AIPS++ Documentation System (rev 2)
DRAFT Version

Darrell Schiebel

24-June-1996

1 Introduction

The objective of this system is to provide a way for programmers to document source code in a way which allows documentation to be extracted and converted into a reference document. To accomplish this, it is necessary to provide a way for the programmer to specify information which supplements the regular comments. This "extra" information is know as markup. The markup provides a means of structuring the comments, and allows them to be formatted in ways more sophisticated than ASCII text.

This system allows both the source code and the documentation for the source to exist in the same file. This close proximity will increase the chances that the reference document will remain up to date. In addition, the goal is to have a parser which understands most of the C++ language. This allows much of the reference document to be generated automatically from the source code.

The first revision of this system used a language similar to the `roff` dialects. This language was abandoned in favor of SGML, Standard Generalized Markup Language. There are a few reasons for this:

- Since SGML is an ISO/ANSI standard many tools are likely to be available in the future to support it.
- Generalized markup is a better model for information reuse and flexibility. It allows the information it structures to be used by a variety of applications.
- SGML provides a good basis for future expansion.

- Many successful applications already use SGML, and the work they and the various standardization groups have done can be leveraged to our benefit.

1.1 Generalized Markup

The motivation for generalized markup is a departure from that of traditional markup languages. With traditional markup languages, the goal is to add additional information to a document to allow it to be formatted for presentation. The markup is formatting information. With generalized markup, however, the markup is used convey the logical structure of the information. How this structure is formatted should not be the concern of the writer. The provider of the information should only be concerned with providing the important information with the appropriate logical structure. The actual formatting will be performed by systems which will use this information. The logical structure of the information is much more valuable than the way it should be formatted. If the information is structured correctly, a variety of post-processors should be able to use the information which is based on a given tag-set.

So for example a piece of procedural markup might look like:

```
.TH DF 1V "16 September 1989"
.SH NAME
df \- report free disk space on file systems
.SH SYNOPSIS
.B df
```

This is a section taken from the `df` Unix man page. Here, the *.TH* command sets up the header – the reference page is `DF`, the section is *1V*, and the last field is the date of the most recent change. The *.SH* command sets up a section heading with the given label, and the *.B* command displays text in bold. This is relatively typical of procedural markup. While all of the information necessary to present a nicely formatted man page is there, it is all superficial. The sections do not convey the content. The section command is just a section command, and does not convey the fact that the section is the synopsis. In a generalized markup language, SGML in this case, this section of the man page might be represented as:

```
<man ref=DF sect='1V' date="16 September 1989">
<name> df
<summary> report free disk space on file systems </summary>
<synopsis> <com>df</com>
```

Here the document is labeled as a man page and the <*man*> attributes [1] are the information that previously generated the header. The name is no longer the text of a section heading; it is labeled as a <*name*>. The synopsis is labeled as a synopsis, <*synopsis*>, instead of a section with the heading "SYNOPSIS". The "df" reference in the synopsis is a command, <*com*>, instead of a bold section of arbitrary text.

Generalized markup allows the important information to be processed unambiguously by many different systems. With generalized markup, a <*synopsis*> may be treated as a section for printed text, but for a hypertext system the synopsis may only be displayed if someone presses the synopsis button (however that is done). The information represents a synopsis instead of formatting details. The formatting decisions are left to the discretion of the system which will use the information.

Generalized markup is one tool which will allow information to be reused for a variety of purposes. Once freed from a particular formatting language, many applications can utilize the same information source for many purposes. The same logically structured information can be used to generate a printed manual, a hypertext system, a database, or practically anything else that can be done with information. All that is necessary is that the proper set of tags are defined and processors exist to manipulate information structured with the tags.

1.2 Standard Generalized Markup Language

SGML is in some sense a meta-language. It allows one to specify one of a family of generalized markup languages. SGML provides the language to specify the elements of the markup language and the rules for composition of these elements. This capability enables mechanical analysis of documents which utilize an SGML specified markup language. In addition, since SGML is an ANSI/ISO standard compatibility of SGML systems is guaranteed. In fact, SGML has been used for representing brail, music, hypermedia, information for tutoring systems, published books. SGML will be one of the tools which will help to make free interchange of information a reality.

1.2.1 Document Type Definition

One of the most important sections of an SGML document is the section which defines the tag-set and composition rules which will be used in the

[1] Single or double quotes can be used to distinguish literal values in SGML. If an attribute value is a single word, quotes are not required.

document. This "grammar" is typically located in another file and is included at the top of the document in much the same way that LaTeXmacro packages would be included in LaTeXdocuments. This "grammar" section is called the *Document Type Definition*, **DTD**.

Elements The **DTD** specifies the elements, the tags which delimit elements, and the rules for composition of elements. An element is the smallest unit of concern for SGML. It represents a single concept or logical unit. So for example, a paragraph might be one element in the **DTD**, and one paragraph element might be delimited:

```
<p> This is my paragraph. </p>
```

This forms one instance of the paragraph element. The starting, * *, and ending, * *, tags delimit the element. The "p" in the tags is called a generic identifier, and it serves to distinguish paragraph elements from other sorts of elements.

Generic identifiers, **GI**s, can have attributes. These attributes describe important characteristics of the **GI**. The attributes are specified within the starting tag of the element. For example:

```
<category lib=aips sect="math">
```

Here the attributes are *lib* and *sect*. The value for *lib* is "aips" and the value for *sect* is "math". Either single quotes, " ' " or double quotes, " " ", may be used to delimit literal attribute values. If the attribute value is one word, no quotes are necessary. The attribute name is typically limited to eight characters.

Entities In addition to tags, SGML also allow for the definition of entities. Entities provide a means for simple keyword expansion. This is useful not only for a shorthand notation, but also as a way of parameterizing a document to make future changes easy. Entity references begin with an ampersand and end with a semicolon. So for example, a reference entity for the less than symbol, "<", might be *<*. All entities are typically defined in the **DTD**.

Entities are particularly useful to prevent text from being interpreted as markup. The characters which are of particular concern are "<" and "&" because the introduce element references and entities references respectively. However, this should typically not be a problem because misinterpretation is only possible when these opening characters are followed directly by a

non-space character. So "<*test*" could be interpreted as a tag but "<*@: test*" could not. Other characters which could cause problems are ">" and ";" because these are the characters which end element references and entity references respectively.

1.2.2 Body

Once the **DTD** has been specified, the user can then structure the information in the body of the SGML document. Once the tags are specified they can be used like the commands in other markup languages. If the elements are composed incorrectly, a parser will point out the problems. This is the level at which most SGML users will operate. This is the level at which developers entering comments will operate. For the developer entering comments, there will not be a "body" as such. The comments will each be a portion of the "body" of the SGML document. These pieces will be ordered and assembled into the body of the document by an extractor.

2 Document Generation

The conflicts and complications between the language in which the source code is expressed and the language in which the comments are expressed must somehow be resolved. The information contained in these two languages must be used to generate a reference document. These issues are discussed in this section along with the tags which are used to structure the comments.

2.1 Mixing Languages

The introduction of a structuring language for comments implies a mixing at some level of the language being documented and the language in which the documentation is expressed. This mixing can happen in one of three ways:

1. Provide a system which can integrate the languages at a level higher than the file level. In this way, the system would provide mechanism to edit code where the code would be brought up in the programmers favorite editor, but the documentation would be brought up in a SGML editor. This gives the best of both worlds. A text editor is used for code, but a more powerful WYSIWYG editor can be used to construct the documentation.

2. Provide a system where the code is described as part of the documentation system. In this system, the code is extracted from the documentation for compilation purposes. This is the literate programming environment envisioned by Knuth.

3. Provide a system where the structure of the documentation is described inside of comments in the source code, and then the whole thing is massaged into a useful shape by a processor which turns the code/comment combination into a nice document.

Of these three alternatives I think the first is the cleanest, and it could be the most programmer friendly. However, it also involves the most work, especially if it is to be truly programmer friendly. It involves much more work than the latter two options. In addition, it would require a great deal of trial and error to arrive at a system which was flexible enough to be used for serious development. The second choice seems nice in theory, but I fear that this "literary" nature of development is quite foreign to many developers. The extra processing of turning documentation into code could also add too much extra time in the edit–compile–debug loop for it to be useful in practice. Thus, the third alternative was chosen. It has the advantage that the source is always kept in a state which can be compiled.

Reference documents will be assembled out of the documentation in the comments and the information extracted directly from the source code. This mixing of source code objects and comment objects in the same place requires a processor to mesh the two languages and rearrange the derived SGML objects into a coherent reference document.

2.2 Language and Comment Elements

The code units in a source file will be called "language elements". This includes for example a class definition, a function definition, a definition of one or more variables, a class declaration, a function declaration, etc. Basically, a language element is any complete statement in the grammar for the language. These are the pieces of the language which can have comment elements associated with them. Likewise, the comment elements are the comments which must *precede* the language element with which they are associated. So, for example, the following code fragment shows a function definition along with its associated comment.

```
// <div>
// <p> This function computes the area of a triangle and returns the
```

```
// result.
// </div>
float triangle::area() \{
  return (0.5 * base() * height());
\};
```

In this example, the code element is the definition of the function *triangle::area()*. The comment element preceding the definition is associated with the function. This pair will be converted into one piece of the documentation generated by the documentation extractor.

While comment elements can be provided for any C++ language elements or preprocessor elements, all comments will not be extracted. Initially, comments will only be extracted for functions and classes. Later elements like "*#define*"s, structs, enums, etc. will be added.

2.3 SGML Tags for Documentation

This section discusses the tags which were developed for describing the logical structure of elements of comments. These are described as an SGML specification in appendix A.

2.3.1 DTD Elements for Source Code Documentation

Title – *title* Many documentation elements contain an optional <*title*> as their first component. This allows for the specification of a label for that section of the overall document. So for example, one might have:

```
<div> <title> A simple section </title>
<p>The contents of the section.
</div>
```

The elements which can have titles are <*div*>, <*warn*>, <*note*>, <*verbatim*>, <*literal*>, <*code*>, <*enum*>, and <*list*>. These are explained below.

Text Separators – *div*, *p* There are two basic levels of division where "plain" text is concerned. These divisions are at the paragraph and section level.

To divide paragraphs the <*p*> tags should be used. These tags mark off a section of text indicating that it makes up a paragraph. The paragraph contains parsable character data, "#PCDATA". This means that the data in the paragraph can be fully processed by SGML for the expansion of entities

and elements. Both starting and ending paragraph tags are optional in the cases where no ambiguity exists.

As a short-cut, paragraph beginnings and ends can be implied by blank lines. So for example, the following comment:

```
//
// This function computes the volume of a cylinder
//
// Its OK
//
float Cylinder::volume() {
  return C::pi * radius() * radius() * height();
}
```

would be converted into:

```
<div>
<p>
This function computes the volume of a cylinder
</p><p>
Its OK
</p>
</div>
```

The <*div*>s are added automatically by the documentation extractor (discussed below). and the paragraph elements, <*p*>, are implicit. Note that the leading and trailing blank lines are significant. They are the short-hand notation to start the initial paragraph and end the last paragraph.

The <*div*> tag is the generic tag for sections within a document. These sections are designed to be referentially transparent and readily relocatable. Typically comment elements will be divisions. Both the starting and ending tags are required for divisions. To obtain subsections, <*div*>*s* can be nested. For example,

```
<div>
  <div> <title> Nested Division </title>
     <p> This is the first simple paragraph.
     <p> This is the second simple paragraph.
   </div>
</div>
```

In this case, the closing paragraph tags can be left off because a paragraph cannot contain another paragraph. Typically the closing paragraph tag can be left off.

Warnings and Notations – *warn*, *note* These sections of text are for the display of information which should be important to the user. In the case of <*note*>, this should be used to bring a portion of text to the attention of the user. The <*warn*> text should tell the user information which can result in damage (of some sort) if the information is not known. For example,

```
<warn> ... deletion of this object will result in a memory leak under
       these circumstances.
<note> The new operator is private to prevent dynamic creation of
       objects of this class.
```

Literal Sections – *verbatim*, *literal*, *code* Literal sections allow one to create sections which undergo very little modification in the process of generating output. There is one subtle distinction between these sections.

In the case of <*verbatim*> and <*code*>, very little processing is done. The characters between the starting and ending tags for these elements are simply treated as a character string; no elements are processed and no entities are expanded. The one difference between these two is that <*code*> should be used to display sections of source code, otherwise <*verbatim*> should be used. These are to be used when **no** expansion of entities is required.

In the case of <*literal*>, minimal processing is also done on the characters in a <*literal*> section. In this case, however, the entities are expanded correctly.

Lists – *enum*, *list* All of the lists that are used have a common tag for the elements of the list, <*item*>. The contents of this tag can only be parsable character data and the list items do not nest. So for the most part, the closing <*item*> tag can typically be omitted.

There are two "generic" list choices. The first type of generic list is a numbered list, <*enum*>. The second generic list type is a list which distinguishes the elements of the list with bullets, <*list*>. Both of these lists can have a <*title*> before any of the elements of the list. So these might look like:

```
<list>
  <item> First Element
  <item> Second Element
</list>
<enum> <title> An empty list </title>
```

```
  <item> first element
</enum>
```

Hyperlinks *To be added later (probably based on* HTML *or HyTime) ...*

Descriptors There are several elements which are devoted to describing details about a given class or function. These descriptors list things like the exceptions thrown, the I/O devices accessed, etc.

Class Descriptors

The <*descriptor*> element list several important aspects about a given class. A given <*descriptor*> might look like:

```
<descriptor>
  <execution> <sequential>
  <bounded>
  <memory> <counted>
  <iterator>
  <cached>
</descriptor>
```

So, this description would describe a class that is purely sequential, has a maximum object size, is reference counted, has an iterator class, and has a builtin caching mechanism. If <*bounded*> were not specified, the assumption is that the object size is unbounded.

This <*descriptor*> list corresponds to the object descriptors required in the AIPS++ coding standards. The options are as follows:

- <*execution*> can contain <*sequential*>, <*guarded*>, <*concurrent*>, or <*multiple*>.
- If <*bounded*> is specified then the object size growth is bounded, otherwise it is unbounded.
- <*memory*> can contain <*counted*>, <*gc*>, or <*unmanaged*>
- If <*iterator*> is specified, then the object has an iterator, otherwise it does not.
- If <*persistent*> is specified, then the object is persistent, otherwise it is not.

- If <*cached*> is specified then the object has cached management, otherwise it does not.

Using this list of attributes a great deal of information about the class can be expressed succinctly.

Device I/O

Often, it is useful to have the ability to track functions which depend on particular files or I/O devices. This will often target which functions have direct access to the operating system. So the following would label a function which performed I/O on the files `/etc/hosts` and `/etc/motd` using OS specific routines, e.g. *"<stdio.h>"*, *"<iostream.h>"*:

```
<iodev> <level><os>
   <item> /etc/hosts
   <item> /etc/motd
</iodev>
```

It is also useful to be able to target functions which access files using a particular abstraction, e.g. **AipsIO**, because although these files do not depend directly on the operating system function calls they are tied to particular files. So for example a function that performs operations on the file `/usr/local/var/aipsppdb` using **AipsIO** would be labeled:

```
<iodev> <level> AipsIO
   <item> /usr/local/var/aipsppdb
</iodev>
```

In this way, the dependencies between the AIPS++ and the environment in which it operates can be tracked.

Exceptional Conditions

It is important to know the exceptions which can be thrown by a given class or function. This information gives the user of a function the list of exceptions which he may be required to catch. This information can be presented as follows:

```
<thrown>
  <item> AllocError
  <item> ArrayNDimError
</thrown>
```

These thrown exceptions should be specified at the member functions which throw the exceptions. However, the <*thrown*> descriptor could also be used in the comment element for a class. At some time in the future, the extractor will hopefully be able to generate call trees, and thus pick up a complete list of all of the exceptions thrown by a given function automatically. However, this ability depends on having a fully functional C++ parser available.

2.4 Extractor Commands

There are extractor commands which provide the user with control over how and if the comments are extracted. All of these commands have the form //* where the "//" is the beginning of a C++ comment, and the "*" is one or more consecutive non-space command characters.

2.4.1 Non-processable Comment

Often one will want to specify a comment which only belongs in the source file, and should not be extracted. This can be accomplished as follows:

```
//\#   This comment will not be extracted
//\#Neither will this one
```

2.4.2 Group Command

It is useful to provide a comment and specify that this comment applies to a group of language elements. This prevents the comment inconsistencies which can result from duplication of comments. This can be achieved as follows:

```
// Generally use of this should be shunned, except to use a FORTRAN
// routine or something similar. Because you can't know the state of
// the underlying implementation.
//+grp
T *getStorage(Bool \&deleteIt);
const T *getStorage(Bool \&deleteIt) const;
void putStorage(T *storage, Bool deleteAndCopy);

// <warn> An added problem with freeStorage is that it ...
void freeStorage(const T *storage, Bool deleteIt) const;
//-grp
```

The *"//+grp"* is the start group command. This command tells the extractor that the comment it just encountered should be used as the "base" comment for languages elements until a end group command, *//-grp*, is encountered. Any comments that are introduced within the group will be added to the end of the base comment for the appropriate language element(s). Group commands can be nested as long as the start and end group commands are balanced.

2.4.3 Literal Command

All comment elements, the comment preceding a language element, are typically a division, <*div*>. However, inside source files this results in a great deal of clutter in comments which would otherwise be quite readable. As a result, the extractor will attempt to correctly add the <*div*> tags. The user, however, can prevent the extractor for adding these by using the literal extractor command. The extractor will *not* try to spruce up documentation between the start literal command, *"//+lit"*, and the end literal command, *"//-lit"*. The start literal command should occur at the beginning of the comment block, and the end literal command should occur at the end. The literal command does not extend over multiple comment elements. The following example demonstrates the use of these commands.

```
//+lit
// <category lib=aips sect=io>
// <div> <title> Problems with this implementation </title>
//   The body of the section.
// </div>
//-lit
```

These commands prevent the extractor from adding any *user level* SGML commands. Typically all that the extractor would add is <*div*>s around a comment element. This allows the user to specify:

```
//
// This computes the area of the circle.
//
float circle::area() {
  return(C::pi * radius() * radius());}
```

instead of:

```
// <div>
```

```
// This computes the area of the circle.
// </div>
float circle::area() {
  return(C::pi * radius() * radius());}
```

A Document Type Definition

This appendix has the document type definition for the comment markup language. It is written in SGML syntax. The *DTD* defines "elements" and "entities". The elements are the lowest level of granularity with which SGML deals. The entities function much like macro; entity references are replaced by the "body" of the entity.

There are two basic kinds of entities, general and parameter. The primary reason a distinction is drawn between these two types of entities is so that the general entity name space is unique from the parameter entity name space. Parameter entities are typically used by the designer of the *DTD*. As a result, parameter entities are typically used within other entity or element definitions. This leaves the person writing a particular document free to use any names when defining general entities.

Entity definitions begin with "*<!entity*". So the definition of a general entity might look like:

```
<!entity dquot sdata '\{\tt "\}' >
```

This defines a general entity which can be used to insert "double quotes" into a document. This is done by placing a reference to the entity in the document, i.e. "*&dquot;*". Here the "&" introduces the entity reference and the ";" ends the entity reference. This double quote entity is defined in terms of the formatting language into which the SGML document will be translated, in this case LaTeX. So these, entities must be changed for each SGML processor. For a plain ASCII system, the entity might look like:

```
<!entity dquot sdata '"' >
```

Parameter entities are defined and used in much the same way as general entities. Except that parameter entities are typically used by the *DTD* designer. The following is an example of a parameter entity definition:

```
<!entity \% bridgecontent "(p$|$warn$|$note$|$code$|$enum$|$list)" >
```

This defines a entity *bridgecontent*. The *%* indicates that this is a parameter entity instead of a general entity. This parameter entity is later used to define the elements that a *<bridge>* element can contain:

```
<!element bridge - o ((\%bridgecontent)*) >
```

The *%bridgecontent* entity reference is replace by the body of the entity, *(p|warn|note |code|enum|list)*. Element definitions are introduced with the" *<!element*", as shown above. The element definition has four fields, the name e.g. "bridge", a pair of fields which indicate if the opening tag and closing tag are optional or required — an "o" indicates that it is optional and an "-" indicates that it is required —, and a field which indicates the contents of the element. The syntax of the contents field is similar to lex:

- "*" – indicates zero or more occurrences
- "+" – indicates one or more occurrences
- "?" – indicates zero or one occurrence
- "," – indicates a sequence, e.g. (foo,bar) is <*foo*> followed by <*bar*>
- "|" – indicates an "or", e.g. (foo|bar) means one of <*foo*> or <*bar*> must occur
- "&" – indicates an "and", e.g. (foo&bar) means <*foo*> and <*bar*> must occur, but in any order
- "()" – parentheses provide grouping

So, the <*bridge*> example above declares a <*bridge*> element which can in turn contain zero or more <*p*>, <*warn*>, <*note*>, <*code*>, <*enum*>, or <*list*> elements. The opening tag is required, but the closing tag is optional. Elements are the most basic information units handled by SGML. They serve a purpose very similar to productions in a grammar.

In addition to elements and entities, there are several "special" characters which are useful to know:

- *&#TAB;* – Horizontal tab
- *&#RE;* – Record end
- *&#RS;* – Record start
- *&#RS;B* – Record start followed by one or more spaces and/or tabs
- *&#RS;&#RE;* – Empty record
- *&#RS;B&#RE;* – Record containing only one or more spaces and/or tabs

- *B&#RE;* – One or more trailing spaces and/or tabs
- *BB* – Two or more spaces and/or tabs.

These "invisible" characters are most useful when trying to define keyboard shortcuts which prevent document preparers from having to enter all of the necessary tags. For instance, inserting paragraph tags can be made easer as follows:

```
<!entity psplit '</p><p>' >
<!shortref pmap "\&\#RS;B\&\#RE;" psplit
                "\&\#RS;\&\#RE;" psplit >
```

This allows one to insert paragraph breaks by entering a blank line, i.e. a blank line or an empty line.

A.1 Parameter Entities

A.1.1 Sections

- <!entity % divcontent @w"(p|warn|note|verbatim|literal| code|enum|list|descriptor|iodev|thrown)'
- <!entity % bridgecontent "(p|warn|note|code|enum|list)" >

A.1.2 Paragraphs

- <!entity % pcontent "#PCDATA" >
- <!entity ptag '<p>' >
- <!entity psplit '</p><p>' >
- <!shortref pmap "&#RS;B&#RE;" psplit "&#RS;&#RE;" psplit >
- <!usemap pmap p>

A.2 Elements

A.2.1 Sections

- <!element div - - (title?, divbody) >
- <!element divbody o o ((%divcontent)* | divset) >
- <!element divset o o ((div,bridge?)*, div) >
- <!element bridge - o ((%bridgecontent)*) >

A.2.2 Paragraphs

- <!element p o o (%pcontent) * >

A.2.3 Warnings and Notations

- <!element warn - o (title?, #PCDATA) >
- <!element note - o (title?, #PCDATA) >

A.2.4 Literal Sections

- <!element verbatim - - (title?, (#CDATA)) >
- <!element literal - - (title?, (#RCDATA)) >
- <!element code - - (title?, (#CDATA)) >

A.2.5 Lists

- <!element enum - - (title?, (item*)) >
- <!element list - - (title?, (item*)) >
- <!element item - o ((p*)|div) >

A.2.6 Titles

- <!element title o o (#PCDATA) >

A.2.7 Class Categories

- <!element category - o (#EMPTY) >
- <!attlist category lib CDATA "misc">
- <!attlist category sect CDATA "general">

A.2.8 Descriptors

- <!element descriptor - - ((execution|bounded|memory|iterator|persistent)*) >
- <!element execution - o (sequential|guarded|concurrent|multiple) >
- <!element bounded - o EMPTY >

- <!element memory - o (counted|gc|unmanaged) >
- <!element iterator - o EMPTY >
- <!element persistent - o EMPTY >
- <!element cached - o EMPTY >
- <!element concurrent - o EMPTY >
- <!element memory - o EMPTY >
- <!element manage - o EMPTY >
- <!element iter - o EMPTY >
- <!element persist - o EMPTY >
- <!element counted - o EMPTY >
- <!element gc - o EMPTY >
- <!element unmanaged - o EMPTY >

A.2.9 Device I/O

- <!element iodev - - (level?,item*) >
- <!element level - o (os|(#PCDATA)) >
- <!element os - o EMPTY >

A.2.10 Exceptions

- <!element thrown - - (item*) >

A.3 General Entities (for LaTeX)

A.3.1 Required Because of Tags

- <!entity lt sdata "{\lt}" >
- <!entity amp sdata "\&" >

A.3.2 Typically Unnecessary

- <!entity dquot sdata '{\tt "}' >
- <!entity num sdata "\#" >
- <!entity gt sdata "{\gt}" >
- <!entity percnt sdata "\
- <!entity lpar sdata "(" >
- <!entity rpar sdata ")" >
- <!entity ast sdata "\mch{\ast}" >
- <!entity plus sdata "+" >
- <!entity comma sdata "," >
- <!entity hyphen sdata "-" >
- <!entity colon sdata ":" >
- <!entity semi sdata ";" >
- <!entity equals sdata "=" >
- <!entity commat sdata "@" >
- <!entity lsqb sdata "[" >
- <!entity rsqb sdata "]" >
- <!entity circ sdata "\verb+$\hat{}$" >
- <!entity lowbar sdata "_" >
- <!entity lcub sdata "\{" >
- <!entity rcub sdata "\}" >
- <!entity verbar sdata "{\verbar}" >
- <!entity tilde sdata "\verb+ +" >
- <!entity mdash sdata "{-{}-{}-{}}" >
- <!entity ndash sdata "{-{}-{}}" >
- <!entity hellip sdata "{\ldots}" >

B Example Header File

```
\#if !defined(AIPS\_COUNTEDPTR\_H)
\#define AIPS\_COUNTEDPTR\_H
//\# Copyright (C) 1992,1996
//\# Associated Universities, Inc. Washington DC, USA.
//\#
//\# This program is free software; you can redistribute it and/or modify
//\# it under the terms of the GNU General Public License as published by
//\# the Free Software Foundation; either version 2 of the License, or
//\# (at your option) any later version.
//\#
//\# This program is distributed in the hope that it will be useful,
//\# but WITHOUT ANY WARRANTY; without even the implied warranty of
//\# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
//\# GNU General Public License for more details.
//\#
//\# You should have received a copy of the GNU General Public License
//\# along with this program; if not, write to the Free Software
//\# Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
//\#
//\# The AIPS++ consortium may be reached by email at aips2-request@nrao.edu.
//\# The postal address is: AIPS++ Consortium, c/o NRAO, 520 Edgemont Rd.,
//\# Charlottesville, Va. 22903-2475 USA.

template<class t> class CountedPtr;
template<class t> class CountedConstPtr;

//
// This is a dummy class which is not part of "aips/CountedPtr.h". I've
// included it for the purpose of demonstration. The things to note in
// the comments in general
// are:
//  <list>
//    <item> The opening blank comment line is significant. It is the
//           marker for the first paragraph.
//    <item> Blank lines between paragraphs are significant. They
//           separate paragraphs.
//    <item> The blank comment line at the end of the comment is
//           significant. It closes the last paragraph.
```

```
//  </list>
//
// In addition, although typically the divisions, "div", are not
// specified for a comment they are generated before and after the comment.
//
template<class t> class  generic\_useless\_template \{
private:
  t *val;
public:
  //
  // Generic ctor comment.
  //
  generic\_useless\_template(t *nv) : val(nv) \{\}
  //
  // This is one of the indirection operators. It is here to demonstrate
  // the use of the "+grp" extractor command.
  //+grp
  t \&operator*() \{ return *val;\}
  t *operator->() \{return val;\}
  //-grp

  //
  // Just another function.
  //
  void set(t *n) \{
    if (val) delete val;
    val = n;
  \}
  //
  // Generic dtor comment.
  //
  ~generic\_useless\_template() \{ delete val;\}
\};


//
// <category lib=aips sect=memory>
//
// This class stores the reference count and the pointer to the
// "real data".
```

```
//
// It is currently a template and is used such that
// "t" is the "true" type of the stored pointer. This means, however,
// that when it is used a template instantiation must be done for
// each type which "t" assumes. This makes debugging easier, but
// in the future all of these pointers could be declared with void
// type to avoid template instantiations.
//
template<class t> class PtrRep \{
friend class CountedPtr<t>;
friend class CountedConstPtr<t>;
private:
  t *val;
  unsigned int count;
protected:
  //
  // This constructor sets up the reference count to "1" and
  // initializes the pointer to the "real" data.
  //
  PtrRep(t *v) : val(v), count(1) \{\}
  //
  // <warn>
  // This destructor, if called, deletes the "true" data.
  //
  ~PtrRep() \{delete val;\}
\};

//
// <category lib=aips sect=memory>
//
// This class maintains a count of pointers which point to particular
// data, and it deletes the data only when no other counted pointers
// are pointing at it.
//
// This class is used as a pointer to constant data. As such, it
// only has the subset of the "CountedPtr" functions which are relevant
// for constant data.
//
template<class t> class CountedConstPtr \{
protected:
```

```
  PtrRep<t> *ref;
public:
  //
  // After the counted pointer is initialized the value should no
  // longer be manipulated by the raw pointer of type "t*".
  //
  CountedConstPtr(t *val) : ref(new PtrRep<t>(val)) \{\}
  //
  // After the counted pointer is initialized the value should no
  // longer be manipulated by the raw pointer of type "t*".
  //
  CountedConstPtr(const CountedConstPtr<t> \&val) : ref(val.ref) \{
    if (ref)
      (*ref).count++;
  \}
  //
  // The destructor deletes the data only when there are no other
  // counted pointers pointing at it.
  //
  ~CountedConstPtr() \{
    if (ref) \{
      (*ref).count--;
      if ((*ref).count == 0)
delete ref;
    \}
  \}
  //
  // The CountedConstPtr indirection operator simply returns a
  // reference to the value being protected.
  //
  // <note> The address of the reference returned should not be
  // stored for later use.
  //
  const t \&operator*() const \{
    return(*(*ref).val);
  \}
  //
  // This dereferencing operator behaves as expected; it returns the
  // pointer to the value being protected, and then its dereferencing
  // operator will be invoked as appropriate.
```

```
  //
  const t *operator->() const \{
    return ((*ref).val);
  \}
\};

//
// <category lib=aips sect=memory>
//
// This class maintains a count of pointers which point to particular
// data, and it deletes the data only when no other counted pointers
// are pointing at it.
//
// This class is used as a pointer to non-constant data, and it has
// all of the possible manipulation functions.
//
// <note> It is possible that use the <code>replace</code> function
// can change the data to which a <code>CountedConstPtr</code> points.
//
template<class t> class CountedPtr : public CountedConstPtr<t> \{
public:
  //
  // Simple function which leaves the work to the parent constructor.
  //
  CountedPtr(t *val) : CountedConstPtr<t>(val) \{\}
  //
  // Simple function which leaves the work to the parent constructor.
  //
  CountedPtr(const CountedPtr<t> \&val) : CountedConstPtr<t>(val) \{\}
  //
  // This assignment operator allows CountedPtrs to be freely assigned to
  // each other.
  //
  CountedPtr<t> \&operator=(CountedPtr<t> \&val) \{
    if (ref) \{
      (*ref).count--;
      if ((*ref).count == 0)
delete ref;
    \}
    ref = val.ref;
```

```
    if (ref)
      (*ref).count++;
    return *this;
  \}
  //
  // This function changes the value for this CountedPtr and all of
  // the other CountedPtrs which pointed to this same value.
  //
  // <note> This can change the value to which a CountedConstPtr points.
  //
  void replace(t *v) \{
    if (ref) \{
      if ((*ref).val)
delete (*ref).val;
      (*ref).val = v;
    \}
  \}
  //
  // This assignment operator allows the object to which the current
  // CountedPtr points to be changes.
  //
  CountedPtr<t> \&operator=(t *v) \{
    if (ref) \{
      (*ref).count--;
      if ((*ref).count == 0)
delete ref;
    \}
    ref = new PtrRep<t>(v);
    return *this;
  \}
  //
  // The CountedConstPtr indirection operator simply returns a
  // reference to the value being protected.
  //
  // <note> The address of the reference returned should not be
  // stored for later use.
  //
  t \&operator*() \{
    return(*(*ref).val);
  \}
```

```
  //
  // This dereferencing operator behaves as expected; it returns the
  // pointer to the value being protected, and then its dereferencing
  // operator will be invoked as appropriate.
  //
  t *operator->() \{
    return ((*ref).val);
  \}
\};

\#endif
```