be:

```
#if !defined(COMPLEX_H_)
#define COMPLEX_H_

#include <iostream.h>
#include <string.h>

/*********************************************************************
.SUMMARY Simple complex number class.
.LIB(math)

This class implements a simple complex number class. It is relatively
simple and provides only for the simple arithmetic operations on
complex numbers.
  .SEC (Complex Problems) The problems with using this class. {
      This class is a very simple class the number of methods needs
      to be expanded and the class needs to be made more general. }
*********************************************************************/
class Complex {
  public:
```

etc.

4 Conclusion

In this paper, I have presented a *bare bones* set of commands which can be used to create hyper-text documentation for a software system. As much information as possible is extracted from the source actual code. I have also tried to make the system as flexible and simple as possible.

The commands and conventions discussed in this document are intended to allow for easy extraction of the information from the source files and to allow for easy insertion into a hyper-text system, like the GNU *texinfo* system. However, the information can also be extracted and used in other sorts of systems. For example, each of the nodes created from the global symbols, classes, enums, etc., are the documentation for all of the global symbols in the system. Thus, it would be relatively easy to integrate this information in to a key value scheme to easily retrieve the documentation on a particular symbol, or even to check to see if a symbol has been used when defining another global symbol.

While the keywords discussed in this paper are simple and small in number. They can express the required information for a documentation system. The approach, however, is general and easily extensible. However, there are other AIPS++ notes which discuss possible documentation systems[1]. These papers provides valuable information, for example AIPS++ note 104 provides a discussion of more formal function specifications. However, to my knowledge, these papers do not provide the notion of scope of comments which are important for intimately connecting source code and comments and for separation of comments into nodes.

[1] See AIPS++ notes: 104.texi, 124.text, and 137.text.