NOTE 167
AIPS++ Programming:
STANDARDS AND GUIDELINES

Paul Shannon, NRAO
Revision: Wim Brouw, ATNF

February 27, 1995
Revised: December 21, 1999

1 Introduction

These standards and guidelines have been chosen to promote a consistent style, clarity and organization in AIPS++ code. *Standards* are coding practices which everyone must follow; *guidelines* are recommended practices, and are identified below by (∗). Use of standards and guidelines will in general reduce the number of initial defects by about 25%, and make maintenance of code easier.

Effective C++ and *More Effective C++* by Scott Meyers is a good guide to C++ programming. Most of his guidelines apply to AIPS++.

2 Code Organization

2.1 Classes

Every class will have a header file,[1] an implementation file,[2] a test program [3] and (optionally) a demo program. Very closely related classes[4] may sometimes be combined into single header and implementation files.[5]

All files shall follow the latest template file as a guide for the standard layout.

It is sometimes advisable to separate the implementation into different files. Examples are to separate templated and non-templated methods, and when a sub-set of the methods

[1]See *code/install/codedevl/template-class-h*

[2]See *code/install/codedevl/template-class-cc*. Neither totally inlined classes, nor abstract base classes may need an implementation file.

[3]See *code/install/codedevl/template-main-cc*

[4]A good example is a read-only class, and a read-write class derived from it.

[5]If you do this, make sure that the documentation sections – *synopsis, etymology, motivation*, etc. – are completed for *every* class contained in the file.

is often used. Such implementation files have in general names like *name2.cc* etc. See *code/aips/implement/Arrays/Array2.cc* for an example.

The test program(s) shall exercise every member function, every exception, and cover at least 90% of the lines of code in the implementation file.[6]

2.2 Include files

Whenever possible the header files should use forward declarations[7] in stead of include files. Include files should be reserved for the implementation files.

2.3 Data Types

Only the standard AIPS++ data types [8] shall be used, coded in the AIPS++ way.

2.4 Modules

Closely related classes should be organized into a module – which requires an appropriately named, separate directory.[9] Modules are assigned by the AIPS++ system manager. The module will have a "module header file" – whose primary purpose is to simplify things for the application programmer by providing sufficient help information for the general use of the classes in the module. Whenever possible, application programmers should not be required to learn the intricacies of a complicated module, nor to understand specific implementation classes within the module. In these cases they should only have to include a single header file into their application program. In cases where the role of the module is more a conglomerate of related, but independent, classes, individual header files should be used.[10]

The module header file [11] will live in the module's parent directory: *code/aips/implement/Tables.h*, for example, is the module header file for, and describes the classes in *code/aips/implement/Tables/*.

2.5 Global Functions

It is sometimes appropriate to write global functions to act on class objects, rather than creating class member functions. As a general rule, you should do this if there is no

[6]Special language tools can measure coverage. See AIPS++ Note 170, "The AIPS++ Code Review Process" for suggestions on writing test programs.

[7]Explicitly or implicitly like in *iosfwd* or *Complexfwd*

[8]Char, uChar, Int, uInt, Float, Double, lDouble, Complex, DComplex, String. In special circumstances the use of Long and uLong is allowed.

[9]See *code/install/codedevl/template-module-h*. The Tables module, *code/aips/implement/Tables* is a good example to study.

[10]As a rule, it is advisable for application programmers to include in their code the include files for the individual classes being used.

[11]See *code/install/codedevl/template-module-h*

class object state preserved from one function call to the next, and all of the information about the class object/s can be obtained from the public interface. However, for reasons of namespace management it is often advised if there exists a strong binding between the class and the global function, to make the global function a static member of the class; or use the namespace keyword. There are two places where global functions may be declared and defined: as part of the *.cc* and *.h* files of the class they are most closely associated with, or in their own, separate *.cc* and *.h* files. Use these criteria to decide:

1. If there is a large conceptual distance between the functions and the class, then it is probably best to use separate files.
2. If there are a large number of related global functions, use separate files.
3. If the functions are templated, using separate files (at least separate implementation files) will reduce template instantiation dependencies.

2.6 Templates

Template instantiations are done explicitly within AIPS++. Make sure templates needed by your code are defined in the package *templates* file in the *_Repository directory.* Templates needed only in your test or demo programs should exist in your *test/templates* file. See the *System Manual* for details about template management.[12]

2.7 File Size (∗)

Files should rarely – if ever – exceed 2000 lines: a 1000 line maximum will usually apply. Well-designed classes often have short header and, certainly, short implementation files, usually less than 500 lines including documentation.

2.8 Function Length (∗)

Functions should rarely exceed 100 lines in length; shorter, well-focused functions should dominate, and will usually be less than 50 lines.

3 Documentation and Naming

3.1 Documentation in Header Files

Header files shall contain clear and complete documentation for classes and modules. Templates (or "boilerplate") for these files are *code/install/codedevl/template-class-h* and *template-module-h.* That directory also has templates for other standard kinds of files.

[12]reident, unused, duplicates programs

3.2 Documentation in Implementation Files

Background, usage, and overall design is presented in class header files, and implementation files do not need to repeat them. But non-obvious sections of code do need to be accompanied by explanation, including references to manuals or texts as appropriate. "Obvious" is to be judged from the perspective of a competent programmer, generally knowledgeable – but not necessarily expert – in the program domain. The guiding principle is: do not force those who read your code to spend unnecessary time deciphering it. It will *always* be a net savings of time if you document what you have done, and why you did it.

3.3 Names for Classes, Functions and Variables

All class, function and variable names will reflect a balance between clarity and convenience, with clarity having priorty. Idiomatic abbreviations, acronyms and contractions are discouraged. Names for classes, structures, and enumerated types shall begin with an uppercase letter. Identifers which consist of more than one word (a recommended practice) shall use uppercase at the beginning of each new word ("*GoodClassName*"). Local variables, class data members, class member functions, and global functions names shall begin with lowercase letters ("*goodVarName, goodFunctionName*"). Class data members shall have names prepended by *its*, or postpended with *_p* ("*itsCount, blockPointer_p*").

Note: Glish function names follow a different policy: they shall use lower case letters exclusively.

3.4 Names for Files

The only restrictions here are commonsense, and the Unix library archive utility "*ar*". It is strongly suggested that the file name equals the name of the one or dominant class in the file. The above suggests that file names should be long enough to convey meaning to the reader, but not so long as to be a burden to type. This translates to: file names should be from about 8 to about 30 characters long.

"*ar*" requires that all object file names be unique in the first 14 characters.

Underscores are not allowed in file names.

Any file (a shell script, or a C++ program, for example) which can be invoked at the command line as an executable program, must be named with lower case chararacters only. (This policy follows the glish function naming policy of the previous section; both are designed to present the end user with only lower case commands to type, but note that underscores are prohibited in file names.)

Use the standard extensions (*.c, .cc, .h, .f, .g*).

3.5 Format for Dates

Programmers (and code reviewers) must enter dates when documenting code. These dates indicate, for example, when code has been reviewed, and when a list of "to do" tasks was last updated.[13] AIPS++ uses a single mandatory format for expressing dates. It has the virtue of being unambiguous, it is comprehensible to readers around the world, and it is also used by the project's version control system *RCS*. This format is *yyyy/mm/dd*. An example is *1995/02/27*.

4 Coding

4.1 Forward Declarations

#include only those header files your class absolutely requires. Use forward declarations when that will suffice.[14] ||

4.2 Protected Data Members (∗)

Avoid protected data members, using protected member functions for access to private data instead.

4.3 Access to Private Data

Do not return pointers or references to private data, except to accomodate Fortran or C libraries, or if efficiency absolutely requires it.[15]

4.4 Label All Virtual Functions Explicitly

Virtual functions in a derived class shall be explicitly labelled with the *virtual* keyword in the class declaration, even though this is not required by the language.

4.5 Document Loose Ends

Indicate unfinished or questionable code with the documentation extractor's tag <*todo*>.

[13] See, for instance, the documentation tags *reviewed* and *todo* in *template-class-h*.

[14] This may sometimes involve the use of special forwarding include files like iosfwd.

[15] Or declare the references and pointers const – but be aware of threading problems in that case

4.6 Standard Class Member Functions

Provide definitions for all of the standard member functions which the compiler would (otherwise) generate automatically: default constructor[16], destructor[17], copy constructor, and the assignment operator. For any of these which you really wish to disallow, declare them private, and create minimal implementations (if any at all). For example:

```
...
private:
   SomeClass () {;}   // disable the default constructor
...
```

4.7 The Order of Function Arguments

Functions shall be declared so that their arguments (parameters) appear in this order: output parameters first; input/output parameters next; input parameters last.[18] Functions which return only one value should usually use the function's return-value to return that value, rather than add an additional output parameter to the parameter list. If a function modifies its single argument *in place*, then that argument is best understood as an input/output argument.[19]

4.8 Formal Arguments and Default Parameters

Specify formal arguments and default parameters in the function declaration; use the same names in the function definition.[20]

4.9 One argument constructors

Constructors with one argument should be made explicit, or have an explanation why automatic conversion is wanted (as an example: *Unit(String)* is an obvious candidate for automatic conversion, to be able to write e.g. "km/s" rather than Unit("km/s")).

4.10 Return Types

Specify explicit return types for all functions. Return an appropriate integer value from *main*.

[16]Note that this is only created when no user constructor at all present

[17]always supply a virtual destructor in a base class

[18]This order may seem unnatural, but it is a direct consequence of the C++ rule that default parameters appear *last* in the parameter list.

[19]But also consider *strcpy* from the standard *C* library, which both modifies an argument, and returns that same value as the function return value.

[20]In the case of a "placeholder" parameter that is not yet used in the definition, omit the name *in the definition* to stop warnings.

4.11 const Functions

All functions that do not change the state of the class should be declared *const*. If only the "hidden" state (like a cache) changes, the function should be declared *const*, but with a *mutable* cache variable. *mutable* should be used carefully.

4.12 Compound Statements

Multi-line compound statements should have braces. E.g.

```
if (condition) {
   statement1;
} else {
   statement2;
};
```

4.13 Template arguments

Minimize the number of template arguments by utilising the *typename* construct for user classes, and the *traits* methodology for fundamental types.[21]

4.14 Casting

Only C++ style casting is allowed. The use of *reinterpret_cast* is not allowed. Allowed are: *static_cast*, *const_cast* (but be aware of what you are doing) and *dynamic_cast*.

4.15 Macros

(∗) Macros (especially *#if* type, should be limited to the *include guard*. Other cases should be checked with the QAG, who often can point out other solutions.

4.16 Exceptions

Exceptions should be reserved for truly exceptional circumstances, and not used to replace status flags or condition tests. Exceptions should be documented with the <thrown> tag (and maybe with the thrown declaration specifier).

4.17 Compiler Warnings

All code should compile without producing warnings from the compiler. Most compilers do not produce many spurious warnings any more.

[21] A good example can be found in the MeasConvert class