How To .* for AIPS++ Programmers

W Young (Ed.)

October 15, 2006

Contents

Copyright copyright 1996,1998,1999 Associated Universities Inc., Washington, D.C.

AIPS++ How To .* for AIPS++ Programmers.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the AIPS++ Consortium.

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.

Part I

General Help

Chapter 1

Where to Look

You are in a maze of twisty passages — all alike. This famous line from the game "Adventure" is also a common reaction to people who attempt to find things, documentation especially, in the AIPS++ system. In the belief that the best way to learn is to do, we've assembled this "How To .*" document to help you (and us too).

1.1 Who Can Help Me?

If you just plain get stuck, don't despair. There are a lot of knowledgable, friendly people in the AIPS++ project. If you cannot find the help locally, one of the following people will help your or direct you to someone who can help you:

Tim Cornwell(tcornwel@nrao.edu) Astronomical analysis and design, questions about AIPS++.

Athol Kemball (akemball@nrao.edu) Class libraries and coding, Synthesis development.

Darrell Schiebel (drs@nrao.edu) Glish programming, System installation and utilities.

Wes Young (wyoung@nrao.edu) Documentation.

1.2 Directories

There are several major directory areas you should understand if you wish to program or write documentation for AIPS++. You may wish to refer to the AIPS++ System Manual for further details.

1. There are the *master* source directories, which contain the sources of code and documentation [1]. Since these are *master* sources, you cannot directly modify them, rather you must modify them by checking them into your *personal* (see below) directory, modify them, and then check them back.

2. You modify source (program or documentation) in your *personal* directories. These directories shadow the *master* directories. Because new directories are created or deleted on occasion, you will sometimes need to run the `mktree` program to reflect any directory additions or deletions in the *master* directories.

3. *Processed documentation* directories. Documentation which has been processed (normally by being turned into PostScript and HTML) are placed into common directory tree. During development of the documents you will be working in your *personal* directories.

4. *Binary* directories. These are where common (shared among all programmers) programs and libraries go. These are architecture dependent. During class and program development you will be working in your *personal* directories.

The exact location of these directories will depend on how your local system was set up, but will be of the following form:

master `/aips++/code`

personal `/aips++/code`

processed documentation `/aips++/docs`

binary `/aips++/sun4sol_gnu` or whatever architecture(s) you have at your installation.

Sometimes instead of `/aips++` your common directories will be under `/aips2/aips++` or `/u/aips++` or some other location. You will need to ask locally.

Similarly, your personal directory need not be directly under your home directory — you may put it anywhere you have Unix "write" permission.

[1] For historical reasons these are all referred to as *code*, although *source* would be a more accurate name

The directory location of a source file is often written as something like `code/doc/reference/Coding.latex`. If you are working on it in your `personal` directory you would prepend this path with `/aips++` and if you wanted to look at the master version you would prepend the path with `/aips++` (or wherever your common installation is).

1.3 Documentation

There are several types of documentation in the AIPS++ system.

Memos and Notes Various pieces of documentation for AIPS++. The distinction is that *Memos* are meant for external distribution and *Notes* are more internal. These are under `code/doc/memos` and `code/doc/notes` respectively.

Specifications User requirements and specifications for AIPS++. These are under `code/doc/specs`.

Papers Various papers that have been written (*e.g.*, for conference proceedings) are available under `code/doc/papers`.

Reference Several reference documents exists

- **The AIPS++ Programming Manual** is found in `code/doc/reference/Coding.latex`.
- **AIPS++ System Manual** is found in `code/doc/reference/System.latex`.
- **How to .* for AIPS++ Programmers** (this document) is found in `code/doc/reference/HowTos.latex`

1.3.1 Multi-file Documents

A simple convention is used to locate and place the individual files used in a multi-source document. Basically, the "included" files are placed in a subdirectory with a `.dir` extension, however the including file is kept at the outer level. This scheme has the advantage so that if a single-source file later becomes split up, the "including" file itself does not move so you always find it in the same place.

So, as an example, we have a **A Guide to Synthesis Processing in AIPS++** under the `code/doc/user` directory. We call it `synthesis.latex`. At a later point we might want to split this document into multiple `.tex` files for convenience, or we might have some `.ps` files. In this code we would make a directory `code/doc/reference/synthisis.dir` and place those files within it, however synthisis.latex would remain in `code/doc/user`.

1.3.2 Processed Documentation

Processed source files (PostScript or plain text, depending upon the source document) are all available under `/aips++/docs` (assuming your common aips++ area is /aips++). The directory structure of processed documents follows that of the source documents. So, for example, the postscript version of the "Coding" volume (**The AIPS++ Programming Manual**) would be `/aips++/docs/reference/Coding.ps.gz`.

1.3.3 Online Documentation

All of AIPS++ documentation is available via the WWW. Some is in HTML, some in postscript, and some both in postscript and HTML.

For an AIPS++ note or memo to appear in the AIPS++ web, the documenter must add a pointer to the document in either the `code/doc/notes/notes.dir/index.tex` or `code/doc/memos/memos/dir/index.tex` files. To generate HTML from source, the documenter must add the document root name to the makefile file with the desired `latex2html` options (typically, `-split 1`).

Chapter 2

Getting Help

The AIPS++ team has developed many useful tools, classes, and programming techniques. These series of "How To's" will hopefully get you started on the road through the *twisty little passages*.

2.1 Where to get Help

Help today is "Hypertext". There are many avenues to get to the help you need. For programmers, you might want to start with the "programmer documentation" http://aips2.nrao.edu/docs/programmer/programmer.html. This web page has many links to useful documentation. Users should check out **Getting Started in AIPS++**. If you're primarily interested in Glish Programming you will want to take a look at the **The Glish User's Manual** and Rick Fisher's **Guide to Getting Started with Glish**.

Most documents have postscript version which you may print out. There are a few "web only" which will require you to print the postscript from your web browser.

You may also search the AIPS++ documentation tree from the URL http://aips2.nrao.edu/aips++/doc/search/search.html

2.2 What Kind of Help

We think of three types of programmers who will program in AIPS++, the library builder, the applications, and the glish programmer. All programmers need to pay particular attention to the coding style, documentation requirements, and proper code checking.

If you're a glish programmer you probably want to look at the glish tutorial written by Rick Fisher or the **Glish user's manual**.

If you're writting library tools, look at the modules inside the aips package.

If you're an applications programmer, you may want to start with **AIPS++ Programmer Reference** or from the "On-line programmer documentation", http://aips2.nrao.edu/docs/reference/online/online.latex web page.

Chapter 3

Getting Started

3.1 Setting up your account to use aips++

3.1.1 aips++ user setup

The AIPS++ environment is defined via a once-only modification of your shell startup script.

Assuming that AIPS++ has been installed under */aips++*, users of Bourne-like shells (*sh*, *ksh*, *bash*) must add the following line to their *.profile* file at a point *after* `PATH` (and `MANPATH`) are defined:

```
[ -f /aips++/aipsinit.sh ] && . /aips++/aipsinit.sh
```

The equivalent entry in *.login* for C-like shells (*csh*, *tcsh*) is:

```
if (-f /aips++/aipsinit.csh) source /aips++/aipsinit.csh
```

The *aipsinit* scripts define a single environment variable called `AIPSPATH`, and prepend the AIPS++ *bin* area to the `PATH` environment variable, and the AIPS++ *doc* directory to the `MANPATH` environment variable (There are no UNIX style man pages available, AIPS++ documentation is available on-line via the world-wide-web).

For more detailed information, see aipsinit(see AIPS++ System Manual, Section **??**for more details) including an explanation of `AIPSPATH` and a mechanism for controlling the point where the AIPS++ *bin* areas are added to `PATH`.

3.1.2 AIPS++ programmer setup

AIPS++ Environment Variables Normally, AIPS++ programmers must belong to the AIPS++ programmer group (in the unix sense) in order to have permission to write to the AIPS++ source directories. The conventional name for this group is `aips2prg`, but it may be set up differently at your site. Consult your local AIPS++ manager.

AIPS++ programmers must invoke the *aipsinit* scripts from their *.profile* or *.login* files as do users. However, programming is normally done on the `gnu` version of AIPS++. The required incantation of *aipsinit* for Bourne-like shells is

```
if [ -f /aips++/aipsinit.sh ]
then
   aips_ext=gnu
   . /aips++/aipsinit.sh
fi
```

and for C-like shells

```
if (-f /aips++/aipsinit.csh) then
   set aips_ext = gnu
   source /aips++/aipsinit.csh
endif
```

Note that the *aipsinit* scripts explicitly unsets `aips_ext` In Socorro, the default aips_ext is gnu, so there is no need to set it.

AIPS++ Directory Tree Programmers also need to create a shadow copy of the AIPS++ `code` directory tree to serve as their AIPS++ workspace. The *mktree* utility does this automatically:

```
mkdir $HOME/aips++
cd $HOME/aips++
mktree
```

Apart from creating a shadow copy of the AIPS++ *code* directory tree, *mktree* creates symbolic links into the local AIPS++ *rcs* directory tree thereby linking the programmer's workspace directly to the local copy of the AIPS++ RCS repositories. However, do not use *ci* or *co* to check code in or out, see Checkout (see AIPS++ System Manual, Section 3.2 for more details) for instructions on how to do that.

mktree works incrementally so that if any workspace directories or RCS symbolic links are accidently deleted, or if new AIPS++ directories are created, *mktree* will recreate only what is necessary. For a more detailed description, see mktree (see AIPS++ System Manual, Section **??** for more details)

3.2 Getting started on aips++ programming

3.2.1 Checking source code in and out

Code Management Background The first thing to realize about AIPS++ code management is that, on account of the distributed nature of AIPS++ code development, a master copy of the AIPS++ sources resides in Socorro, and some consortium sites have slave copies which are updated automatically at least once a day by the AIPS++ code distribution system (if all goes well that is - and it usually does).

RCS is used for source code management in AIPS++, refer to the unix manual page for *rcsintro* for more information. On both master and slave systems the AIPS++ sources reside within dual directory trees which exist under the AIPS++ root directory. One of these, *aips++/rcs*, contains the RCS source code repositories, and the other, *aips++/code*, contains plain-text sources. Symbolic links named `RCS` exist in each subdirectory of *aips++/code* and point into the corresponding subdirectory of *aips++/rcs*, thereby tying the two directory trees together for checkout of sources. The code distribution system updates the *rcs* tree, and the *code* tree is then updated from it. The sources under *aips++/code* are used to build the AIPS++ system locally, including object libraries, executables, documentation, system databases and more.

When it comes to modifying or adding new AIPS++ sources, the RCS checkin and checkout operations must be performed directly on the master RCS repositories in Socorro. A valid but tedious way of doing this would be to *rlogin* to Socorro, checkout the sources you're interested in, *ftp* them back to your home machine, modify them, *ftp* them back to Socorro, and then check them back in. Alternatively, AIPS++ sources can be checked in and out, updated, or deleted via the *ai*, *ao*, *au*, and *ax* utilities running on your local machine (*ax* may only be run in Socorro). These commands work directly on the master AIPS++ RCS repositories via an NFS automount running across the internet which was (or should have been) set up when AIPS++ was first installed at your site. For this reason the commands may appear to execute slowly, or in some cases you may get a message to the effect that the network is down and to try again later (you will also get this message if the NFS automount was not set up properly).

Note carefully that although the programmer workspace created by *mktree* contains *RCS* symlinks into the local RCS slave repositories, you should never use *co -l* to check code out with a lock, or *ci* to check it back in again. This would only affect the local slave repository, your changes would not be propagated back to the master in Socorro or the other consortium sites. The `RCS` symlinks in programmer workspaces are provided so that you can check out sources without a lock, or use the *rlog* or *rcsdiff* commands (provided you don't mind that the local RCS repository may be out-of-date by up to a day).

Checking out source When you check out a file using *ao* it is checked out from the master in Socorro. The *ao* script is really a front-end for the ordinary RCS *co* command but with built-in knowledge of the AIPS++ code management system. *ao* takes all the same command line options as *co*. In particular, you can use *ao -l filename.ext* to check out a file named *filename.ext*, locking others out of the capability to modify the file while you are working with it; and you can use *ao -u* to cause a checkout to be forgotten. Check the unix manual page for *co* for more information.

Checking in source The checkin script, *ai*, operates in much the same way as *ao*. It is a front-end to the RCS *ci* command and takes the same command line options. However, *ai* does more. After checking the code back into the master in Socorro, it updates the local RCS slave repository by copying the master version back, and then updates the plain-text file in the AIPS++ *code* directory. That way the local slave copy of the AIPS++ sources is updated immediately, and you don't have to wait for the automatic update delivered by the code distribution system.

When to check in new code Since AIPS++ is a production system, we are very concerned about the stability of the system. As a result, one should be very careful about checking in new features. The overall rules are:

- Maintaining the stability of daily is a high priority for everyone in the project. Breaking daily affects the work of a large number of people.
- Maintaining the stability of weekly is essential. Breaking weekly is disastrous for the work of a large number of people. Our goal should be that every weekly can be marked as stable.

- Checkins of changes to existing code should be made with the above two goals in mind.
- Changes to code that is used by others must be well-tested on the project compiler (egcs 1.1.2) before checkin. Checkin of purely developmental code is less crucial but still good habits should be followed.
- All checkins except simple ones, and bug fixes should be done early in the week. Early means with two more of your inhales to get it right before the final weekly.
- If you have a *really* substantial change, alert others in the project via your weekly report beforehand.
- All development sites should be running a daily using the project compiler.

Other code management utilities On occasion it may happen that someone from another consortium site checks in some source code and you want your local slave repository to be updated immediately. The *au* command does this for you by copying the RCS version file from the master and then updating the plain-text copy.

Deletion of sources is handled by the *ax* utility. Since all deletions must be propagated to the slave copies, and also because of certain timing questions, *ax* can only be run on the master sources in Socorro.

For a more detailed description of the various utilities discussed, see Code management (see AIPS++ System Manual, Section **??**for more details)

3.2.2 Compiling your source code

The make system System generation in AIPS++ is handled by a hierarchy of GNU makefiles. These makefiles are quite complex because of the complex job they have to perform, building AIPS++ on various platforms at any number of sites at which AIPS++ has been installed.

For a programmer, however, the main thing to recognize about the makefiles is their two-sided nature. Apart from rebuilding the AIPS++ system from installed sources, they also support programmer workspaces. Where a programmer has created new sources, or checked out and modified old ones, they must recompile them locally and leave the results in the programmer's private workspace. Furthermore, programmers are not required to check out associated sources, such as header files, just for the sake of the compilation. After searching unsuccessfully for a file in the programmers own workspace, the makefiles carry the search into the standard AIPS++ source code directories.

System-targets In accordance with the dual role of the makefiles their targets are of two kinds, system-oriented and programmer-oriented. The default target of every AIPS++ makefile, `all`, is programmer-oriented, and recompiles everything in the programmer's workspace which is out-of-date. It is usually sufficient for programmers to invoke *gmake* without arguments to recompile their sources. The corresponding system-oriented target is `allsys` and, like all system-oriented targets, this recompiles any new sources in the standard installation without any reference whatsoever to anything in the programmer's workspace.

Programmer-targets The programmer-oriented targets in the makefile for class implementation files and applications deserve special mention. They allow programmers to specify private include file directories via the `MYINCL` variable which can be defined on the command line

```
gmake MYINCL=/home/bloggs/include MyClass.o
```

Likewise, the applications makefile provides a `MYLIBS` variable which allows for private libraries. If you wanted to compile an AIPS++ application and link it to class implementation files which you had modified but not checked in, you would need to build a private library by using the programmer-oriented `mylib` target in the implementation makefile. You could then link your application to this using the `MYLIBS` variable.

The AIPS++ makefiles and the standard target names are discussed at length in makefiles (see AIPS++ System Manual, Section **??** for more details)

Chapter 4

Writing Glish Functions

Users should consult ***The Glish User's Manual*** *for general glish programming.*

Writing glish functions is very useful for assembling AIPS++tools. Utilities checked into the system should follow a set guidelines.

4.1 Guidelines

As a glish tool writer it helps to distinguish between private member functions and global functions (This will help avoid name-space confilicts with other scripts that might be included). Write private functions as a record i.e. app.function := function At the end of the script declare the record const to make sure a user doesn't accidently overwrites it.

Glish functions intended for user interaction need both programmer and user documentation. Functions not visible to users need programmer documentation similar to the C++/C function format.

Glish functions that users interact with are documented using "Help Atoms" (See Paul Shannon's note 187 for More details). Essentially you specify a series of attributes that describe the function. All global functions in the system library must have the help atoms defined! Please follow the glish script template. Until we implement code copping on AIPS++scripts we need to exercise self-discipline.

Include glish code only once. How do you do this? Put the glish code inside an "include" guard. An example follows:

```
if(!is_defined("myGlishScript_included")){
   myGlishScript_include := 'yes';
   ....
```

```
}
```

Now how do we prevent the clients starting before we really need them? Write a function similar to the start_myClient example shown below.

```
const start_myClient := function()
{
   if(!is_defined("myClient_started")){
      myClient_started := 'yes'
      global myClient := client("my");
   }
}
```

You call start_myClient in your glish script. It will only execute if the variable myClient_started has not been defined.

4.2 Conventions

Functions are define as lowercase with underscores between words.

Variables are mixed upper and lowercase with the first letter always lower case. Use descriptive variable names. It will help others understand your code.

Use comments when needed and lots of whitespace to make things readable!

4.3 Pitfalls and Troubles

- Avoid terminating glish clients.

4.4 Glish Template

Please use the following template to write your glish scrips. It's available from code/install/codedevel/template-glish (or will be shortly).

```
Text enclosed in <> needs to be changed to the appropriate function name.

------------cut here--------------------------

# <GlishFunction.g>: short description ...
```

```
# Copyright (C) 1996
# Associated Universities, Inc. Washington DC, USA.
#
# This library is free software; you can redistribute it and/or modify it
# under the terms of the GNU Library General Public License as published by
# the Free Software Foundation; either version 2 of the License, or (at your
# option) any later version.
#
# This library 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 Library General Public
# License for more details.
#
# You should have received a copy of the GNU Library General Public License
# along with this library; if not, write to the Free Software Foundation,
# Inc., 675 Massachusetts Ave, Cambridge, MA 02139, USA.
#
# Correspondence concerning AIPS++ should be addressed as follows:
#        Internet email: aips2-request@nrao.edu.
#        Postal address: AIPS++ Project Office
#                        National Radio Astronomy Observatory
#                        520 Edgemont Road
#                        Charlottesville, VA 22903-2475 USA
#

# Create an include guard

if(! is_defined("<GlishFunction_g_included>")) {
  <GlishFunction_g_included> := 'yes'

#
#
# include "other.g" files as needed
#

#
#Define glish functions (Remove const while debugging otherwise...)
#

const <GlishFunction> := GlishFunction(arg...) {
```

```
  .... }

} # Terminates the include guard
```

Chapter 5

Distributed Objects

Chapter 6

The AIPS++ Code Review Process

written as Note 170 by Paul Shannon, NRAO-NM
June 23, 1995

6.1 Introduction

The AIPS++ project adopted a formal procedure for quality checking code in the late summer of 1994. The procedure requires that all new core[1] source code be submitted to the "code cop" (*email: aips2-cop@nrao.edu*) before it can become sanctioned AIPS++ code. Once received, the cop assigns a reviewer, who judges the code against the **AIPS++ Standards and Guidelines** (note 167). A dialog between the programmer and the reviewer may follow, eventually leading to approval of the code, and its formal inclusion into AIPS++.

In this note, I discuss the review procedure

- in general
- from the perspective of the programmer submitting new code
- from the perspective of the reviewer
- with special attention to test programs

[1] *aips, dish, synthesis* and *vlbi* packages

6.2 General

There are eight documents, in addition to this one, which are essential to the code preparation and review process. Both programmer and reviewer should study them carefully, and base their work upon them.

- *code/install/codedevl/template-class-h:* how to construct a class header file.
- *code/aips/implement/Lattices/Slicer.h:* good example of a class header.
- *code/install/codedevl/template-module-h:* how to construct a module headers file.
- *code/aips/implement/Tables.h:* good example of a module header.
- *code/install/codedevl/template-global-functions-h:* how to construct a header file for global functions.
- *code/aips/implement/Arrays/ArrayIO.h:* good example of a global function header file.
- *code/doc/notes/167.latex:* Standards and Guidelines Checklist.
- *http://www.cv.nrao.edu/aips++/docs/html/cxx2html.html* An online guide to the documentation extractor, with good descriptions of the markup tags to be used in creating header files.

There are three main goals for the code review, all of which are equally important:

- Guarantee that the fundamental design of the class, as exposed in the public interface, is sensible and reasonably complete. We are more concerned about overall design than implementation details, because we assume that the programmer is competent; that the implementation may be too complicated to analyze in a reasonable amount of time; and that test programs and early usage will indicate errors and gross inefficiencies. The public interface, though, is everyone's concern, it is a good indicator of overall design, and it is painful to change once a class has been checked in.
- Make sure that the class header file is a sufficient guide to the class for client programmers. This is a very high standard, and demands a

great deal of work from the programmer and the reviewer. "Sufficient" means that a programmer, previously unacquainted with the class, can read the header file and quickly reach a point where he or she can write code that uses the class. This standard of explanation is often met by well-written programming books, and is rarely met by programmers in their own internal documentation. It is, nonetheless, the standard for all AIPS++ source code.

- Use the test program(s) to make sure that memory leaks are discovered and removed. See that code coverage is about 90% complete[2].

A great deal can be learned from reading through a class, and running test and demo programs. But no review is complete until other programmers use the new code. For this reason, we have a package called **trial**.[3] This package is automatically distributed by the AIPS++ code distribution system. If you ask the code cop to install your code in the trial directory, it will be available for everyone to examine and use. (All of these possible users will understand that *trial* code is subject to change with little or no notice.)

6.3 For the Programmer

- When you are within a week of submitting code, send the cop email briefly explaining the purpose of your class, and when it will arrive. This lead time will allow the cop to begin the search for a suitable reviewer.
- If your class is called, for example *"NewClass"*, then you need to submit the following files:
    1. *NewClass.h*: the class header
    2. *NewClass.cc*: the class implementation
    3. *tNewClass.cc*[4]: a test program – see section devoted to this topic below
    4. *dNewClass.cc* (optional): a short, heavily commented and well-focused program demonstrating how to use *NewClass*.

[2] Tools like "Purify" and "TestCenter" should be used if they are available.

[3] *(your-workspace)/code/trial* – at the same logical level as *aips, dish, vlbi, synthesis* and all of the site-specific packages

[4] If there is more than one test, exec, output or demo program, name subsequent files, for example *tNewClass1.cc, tNewClass2.cc* – the first program is implicitly number zero.

- Your code (header, source, test, and demo programs) should be combined into a standard unix "shar" file and emailed to *aips2-cop@nrao.edu*[5]. This shar file should preserve the directory structure needed to build the code once it is unshar'd.
- The cop will pass the shar file onto the reviewer, and tell you who the reviewer is.
- All subsequent email exchange between you and the reviewer should be CC'd to the cop.
- If you have test coverage and memory leak statistics (*strongly* recommended for all programmers who have access to the right tools) please send them to the cop in a separate email message.

6.4 For the Reviewer

Your job is a difficult one and requires a lot of time and effort. It may also require imagination and diplomacy. Imagination is needed because you must cultivate the perspective of the non-expert client programmer, intentionally forgetting things that you know, and scrutinizing the code (the class header in particular) to make sure that it explains everything the client programmer will need to know. You must also be able to take on the domain expert's point of view, and carefully examine the new class for the quality of its overall design. You may need considerable diplomacy to persuade a recalcitrant author that they need to make changes in their code and documentation.

Your responsibililites include these specific actions:

- Let the cop and the author know how long it will take to make the first review of the code, and how well you are sticking to that schedule as the time goes by.
- All email between you and the author which is related to the new class should be CC'd to the code cop.
- Be prepared for the cop to suggest changes beyond those you require.
- Base most of your evaluation on the header file for each class. A quick reading of the implementation file will allow you to judge the adequacy of its documentation, and detect any troubling features.

[5]If the class source code amounts to many thousands of lines of text, explain this to the cop – it may be possible for the reviewer to examine the code "in place", on your machine, or checked in to the *trial* directory.

6.5 Test Programs

The programmer must provide one or more programs that thorougly test the newly submitted code. These programs should be very simple to run, and should exercise all aspects of the new code.[6]

To indicate a successful test, it will often suffice for the test program to simply print"OK" to standard output, and return *0* as the value of the main program. In the test program fails, it should print a descriptive error message, and return some non-zero value to the environment. Some classes and global functions (especially those which employ numerical methods) can only be adequately tested by comparing a list of expected results against actual results, making allowance for tolerable roundoff differences[7].

The programmer may use two strategies here:

1. Create tMyClass.out, which contains the *expected* results, and tMyClass.exec, a shell script which runs tMyClass, redirects the actual results to a file, and compares them to the contents of tMyClass.out.

2. Alternatively, the tMyClass program *itself* creates the new results file, and runs the comparison, using a *system* call; when the comparison of results is complete, tMyClass prints "OK" and returns *0* to the environment if it is successful, just as the simpler test programs do.

---

[6] A handy way to construct a test program for a new class is to copy the class header file into a new file – which will become the test program – and edit the member function declarations into member function calls. This simple approach guarantees that all of the member functions are called. The next step would be to intentionally call all of the member functions in such a way that all the exceptions are thrown: if you trigger and catch all of the exceptions, then you have gone a long way towards exercising all of the code. "Domain" coverage should be attempted too: exercise the member functions by calling them with a range of inputs, paying particular attention to the boundaries of the domain.

[7] We hope that, in time, the AIPS++ project will find or develop a "diff" tool that is smart enough to compare numbers, and distinguish between allowable differences (due to word-length round-off) and unacceptable numerical error.

Part II

Useful Notes from AIPS++ Programmers

o

Chapter 7

Notes on using the g++ compiler to compile AIPS++ code

written as Note 196 by Ralph Marson, NRAO-NM
May 1996 (updated October 15, 2006) with assistance from Mark Calabretta, Mark Wieringa, Shelby Yang, Brian Glendenning & Wim Brouw

7.1 Motivation

The main reason I had for using the g++ compiler was that the template instantiation bug in the Sun native compiler was really getting to me. It meant for me that the time to compile nearly identical pieces of code could vary from $\sim$ 40s to $\sim$ 10mins, depending on whether the compiler (for unknown reasons) decided to re-instantiate everything or not. Sun is aware of this bug and a recent patch has alleviated this problem somewhat.

In g++ the template instantiation is done by hand (with assistance from the aips++ system) so this problem cannot occur. On a line by line basis the g++ compiler is slower than the Sun native compiler. In practice (my experience, and in the time taken for system rebuilds) it appears to be faster, because it does not do excessive recompilations.

Because of these problems with the standard Sun compiler and to aid porting of AIPS++ to other architectures, g++ has become the standard C++ compiler for AIPS++.

This note describes details on compiling AIPS++ code that are specific to using g++ and the “do it yourself” template instantiation mechanism that is used to specify which templates are created.

Code compiled and optimised under g++ is in my experience about 30% slower than code compiled and optimised under the Sun native compiler. Currently we are investigating using the using the "do it yourself" template instantiation mechanism in conjunction with the latest release of the Sun compiler and other compilers.

7.2 g++ system setup

These notes assume you have the g++ compilers installed and aips++ setup to use it. I know nothing about this and you should contact (in order) either, Pat Murphy or Brian Glendenning (bglenden@nrao.edu) on how to do this. The g++ compiler is now available at all of the consortium sites, and version 2.7.2 has been extensively used at Socorro. These notes relate to that version.

7.3 Switching to g++

Most sites are setup to have the gnu compiler as the default compiler.

If not you can switch to to the g++ compiler using:

```
aipsinit gnu
```

This assumes that you have already run aipsinit to get the normal AIPS++ environment variables set.

7.4 Setting up your workspace

Ensure that the following directories exist (otherwise create them)

```
${MYAIPS2ROOT}/sun4sol_gnu
${MYAIPS2ROOT}/sun4sol_gnu/aux
${MYAIPS2ROOT}/sun4sol_gnu/bin
${MYAIPS2ROOT}/sun4sol_gnu/lib
${MYAIPS2ROOT}/sun4sol_gnu/bindbg
${MYAIPS2ROOT}/sun4sol_gnu/libdbg
```

where `$MYAIPS2ROOT` is the root of your workspace. (eg. `$HOME/aips++`).

At Socorro where the code is on one partition and the executables are on another (which is not backed up), you should substitute for `$MYAIPS2ROOT` the path `/tarzan2/$USERNAME`, and create a symbolic link pointing from `/tarzan/$USERNAME/sun4sol_gnu` to `/tarzan2/$USERNAME/sun4sol_gnu`

7.5 Compiling

Compiling should be done in the normal way (ie. using gmake in the source directory), and with all the normal flags (eg. OPT=1). Using gmake, in my experience, with the OPT=1 flag results in faster compilation as well as optimised code. But the fastest way to recompile your code is to use the OPTLIB=1 option. This links your code (compiled with debug flags) against the optimised libraries, and compiles about 2-3 times faster than not using any flags. It has the added advantage that you can use the debugger to step through your code (but not the system code). Using gmake without any flags also results in executables that are about ten times bigger than the optimised version (ie. they are huge).

Debugging should be done using gdb (use ddd or xxgdb for a graphical interface), rather than the normal system debuggers (ie. dbx and friends). Note that the commands differ from the standard ones (e.g. no 'stop' but rather 'break'). Help can be obtained by typing 'help' or 'help subject', while information is available with the 'info' command (see 'help info').

If gmake dumps core it may be because you are using a very old version that exists in `/opt/local1/bin` (Socorro only).

7.6 Upcasting Bug

The g++ compiler cannot do a few things the Sun native compiler can. In particular if a global function with templated arguments of some base class is called with a class derived from the base class it will not find a match. (i.e. it cannot do upcasts)

This problem is particularly endemic in the Arrays module and for these classes a particular solution has been devised. The Array class (and hence derived classes) have an upcasting member function called `arrayCast()` (or `ac()` for short).

For example in the aips/Arrays/ArrayMath.h class there is a function

```
T max(const Array<T> &a)
```

which will not compile when called with a vector as an argument. ie.

```
Vector<Float> vec(4);
cout << max(vec) << endl; // error flagged on this line!
```

but will compile if rewritten as:

```
cout << max(vec.arrayCast()) << endl;
```

This solution gets particularly useful when doing arithmetic on Vectors because g++ will complain about:

```
Vector<Float> v1(4), v2(4);
v1 += v2;
```

And this should be changed to

```
v1.arrayCast() += v2.arrayCast(); // make the cast explicit
or
v1.ac() += v2.ac(); // emphasize the arithmetic not the cast
```

This problem has also been recently seen in the Lattice classes and a corresponding set of member functions called `latticeCast()`, or `lc()` for short, have been written.

On rare occasions you may need to selectively do an upcast when a templated argument can be either a builtin data type (Int/Float etc.) or an Array (Vector/Matrix etc.). For these occasions the functions `at_c()` & `at_cc()` exist. These global functions only upcast Arrays and leave the builtin data types alone. They are defined in the Array, Vector, Matrix & Cube classes[1], and in aips.h for the builtin data types (and in String.h also).

The difference between the two functions is that `at_c()` is takes a non-constant input arguement while `at_cc()` takes a constant input arguement. Two function names are used rather than overloading just one because of what we think is a bug in the g++ compiler.

An example of this function in use in is the Measures module where a Quantum has standard templates of `Double` and `Vector<Double>`. Then the floor function is defined in QMath.cc as:

```
template <class Qtype>
Quantum<Qtype> floor(const Quantum<Qtype> &left) {
    Qtype tmp = left.getValue();
    Qtype ret;
    at_c(ret) = floor(at_c(tmp));
    return (Quantum<Qtype>(ret,left));
}
```

If you see this problem in other class hierarchies, you can do an explicit type cast but are instead urged to create an upcasting member function in the base class. The advantage of using member functions, rather than

[1] you need to look at the .h files to see the definition as they do not show up in the .html files

explicit casts, is that if g++ is ever able to do upcasts in the future it will then be easier to search for the upcasting functions and remove them. Please tell me if you do this, so I can add it to the documentation.

It is expected that this bug will disappear in g++ version 2.8.0 (private communication between Brian and one of the g++ developers).

7.7 Typedef Bug

The g++ compiler sometimes complains it cannot find a type definition even when it has definitely been included. For example I found that it was complaining that `Vector<Int>` was undefined even though `aips/Arrays/Vector.h` was included. To solve this add the line:

```
typedef Vector<Int> gpp_vector_int;
```

near the top of the .cc file. the type `gpp_vector_int` is arbitrary but this seems like a good convention (Brian's convention).

This appears to occur when a specific templated types used in the context of a general template definition. eg.

```
template <class T> const Lattice<Bool> & PagedImage<T>::mask() const
```

will trigger this bug because requiring a typedef for `Lattice<Bool>`

I enclose this change and others that are specific to the gnu compiler in ifdefs's to avoid getting other compilers off side. eg.

```
#ifdef __GNUG__
   typedef Vector<Int> gpp_vector_int;
#endif
```

It is not necessary to do this when using the array upcasting functions (arrayCast() etc.) described in section 7.6.

The error messages that indicate you are being affected by this bug can be quite obscure and some are listed in section 7.11

7.8 Do it yourself templates (creating a templates file)

When linking the aips++ system will have a large number of template instantiations already done for you. However there may be some template instantiations you will have to create for yourself. This is done with a personal template instantiation list. To do this create a file called `templates`. This should be in the directory containing the test program for the class you are currently working on (eg., `.../code/trial/implement/MyModule/test/templates`) or in the applications directory (eg., `.../code/trial/apps/MyApp/templates`). gmake will see this file and "knows" how to create the templates you have specified. You should *never* create a templates file in a code module directory (eg. `.../code/trial/implement/MyModule/templates` as the templates in this file will overwrite the system templates (when you check the templates file in).

The format of the template entries is described in detail in Chapter 9 of the AIPS++ system manual (the mkinst section). I will only outline the basics here.

The templates file should contain entries like:

```
1010 trial/Arrays/Convolver.cc
     template class Convolver<Float>
1000 aips/Arrays/ArrayMath.cc aips/Arrays/Array.h
     template Double min(Array<Double> const &)
```

The entries are:

Identification number: The exact number is arbitrary but it should be a unique number for each .cc file, and should be (by convention) a 4 digit number. An value of four zeros will be converted to a unique number by "reident" (described below).

template file(s): The .cc file containing the template. If using templates like `myClass< Array<Float> >` you should also include a .h file containing the definitions of the template arguments. In this example `trial/Utilities/myClass.cc aips/Arrays/Array.h`

template: A keyword separating the filenames from the required template. This usually goes in a separate line although it is not manditory to do this.

required template: This states which specific template is required to be instantiated. It can either be a class or a global function. An example of each is shown above.

See the system template instantiation list in: `.../code/{aips,trial}/implement/_ReposFiller/templates` for further examples and a listing of the system generated templates. These templates are the ones necessary to link the aips++ applications.

Templates that are used by the test programs should not be put in the system file (unless there is an overriding reason), but stored in separate templates files that exist in the test directories. This keeps the aips++ libraries at a manageable size and avoids excessive link times.

If your templates file is getting long and unmanageable there is a utility called `reident` to sort/uniq, re-sequence and "canonicalise"the templates file.

In the default mode[2] reident will do the following:

- Adjust the identification number on the template definitions so that they are unique. It will also add identification numbers if they are missing and replace the identification numbers for all templates below 1000.
- Convert builtin types to AIPS++ standard definitions. eg., float will be converted to Float.
- Remove unnecessary white space.
- Warn if the const keywords are incorrectly positioned
- Warn if global functions do not have a return type
- Check that `#ifdef` and `#else` and `#endif` are properly sequenced.
- Comment out bad templates entries.

Its output is always to the file called templates (and input and output *can* be the same file) so I usually just type

```
reident
```

as the default input file is also called templates and I have already appended my changes tothis file.

reident can also merge multiple template files together so that

```
reident templates templates.additions
```

[2]There is an old mode enabled using the `-z` option which adds identification numbers starting at 1000 for each .cc file and increments by 10 to allow sufficient space for inserting new entries. You should rebuild your object library from scratch after running reident using this mode

will merge both input files into the output file.

The shell comment character `#` can be used to comment entries in your templates file and the preprocessor directives `#if`, `#if defined(symbol)`, `#if` defined(symbol)!, `#ifdef`, `#ifndef`, `#else`, `#endif` can also be used.

Each entry in the templates file gets compiled into an object file called myClass_Ident.o where the myClass is the name of the first .cc file in the template line and Ident is the indentification number. Normally this object file contains only one instantiation. But often templates instaitions are required in groups. For example to use a `CountedPtr<SkyCompRep>` requires you instantiate four other classes, namely: `CountedConstPtr<SkyCompRep>`, `PtrRep<SkyCompRep>`, `SimpleCountedConstPtr<SkyCompRep>` & `SimpleCountedPtr<SkyCompRe`
In this case it is better to group all five instantiations in one object file. This is done by adding multiple template lines as shown below.

```
1000 aips/Utilities/CountedPtr.cc trial/ComponentModels/SkyCompRep.h
     template class CountedConstPtr<SkyCompRep>
     template class CountedPtr<SkyCompRep>
     template class PtrRep<SkyCompRep>
     template class SimpleCountedConstPtr<SkyCompRep>
     template class SimpleCountedPtr<SkyCompRep>
```

7.9 Demangling the template names

When linking the names of the missing class/function instantiations may be printed in a mangled form. To demangle them save the g++ output in a file and use `/usr/local/gnu/bin/c++filt`.
An example (Using csh) is:

```
gmake test.cc >&! g++.out
/usr/local/gnu/bin/c++filt < g++.out
```

or using sh

```
gmake test.cc 2>&1 | /usr/local/gnu/bin/c++filt | tee missing-templates
```

The c++filt command will translate the following mangled name

```
getArray__Ct11MaskedArray1Z14PointComponen
```

to

```
MaskedArray<PointComponent>::getArray(void)
```

and because the getArray function is a member function of the MaskedArray class we need to instantiate the whole class (`MaskedArray<PointComponent>`)

7.10 Checking in your templates

When code is checked in, the templates entries should be added to the relevant system/test template instantiation file:

```
.../code/{aips,trial}/implement/\_ReposFiller/templates
```

if the templates are associated with an application or

```
.../code/{aips,trial}/implement/yourModule/test/templates
```

if the templates are associated with a test program. [3]

Be very careful when adding your templates to the system file as any duplication of the indent. number between your added templates and the already defined ones will cause the next system rebuild to fail (a sure way to become popular). Mark Calabretta suggests:

"When you add idents to the system templates file you should always run 'reident' before checking it back in. A good strategy is to give your new entries a 0000 ident."

A question that is often asked is "when do I put a template in the aips package and when do I put it in the trial package?". If any template required for an application uses code (.cc or .h files) only from the aips package it should go in the `aips/implement/_Reposfiller/template` file. If it uses anything from the trial package it should go in the `trial/implement/_Reposfiller/template` template file. This avoids the problem of having undefined symbols when some purely aips templates are defined in the trial library (at the expense of having a larger than necessary aips library).

Pure aips templates that are needed for test programs in the aips package should still be in `aips/implement/yourModule/test/templates`. The aips test programs should not rely on anything in trial, they are not linked against the trial libraries, and should not use any templates from the trial package.

Any templates (pure aips or involving trial) that are needed for test programs in the trial package should still be in the `trial/implement/yourModule/test/templates` file. This may mean that you may have to use the `EXTRA_PGMRLIBS` flag when compiling these test programs, although we will attempt to automate this.

You should also ensure that the templates you are adding to the system are not defined alsewhere. Duplicate templates lead to link time errors on many systems (eg. Dec Alpha). The policy for checking for duplicates is:

[3] If you are modifying an existing class and you find you need to add additional templates to the test program it is likely that these same templates will be used by other programs using this class. So you will need to go and find them and if they are applications then just put the templates in the system (_Reposfiller) templates file, otherwise you will need to modify the templates file of other test programs that use the modified class

- If checking new templates into `aips/implement/_Reposfiller/templates` then you *must* ensure that the same template is not duplicated in any of the following template files:
  - `aips/implement/anyModule/test/templates` (remove templates from test if any found)
  - `trial/implement/anyModule/test/templates` (remove templates from test if any found)
- If checking new templates into `aips/implement/anyModule/test/templates` then you *must* ensure that the same template is not duplicated in
  - `aips/implement/_Reposfiller/templates`. (template in test should not be necessary)
- If checking new templates into `trial/implement/_Reposfiller/templates` you need to check
  - `trial/implement/anyModule/test/templates` (remove templates from test if any found)
- If checking new templates into `trial/implement/anyModule/test/templates` then you need to check:
  - `aips/implement/_Reposfiller/templates` (template in test should not be necessary)
  - `trial/implement/_Reposfiller/templates` (template in test should not be necessary)

Brian Glendenning has a tool that can automatically check for template duplicates. It is checked into the system and information on how to use it can be obtained by typing `duplicates -help`

7.11 Error Messages

The gnu compiler can generate many different error messages that are in fact caused by the problems described in section 7.6 and 7.7. Here I intend to list the error messages reported for each of the problems as an aid to programmers. Please forward new error messages to me (rmarson@nrao.edu).

7.11.1 Upcasting Bug (section 7.6)

- type unification failed for function template
- too few arguments for function

The upcasting bug has also been known to cause an exception to be thrown. This occured in the following piece of code:

```
template <class Qtype>
ostream& operator<< (ostream &os, const Quantum<Qtype> &ku) {
    os << ku.qVal;
    return os;
}
```

where ku.qVal could be a `Double` or a `Vector<Double>` (but the `operator<<` function is defined for Arrays not Vectors). The compiler did not detect that there was an upcasting problem and examination with a debugger revealed it was getting into an infinite constructor loop! Further investigation revealed that the compiler did not find a function match using an upcast but was able to find another unintended match, that was further resolved to the original function!

7.11.2 Missing Typedef Bug (section 7.7)

- invalid use of undefined type
- parse error at null character

7.11.3 Other Gotcha's

You will probably have problems linking when you modify a class and then create a new set of templates (all in your private workspace), if the same templates also exist in the system library. This is because the newly modified templates may not be required until after your private library has been searched and the linker is using the system library. By this stage it will not know to use the templates in your personal library and instead pickup the system ones. The solution to this problem is.

- Wait for a system update. This may be overnight or take a week.
- Copy the system library into your own workspace prior to compiling the templates.

Chapter 8

Getting Started with Glish

written as Note 195 by Rick Fisher, NRAO-GB
April 1996

8.1 Introduction

This document is an AIPS++ applications-specific supplement to "The Glish 2.7 User Manual.". Glish is both a programming language and an environment for data acquisition and analysis. It contains many of the mathematical features of other languages such as Fortran and C, and it is particularly strong on array arithmetic. To the user, glish becomes AIPS++ or a telescope control system with the addition of functions and "clients" for those applications. In fact, both applications may be run from the same invocation of glish, if so desired.

Here we shall cover the main language features with simple examples drawn from test data acquired by the GBT hardware systems now under development. The emphasis will be on the tools for writing your own data and control functions rather than on a review of AIPS++ library functions. Also, only a few of the built-in glish functions will be mentioned. See section 10.12 of the Glish User Manual for a complete function summary, and the AIPS++ User Reference Manual for information on the AIPS++ library. A few elements of the AIPS++ library are used below; when discussed the relevant section of the AIPS++ User Reference Manual is available via the link.

I shall assume that you have access to a working AIPS++ installation. Details about setting up your AIPS++ environment are given in Getting Started in AIPS++.

8.2 Starting AIPS++ and Glish.

In a new directory you can establish an AIPS++ and glish environment with the following UNIX command:

```
For csh-like shells:
$ source /aips++/stable/aipsinit.csh

For bash-like shells:
% . /aips++/stable/aipsinit.sh
```

Note that the directory name, `/aips++/stable`, may vary depending upon your installation; see Getting Started in AIPS++ for more details.

Then start AIPS++, open the example files, and load the example glish scripts with

```
$ aips++
- include 'glishtutorial.g'
```

The script 'glishtutorial.g' converts the example FITS data files used in this tutorial into tables which can be read directly in aips++. When the conversion process has completed the glish prompt will return.

AIPS++ without the example files and scripts can be started with simply

```
$ aips++
```

The glish prompt is either '-' or '+'. To test glish and plot window operation run the commands:

```
- mp := pgplotter()
- mp.plotxy([1:100]/5.0 ,sin([1:100] / 5.0), title="sine wave")
```

You should see about three cycles of a sine wave. This is plotted using the AIPS++ utility pgplotter.

8.3 Variable Assignment and Automatic Typing

Glish variables are automatically assigned a variable type when they are created. In other words, there is no variable declaration statement. After the following assignments:

```
- x := 2
- z := 23.786
- check := F
- status := T
- amp := 1.66 - 0.994i
```

'x' is an integer, 'z' is a double, 'check' and 'status' are booleans (true or false), and 'amp' is double complex. The available data types are boolean, byte, short, integer, float, double, complex, and double complex. Floating point constants are assumed to be double, and integer constants are assumed to be integer. In most cases the data type is self-evident, but, if your code needs to know a variable's type, there are built-in test functions such as is_boolean(), is_float(), is_dcomplex(), etc., which return a boolean value T or F. Also, the function type_name() will return a character string describing the type. For example,

```
- is_float(z)
F

- type_name(z)
double
```

Automatic variable typing is quite convenient, but it provides no warning when you inadvertently reassign a variable. The assignment

```
- status := 1.65e7
```

is perfectly legal resulting in 'status' now being a double. You can protect a variable from reassignment with the 'const' keyword as in

```
- const Pi := 3.14159
```

Values of variables and constants in an arithmetic expression are all converted to the type of highest numeric precision in the expression before performing the mathematical operations. In an expression using the variables above

```
a := x * z * check
```

'a' is a double. Variable typing may be forced for purposes such as integer truncation or storage economy with the built-in functions as_integer(), as_float(), etc. Keep in mind, however, that a variable is retyped every time it is assigned, so, on a few occasions, your expressions may need to be very explicit about type.

The arithmetic operators are

```
+  -  *  /  %  ^
```

where % is the modulus operator that converts its operand values to integer and returns an integer result, and ^ is the exponentiation operator with floating point arguments and result.

```
- 26.8 % 4.2
2
- 3.01^1.99
8.960811
```

The value of a variable may be printed with the print statement or, in interactive mode, by simply typing the variable name without the ':=' assignment operator. The following two statements are equivalent:

```
- print x
- x
```

Be careful with array names! You can get a real screen full.

8.4 Arrays

Array operations are one of glish's strong points for both code economy and computing efficiency. The notation takes a bit of getting used to.

An array variable may be created either by assigning to it an array constant or the result of an array expression,

```
- indices := 1:512
- ar3 := ar2 * ar1
```

or by using the array initialization function,

```
- image1 := array(0.0, 256, 256)
```

The array variable, 'image1', becomes the type of the first argument of 'array()', which is the array initialization value. The first argument itself may be a scalar or an array. The rest of the arguments are array dimensions of any length and number that will fit in memory.

Most arrays are created by assignment. The primary use for the 'array()' function is to create an empty array to be filled piecemeal later. Again, remember that an assignment can change the size of an array or even turn it into a scalar.

Users of IDL, PV-Wave, and, to some extent, IRAF will find glish's array index notation familiar. Fortran and C programmers will be tempted to attack arrays with integer indices and iterative loops. Resist the temptation when possible because the intrinsic array operations are much faster.

To avoid an abstraction of arrays let's load a continuum receiver data column from one of the sample data tables. A small portion of this table looks like

RECEIVER_ID	PHASE	CAL	SUBSCN	DATA
0	0	0	1	600885
1	0	0	1	572540
0	1	1	1	649341
1	1	1	1	617254
0	0	0	2	601413
1	0	0	2	573158
0	1	1	2	649827
1	1	1	2	617841
0	0	0	3	602472
1	0	0	3	574106

There are many more rows and columns to this table, but these include the interesting columns for the moment.

Now load the 'DATA' column from table t1 into a glish array. This table has already been opened using the AIPS++ table system if you included 'glishtutorial.g'.

```
- data := t1.getcol('DATA')
```

If you want a complete picture of this table, browse the table with

```
- t1.browse()
```

The vector 'data' (1-D array) contains interleaved data from two receiver channels with the noise calibration signal turned on and off. To look at a single element just use an integer index.

```
- data[3]
649341
```

Notice that indices start at 1. To access a contiguous range of values in the vector use an index range.

```
- data[1:5]
[600885 572540 649341 617254 601413]
```

Any subset of vector contents may be acquired with an index array containing the indices of the values you want.

```
- indx := [2,5,9]
- data[indx]
[572540 601413 602472]
```

or just simply

```
- data[[2,5,9]]
[572540 601413 602472]
```

Notice the second set of square brackets which define the index list as members of an integer vector rather than indices of different dimensions of a three-dimensional array. To digress a moment, a multidimensional array is addressed as follows:

```
- mda := array(0.0, 16, 16, 16)
- mda[5, 3, 14] # column 5, row 3, slice 14
- mda[ , 6, 5] # all values in row 6, slice 5
- mda[11, , ] # values in the column 11 plane
- mda[ , ,3:5] # 3 planes of slices 3, 4, and 5
```

The # character is the comment delimiter. Everything from it to the end of the line is ignored by glish.

If you like, you can load an example of a two-dimensional array of data from a different table that contains a column of spectra.

```
- spectra := t7.getcol('DATA')
```

Glish variables have attributes, a few of which are predefined. The dimensions of an array are contained in its 'shape' attribute. The syntax for an attribute variable is

```
- spectra::shape
[1024 33]
```

In this case, there are 33 spectra of 1024 channels each.

Back to our 'data' vector, rather than generate an index vector it is more efficient to create a boolean vector with the same number of elements as the 'data' vector and use it as an element mask. For example, to get all of the 'data' values for receiver 1 with the calibration noise "on" get the RECEIVER_ID and CAL columns.

```
- rcvr := t1.getcol('RECEIVER_ID');
- cal := t1.getcol('CAL')
```

Then make a boolean vector of the same length and use it as an index mask to the 'data' vector.

```
- mask := (rcvr == 1) & (cal == 1)
- mask[1:10]
[F F F T F F F T F F]

- data_cal_1 := data[mask]
- data_cal_1[1:5]
[617254 617841 618768 619804 620325]

- print length(data), length(data_cal_1)
960 240
```

From the values and length of 'data_cal_1' we can see that it contains the values from the 'data' vector where the boolean vector, 'mask' , is true. The function length() returns the number of elements in the array, not the number of bytes. The equivalent subarray access could be written without the intermediate 'mask' variable.

```
- data_cal_1 := data[rcvr == 1 & cal == 1]
```

We have also used the operator precedence rules to eliminate the parentheses, but, when in doubt, use parentheses. The comparison operators are

```
<  >  <=  >=  ==  !=
```

which are less than, greater than, less than or equal to, greater than or equal to, equal to, and not equal to, respectively. The boolean operators are "and", "or", and "not".

```
&  |  !
```

All of the arithmetic, comparison, and boolean operators operate on arrays element by element. Therefore, all arrays in an expression must be of the same size in each dimension. For example,

```
- [1:3] * [1:3]
[1 4 9]
```

It is not a vector dot product.

For a graphical illustration of what we have done with the extracted data column get one more column, 'Time', from the table and plot the two data vectors against it.

```
- times := t1.getcol('Time')
- times -:= as_integer(times) # remove day numbers
- mp := pgplotter()
- mp.env(.96294,.96304,5.6e5,6.6e5,0,0)
- mp.line(times[1:32], data[1:32])
- sometimes := (times[1:32])[mask[1:32]]
- mp.eras()
- mp.env(.96294,.96304,6.1e5,6.3e5,0,0)
- mp.line(sometimes, data_cal_1[1:length(sometimes)])
```

The second line above illustrates one of the combined arithmetic and assignment operators (+:= -:= *:= /:= %:= ˆ:= &:= —:=) that are also found in C. The assignment of 'sometimes' is a combined use of vector ranges and a mask.

If we assume that the receiver 1 noise calibration intensity is 1.4 Kelvins, we can plot a calibrated scan with the additional commands

```
- data_cal_off_1 := data[rcvr == 1 & cal == 0]
- dataK := 1.4 * ((data_cal_1 + data_cal_off_1) / 2.0) /
+ (data_cal_1 - data_cal_off_1)
- mp.eras()
- mp.plotxy([1:len(dataK)],dataK)
```

Note that a different (and simpler) plotting function, mp.plotxy(), replaces the mp.env() and mp.line() commands used previously. The latter functions were used at first because of the unusual requirements on the axes ranges.

You should see four scans through a radio source placed end to end. These are actually four separate scans in the same table that were taken with a cross pointing scan procedure on the 140-foot. Note that the `+` on the third line in the previous example is the glish prompt and not an indication that you should type the plus character.

All of the other column names in this table may be listed with

```
- t1.colnames()
```

Here are some additional indexing ideas.

Rotate a vector five places.

```
- rot_vector := data[[5:data::shape,1:4]]
```

Delete out-of-range values.

```
- purged := data[data < 700000]
```

Make a new time vector to match the 'purged' vector.

```
- ptime := times[data < 700000]
```

Reverse a vector.

```
- flipped := data[data::shape:1]
```

Splice two vectors together.

```
- spliced := [v1, v2]
```

8.5 Scripts and Functions

Glish code may be written in a file with an editor and executed with the 'include' command. These scripts contain glish code that is identical to the text that you type at the glish prompt. Script files are conventionally given the name extension '.g'. The glish command syntax is

```
- include 'myscript.g'
```

The most frequent use of script files is to edit and load function definitions. The syntax may be either

```
function <fn name> ( <arguments> ) {
    statements
}
```

or

```
<fn name> := function ( <arguments> ) {
    statements
}
```

The 'function' keyword may be abbreviated 'func'. The curly brackets are to group all of the statements together. Without the brackets, only the one statement following the function arguments would be part of the function. Glish is pretty liberal about whether brackets and statements appear on same or different lines and about how you use indentation, as long as the meaning is unambiguous. A function may return any variable type including whole arrays.

Here is a simple function for plotting data from one receiver of one scan in a table.

```
plot_scan := function (tab, scan_no, receiver=1)
{
    mp := pgplotter()
    data := tab.getcol('DATA');
    scan := tab.getcol('SCAN');
    rcvr := tab.getcol('RECEIVER_ID');

    mask := scan == scan_no & receiver == (rcvr + 1);
    scan_data := data[mask];
    if (length(scan_data) == 0) {
        fail paste('No data found for scan', scan_no,
                   'receiver', receiver);
    }
    mp.plotxy([1:len(data[mask])], data[mask],
              title=paste('Scan', scan_no, 'Receiver', receiver));
           return
}
```

Type this function into a script file, say, 'test.g'. Load it into glish with

```
- include 'test.g'
```

and execute the function with

```
- plot_scan(t1, 48)
```

There are a few new items in the function definition above. The lines are terminated with semicolons. This is optional both in a script file and at the glish prompt. If you put two lines of code on the same physical line, they must be separated by semicolons. The function contains an if() statement whose argument is normally a boolean expression, but it can be a numeric variable. Zero is false, and everything else is true. 'pgplotter()' and 'mp.plotxy()' are AIPS++ functions. 'paste()' is a glish built-in string concatenation function. 'fail' is a glish built-in that causes an error return; use it like this when you want to signal an error.

Function arguments may be defaulted by assigning them a value in the function definition, as we did with the 'receiver' argument. When a function is called, if no value is given for a defaulted argument, the argument assumes the value given in the function definition. Undefaulted arguments must be given values in a function call. Normally, arguments are given values according to their position in the argument list, but they can be given in any order, if they are named. For example, the plot_scan function call above could also have been done with

```
- plot_scan(scan_no=48, receiver=1, tab=t1)
```

The first arguments may be specified by position, but after an argument is named all the rest must be named.

Variables defined by assignment within functions are local to that function. In other words, when the function execution completes the variable goes away. A variable which doesn't disappear may be created by declaring it to be "global". Likewise, an externally defined variable may be assigned a value within a function by declaring that variable to be global within the function. For example, look at the results of the execution of two functions defined below.

```
x := 10
bill := function () {
    x := 5
}
fred := function () {
```

```
    global x := 6
}

- x
10
- bill()
F
- x
10
- fred()
F
- x
6
```

The `F` that appears after you execute the functions is the glish boolean, False. This is the default return value for functions which do not explicitly return a value.

An unusual feature of glish is that global variables can be implicitly referenced from within a function. However, by default, they become local to the function upon assignment, for example,

```
- gvar := 0.841471
- function x() {
+    print gvar
+    gvar := 90
warning, scope of "gvar" goes from global to local
+    print gvar
+ }
- x()
0.841471
90
- print gvar
0.841471
```

Glish provides a warning whenever the scope of a variable goes from global to local. If you want the global variable to be affected, you must explicitly declare it with the 'global' keyword, as in

```
- gvar := 0.841471
- function x() {
```

```
+    global gvar := 90
+ }
- x()
- print gvar
90
```

Finally, if you don't like the name of a function, or it's too long to type conveniently, you can give it an alias by assigning the function name to another name, just like assigning one variable to another.

```
- ps := plot_scan # note no parenthesis
- ps(t1, 48)
```

Be careful, however! You can wipe out an entire function definition by reassigning it. If you make the assignment,

```
- plot_scan := 5
```

the function plot_scan() is gone until you redefine it by including its script file again. However, the variable ps still contains the copy of plot_scan that you assigned to it earlier.

As with variables, you can protect a function name and its arguments with the 'const' keyword, for example,

```
- const myplot := function (const ary) { ... }
```

After you have finished debugging a function, it is usually a good idea to protect it with 'const'.

8.6 Loops and Conditionals

Three familiar program flow control statements are available in glish, "if/else", "while", and "for" loop statements. The general syntax for each is

```
if (<boolean condition>) {
    statements
} else { # there is no 'else if'
    other statements
}

while (<boolean condition>) {
```

```
    execute all of this
}

for ( <variable> in <vector> ) {
    execute all of this
}
```

The "for" loop is a bit different to its equivalents in Fortran and C in the sense that the running variable doesn't have to make equal steps. It just takes on the values of the elements of the vector in succession, whatever those values may be. A garden variety loop of i from 1 to 10 in steps of 1 is

```
for (i in 1:10) {
}
```

Every method you can think of for generating a vector can be used to create the vector argument in the "for" statement. For example, you can use the 'seq()' built-in function to step from 3.66 to 7.35 in steps of 0.0123.

```
for (x in seq(3.66, 7.35, 0.0123)) {
}
```

Even a boolean vector is OK.

```
for (i in [F, T, T, F, T]) {
    print i;
}
```

8.7 Records and Attributes

Scalars and arrays of the standard data types are not the only variables allowed by glish. Heterogeneous data structures, called records, can be treated as a single variable. These can be passed as arguments to a function and returned as a unit in a function return value. A record may be created all at once or added to piece by piece.

```
- header := [name='3C273', ra=12.12345, dec=2.3318]
- header.flux := 27.6
- header.type := 'quasar'
- print header.ra
12.12345
```

Record members may be strings, any of the numeric data types including arrays, as well as other records.

Any glish variable may be assigned attributes. For example,

```
- velocity::definition := 'relativistic'
- header.flux::units := 'Jy'
```

The distinction between record members and variable attributes is a subtle one, and it is seldom clear which is best suited to a particular application. Both record members and attributes may be accessed by name or by number. All of the following are acceptable:

```
- header.ra
- header[2]
- header['type']
- velocity::[1]
- velocity::definition
- velocity::['definition']
```

Numeric access lends itself to stepping through the fields or attributes with a loop index. Record field names may be retrieved with the built-in function 'field_names()', which returns a string.

```
- field_names(header)
name ra dec flux type
```

The empty attribute construct returns the attributes of a variable as a record, so you can use the 'field_names()' function on attributes, too.

```
- velocity::
[definition=relativistic]
- field_names(velocity::)
definition
```

8.8 Strings

There are two types of string constants in glish. Single quoted strings are treated as a single entity and are most useful in print statements where some control of placement of characters on the screen is important. Double quoted strings are treated as arrays of strings with white space separating the array members. The output of 'field_names()' in the example above is an instance of a string array. The difference is best illustrated with some examples.

```
- s1 := 'This is a single quoted      string'
- s2 := "This is a double quoted      string"
- s1
This is a single quoted      string
- s2
This is a double quoted string
- s1[1]
This is a single quoted      string
- s2[1]
This
- s1[2]
<fail>: index (= 2) out of range, array length = 1
- s2[2]
is
```

Notice that all white spaces have been reduced to single spaces when the double quoted string array is printed.

String handling is fairly primitive in glish. There are no string operators, except possibly for the cautious use of the comparison operators.

```
- twenty := '20'
- three := '3'
- twenty < three
T
```

If one of the variables, twenty or three, had not been a string, an error would have been reported. When in doubt, use the is_string() test first.

There are a few string manipulation functions, including paste(), spaste(), and split(). Paste() and spaste() act like print statements except that the resulting string is returned as a single string instead of being printed.

```
- r := 53.6
- sx := paste('R =', r)
- sx
R = 53.6
```

The paste() function can take a named argument, sep, to change the separator character(s) from the space default.

```
- paste('R', r, sep=' = ')
R = 53.6
```

The function spaste() is the same as paste() with no separator. The string function split() simply turns a single string into a string array or breaks a string array differently. It takes two arguments, the string to be split and a list of characters to be used as separators. The default separator is a space.

```
- str1 := 'This is a single quoted string'
- str2 := "This is a double quoted string"
- sx := split(str1)
- sx[1]
This
- sx := split(str2, "u")
- sx
This is a do ble q oted string
- sx[3]
oted string
```

8.9 Printing

The print statement is quite primitive so output formatting in glish is minimal. The output of print is simply the concatenation of the quantities listed converted to strings and separated by spaces. You can add spaces by printing spaces in single quoted strings, and you can control the number of decimal places shown for floating point numbers with one of two print precision variables.

```
- r := 3; s := 5.6; j := "spa   ces"
- print '     show', r, s, j
     show 3 5.6 spa ces
```

The printed precision of all floating point variables is set by the value of a member of the system record variable.

```
- old_prec := system.print.precision
- system.print.precision := 3
- x := 234.12345
- print x
234
- system.print.precision := old_prec
```

The printed precision of one variable may be set by the member of the variable's print attribute.

```
- x::print.precision := 4
- print x
234.1
```

To avoid the possibility of inadvertently printing a huge array to the screen, you can set a limit on the number of printed array elements with

```
- system.print.limit := 50
or
- any_array::print.limit := 50
```

The effects of print.precision or print.limit may be cancelled by setting them to a value less than zero. You may want to set a system print limit in your .glishrc file. See Appendix A and Getting Started in AIPS++ for more details on the role and uses of your .glishrc file.

8.10 Clients, Events, and Agents

From a software designer's point of view, the important part of glish is its ability to link to any number of independent processes running anywhere on a computer network. To glish these independent processes are seen as "clients", but the client-server distinction is ambiguous since services can flow either way. Communication between glish and its clients is by way of "events", rather analogous to hardware interrupts but with the added feature that events can carry any amount of data to or from a client.

As an AIPS++ user you may seldom encounter the use of events in your glish programming. Events are considerably different in concept from usual linear scientific programming. Hence, they are usually wrapped in more familiar function calls to avoid the need to learn events to use the services of AIPS++ clients. In fact, in the examples above we have used a number of clients, including the table browser and the plot client, without necessarily being aware of events passing back and forth. It's not that clients and events are all that obscure. It is just that they are one more concept to learn that might not be of any direct interest.

Clients can do anything that a normal computer program can do from performing FFT's to driving a telescope to position. Most of the application-specific facilities of AIPS++ are in the clients that glish initiates.

To see how the events work try the following commands associated with the Tk widgets attached to glish.

```
- mp := pgplotter()
- fm := frame()
- b1 := button(fm, 'plot')
- whenever b1->press do {
-  mp.eras()
-  mp.plotxy([1:100],sin(as_float(1:100) / 20.0), title='Button plot')
- }
```

The first statement puts a small window frame on the screen. The second puts a button labeled 'plot' in the window. In the whenever statement, "press" is the name of an event sent by the Tk widgets client when the button is pressed with the mouse. The statements within the whenever block are executed when the 'press' event is received. Try pressing the button.

Clients are started from glish with the client() function. A more complex example than the button client might be a hypothetical FFT engine. Let's say that we have an executable file on disk, called "fft_engine", that performs Fourier transforms and knows how to send and receive glish events. Start it up with

```
- fft := client('fft_engine', host='arcturus')
```

The host argument is not needed if "fft_engine" is on the same machine that is running glish. The variable, fft, is of a special type called an agent.

Suppose that the documentation for fft_engine says that it recognizes an event called "do_fft", and , when the FFT is done, fft_engine will send an event called "fft_done" with a new array of transformed data. First, set up to receive the "fft_done" event and do something with it. Then send the fft request with the data array. Probably in a script called 'fft.g' we'd have

```
fft := client('fft_engine')
whenever fft->fft_done do {
   mp.eras()
   mp.plotxy([1:len($value)], $value, title='FFT Result')
}
```

Then from the command line do the following:

```
- include 'fft.g'
- data := sin(as_float(1:128) / 3.0)
- send fft->do_fft(data)
- await fft->fft_done
```

In this case we have elected to wait for the "fft_done" event to return and be processed, but we could have continued to issue glish statements for other purposes. The FFT result would be returned and plotted whenever the fft_engine client was finished with it.

The parameter, $value, is a special one associated with events. It contains the event's data and can be anything from a single number or string to a very complicated record with arrays and other records as members. Normally, glish and the client know what to expect in $value, but some flexibility of contents and data types can be supported with the is_xxxx() test functions.

A fully implemented glish client is written at least partly in C++ using the classes that support events and glish data structures. However, existing programs may be turned into simple clients using standard input and output ASCII data as event values. Also, clients may be written in the glish language, but there are fewer reasons for doing it this way.

The standard-in, standard-out client is created by the shell() function rather than the client() function. A one-time communication with a client program is simply the UNIX command as a string argument to shell().

```
- file_list := shell('ls *.g')
```

The variable, file_list, is then an array of strings, one string for each file name that ends with ".g" in the directory.

If the program can handle more than one exchange of data by way of UNIX pipes, it may be started and communicated to with events. The startup glish code looks something like

```
a := shell('my_prog', async=T)
whenever a->stdout do {
    print $value
}
whenever a->stderr do {
    print 'Error !', $value
}
```

The async=T parameter assignment causes the shell() function to return an agent type quantity instead of a string. The data events that can be sent by the client started with the shell command are called "stdout", and "stderr" and their $value is always a string. The two events recognized by a shell client are "stdin", whose argument must always be a string, and "EOF". Hence, an event sent to "my_prog" is, for example,

```
- send a->stdin('23.876 14.999')
```

To avoid having an event arrive from a new client before its associated whenever statements are set up, put the client() or shell() function calls and the whenever statement in the same script file, and run them together in the script. Glish guarantees that events are held until the full script file is read.

8.11 What now?

To learn more about Glish, you might wish to read the "Glish 2.7 User Manual". For information on AIPS++, see Getting Started in AIPS++ and the AIPS++ User Reference Manual.

8.12 Appendix A - Glish Environment

Glish can be most easily run by setting up the AIPS++ environment. For information on how to do this, see Getting Started in AIPS++.

Glish, itself, reads an environment file, called .glishrc, which may be used to set a number of system variables. The most important one is system.path.include, which specifies the directories to be searched for ".g" files that are loaded with the include statement. Any ".g" file may be loaded by giving its full path name, but the system.path.include assignment saves a lot of typing. The contents of a .glishrc file might look like

```
print 'reading /hyades1/rfisher/Pulsars/P1823/.glishrc'

system.path.include :=
    ". /gbt3/aips++/sun4sol/libexec /aips2/glishScripts-2509a /";

print 'system.path.include: '
for (i in 1:len(system.path.include))
    print '    path', i, system.path.include [i];
```

The print statements are just embellishments to remind you of where the environment variables are coming from when glish starts up. Notice the '.' path name as well as the system glish directories.

When glish starts up it reads the .glishrc file in the system glish directory. It then looks for a .glishrc file in your current directory. If none is found there, it tries your home directory. Note that the system.path.include assignment shown above will override the one in the system .glishrc file. If you just want to add additional paths, you can use the following construct to expand the string array:

```
system.path.include[len(system.path.include) + 1] := '/home/rfisher'
```

Arrays are automatically expanded in an assignment statement when the index of the assigned array is greater than its current size. "len()" is another name for the length() function.

Finally, it's worth a look at the UNIX script that starts the glish/AIPS++ combination. This script is the command given at the beginning of this document, for example, "aips++". To find out where this script is, use the UNIX 'which' command. Then list the file using the full path name.

```
$ which aips++
/aips++/stable/sun4sol_gnu/bin/aips++
$ more /aips++/stable/sun4sol_gnu/bin/aips++
[ lines setting up the aips++ environment variables deleted ]
exec $BINDIR/glish -l $LIBDIR/aips++path.g -l $LIBDIR/aips++.g
```

The last line executed in this script is the command that starts glish. It automatically includes two files, `aips++path.g` and `aips++.g`. Both files contains glish code which loads the aips++ clients. The "-l" (minus ell) flag tells glish to keep running in interactive mode. Without this flag the script `aips++.g` would be executed and then glish would terminate.

If you want to start glish without any of the AIPS++ functions or clients, just issue the UNIX command

```
$ glish
```

8.13 Appendix B - Example Data Files

The two AIPS++ tables, t1 and t7, used in the examples in this document were created from some GBT system test data from the 140-ft telescope.

Other instruments, such as the VLA, will have a different file and directory structure for the original data from the one outlined below, but many properties of the created AIPS++ tables will be the same.

The original data are in several FITS Binary Table files written by different system modules, e.g., the weather monitor, telescope position control, and one of the data backends. The t1 table contains data from the continuum backend, and t7 is a bit of spectral processor data in spectral line mode. There will be an automated process to convert the raw FITS data into tables readable by AIPS++. For now, the data must be filled by hand. The utility to fill data from the GBT backends, for example, is called gbtmsfiller.

A step to opening tables within glish was hidden by the glishtutorial.g script. Namely, to open a table use:

```
- t7 := table('t7')
```

where the argument to the table() function is the directory produced by the filler.

Index