AIPS++ Programming Standards
DRAFT Version

Darrell Schiebel

5 August 1992

1 Introduction

In this document, will the discuss the more basic standards and suggestions for *C++* development. The conventions of Plum and Saks will be used for terminology [1]; a standard is a rule which is nearly always followed, and a guideline is a recommendation. Wherever possible, a coding example will be provided to clarify the current discussion, and a justification will be given for all standards and guidelines.

This is intended as a living document, and as such all of its contents are open to discussion and debate. Additions can be added if it can be successfully argued that they will help the maintainability, understandability, or reusability of AIPS++ software, or sections can be removed it can be argued that their benefits do not significantly increase the quality of AIPS++ software. It is very important to maintain a minimal set of standards to avoid a overwhelming barrage of rules which could result in the standards not being followed.

There will be two documents which will accompany this document. One will describe the format for source code documentation and the tool(s) for extracting the documentation and creating *texinfo* files or *man* pages. Perhaps, at some point, the coding standards and documentation standards will be combined, but for now they will be separate. The second will describe the exception mechanisms which will be used for signaling and handling errors.

There are several other sources of information utilized in creating this document. Meyers book on effective *C++* programming proved useful [2], but many of the points which this book makes is also in the "C++ Programming

[1]Plum, Saks, C++ Programming Guidelines, Plum Hall, 1991

[2]Meyers, Effective C++, Addison Wesley, 1992

Guidelines". However, the "C++ Programming Guidelines" is much more terse. The ARM [3] was also used. Since the ARM is the ANSI draft standard for *C++*, it is the *last word* when questions arise, and it also contains interesting commentary on language features.

2 Standards

2.1 `0` and `NULL`

The integer constant `0` should be used instead of the usual `NULL` from `<stdio.h>` or `<string.h>`. In *C*, this `#define` was used to insure that pointer assignments had the proper size. In *C++*, however, this can result in type mismatch problems, e.g. `NULL` may be defined to be `(void *)0`. Zero, "`0`", holds a special place in the *C++* type system because even though zero's type is `int`, it is automatically cast to any pointer, char, or float. (See p.30 Plum, p.87 Meyers, p.35 ARM)

2.2 Header Files

Every include file must have protecting `#if defined()`s to prevent multiple inclusion, so for "aips.h" the include file would be enclosed by:

```
\#if !defined(AIPS\_H\_)
\#define AIPS\_H\_

           BODY OF INCLUDE FILE

\#endif
```

Header files should not initialize values. The user of a given header file should be assured that including the header file will not increase the object-code size. This practice forces the data to be owned by a given source file, and avoids complicated initialization constructs in the header files. (See p.154 Plum)

2.3 `const`, `#define`, and macros

Constants, `#define`s, and macros, should have the smallest scope possible; restricted to a single source file when possible to avoid name clashes. The *C++ const*, *inline*, and *enum* facilities should be used as opposed to defines,

[3]Stroustrup, Ellis, The Annotated C++ Reference Manual, Addison Wesley, 1990

because these facilities provide a typesafe means of defining constants, as opposed to the preprocessor which defeats the type system, and can make macro bugs difficult to find. Sometimes, however, `#define`s must be used, for example include file protection against multiple includes.

Another example of preprocessor utility is the `##` operator for concatenation of tokens. This has no equivalent operation in *C++*. In addition, `#ifdef`s also provide the only reasonable means of conditional compilation. e.g.

```
\#if defined(\_\_cplusplus)
           C++ CODE
\#elif defined(\_\_STDC\_\_)
           ANSI C CODE
\#else
           K\&R C CODE
\#endif
```

With the exception of include file protection, preprocessor functions should be used with caution. The `##` operator, for example, is not available on K&R *C* compilers. The typical work around is to define a macro which uses an empty comment, `/**/`, for concatenation. Careless `ifdef`s for conditional compilation throughout code make it difficult to maintain.

When the preprocessor is utilized macro or defines which begin with a double underscore, "__", or single underscore, "_", should be avoided because are used by *C* and *C++*.

2.4 Memory Allocation

`new` and `delete` will be used for memory allocation and deletion instead of the malloc/free family of functions because, for objects, new and delete allow for destructors to clean up storage allocated by the object. In addition, the use of new and delete allows redefinition the global new and delete, `::new` and `::delete`. By redefining these global operators, allocation/deallocation inconsistencies can be tracked down to the source code line, with the help of the preprocessor. Thus new and delete allow for finer control of the memory allocation process for all dynamic data. (p.18 Meyers) However, this method is useful primarily for testing, and no code should be checked into the system with `::new` or `::delete` defined because defining these affects the whole project. Modifications to `::new` or `::delete` must be approved by the various member of the consortium.

2.5 Name Space Management

To avoid name space conflicts, global variables should be avoided. A better approach is to use static member variables of a class. To avoid type clashes we will use the method which was used in InterViews [4] A macro will be defined,

```
\#define \_lib\_aips(name) aips\#\#name
```

which will create a token unique to the aips project. This will be generated by sed/egrep which will pull out class definitions and generate the defines for the class name. So a define will be generated for each class name, `#define MyClass _lib_aips(MyClass)`. Thus aips scope can be entered by including a given file, `aips-enter.h`, and aips scope can be exited by including a different file, `aips-exit.h`.

Thus to enter an AIPS++ code section the programmer would include the file `aips-enter.h` and this file would be automatically generated from the other AIPS++ header files and might look like:

```
// No multiple inclusion protection to allow reentering AIPS++
//    scope in the same file, file contains only \#defines.

\#define String \_lib\_aips(String)
...MORE SIMILAR DEFINES...
```

Thus every time a developer referenced a member of class `String` the class name would be replaced with `_lib_aips(String)` which would expand as described above to `aips##String` and finally to `aipsString`. Then to exit the scope of aips code the developer would include file `aips-exit.h` which is also automatically generated. This file might look like:

```
\#undef String
...MORE SIMILAR UNDEFS...
```

Thus each reference to `String` would now reference a string defined possibly in some other class library. Hopefully, this sort of context switching will be the exception rather that the rule, and developers will simply include `aips_enter.h` at the top of a source file and not have to worry about name clashes.

In addition, the use of identifiers which begin with a double underscore, "__", or a single-underscore, "_", should be avoided. Identifiers which begin with a double underscore are reserved for *C++* and those which begin with a single underscore are reserved for *C*.

[4]Linton, et. al., *InterViews 3.0* Class Library

2.6 Gotos

The use of gotos in code should be avoided. This well used standard is especially applicable in object oriented design. Gotos in code make the code much more difficult to read and violate logical scoping. (See Plum, p.126)

2.7 Switch Statements

Switch statements should be used when applicable. However, designs which rely upon switch statements based on the *runtime type* of objects should be avoided. Occasionally, this may be the only solution, but it should be cause of reevaluation of the design.

3 Guidelines

3.1 Encapsulation

Object operations should be accomplished via member functions. Users of the object should have no access to the data which the object contains and the data contained in the object should be as protected as possible. In addition, member functions should not return pointers or references to any internal protected data.

3.2 Comments

The format of comments is described in another document(work in progress). However, it is often useful to use the *C++* style of comments, `//`, as opposed to the *C* style of comments, `/* */`. By using the *C++* style, whole sections of code can be commented out using *C* style comments while debugging a section of code, without worrying about nested *C* comments.(See Meyers p.16) Of course another technique for *commenting out* a section of code is to put an `#if 0` around it. However, here again care must be taken not to begin or end the `#if 0` inside a *C* comment.

3.3 Functions

3.3.1 Parameters and Return Values

Pass and return large objects by reference instead of by value, although sometime objects *must* be returned by value. This allows for much more efficient invocation of functions because when objects are passed by value, the constuctors for the object are called to create a duplicate copy of the object. This can be an expensive operation. A reference to an object should be returned whenever it is desirable for the return value to be used as an lvalue. For example, it might be nice to have an `operator[]` for an array class. In this case, the return type of the `operator[]` should be a reference, so that operations like `A[2] = 9` can be performed. Likewise the `operator=` should return a reference so that `a = b = c` is possible where a, b, and c are some objects with the `operator=` defined.

However, sometimes an object must be returned as opposed to a pointer or reference to a object. For example, when defining an addition operator for complex numbers:

```
class Complex \{
  friend Complex operator+(const Complex \&,const Complex \&);
  public:
    Complex(float x=0, float y=0) : r(x), i(y)\{\};
  private:
    float r,i;
\}

inline Complex operator+(const Complex \&a, const Complex \&b)\{
  return(Complex(a.r + b.r,a.i + b.i));
\}
```

The important thing to note is that the `Complex` constructed in `operator+` is destroyed as a temporary in expressions like `d = a + b + c` where a, b, c, and d are all instances of the `Complex` class. If the storage were dynamically allocated within the `operator+`, there would be no way of freeing the storage generated in the `a + b` sub-expression above (See Meyers, pp. 78-84).

3.3.2 Operators

If you want conversion to happen on both the left and right hand sides of an operator make the operator a friend instead of a member. For example, if we extended the previous example to provide a `operator-` member function, we would end up with a class that looks like:

```
class Complex \{
  friend Complex operator+(const Complex \&,const Complex \&);
  public:
    Complex operator-(const Complex \&b){ return(Complex(r - b.r, i - b.i));};
    Complex(float x=0, float y=0) : r(x), i(y)\{\};
  private:
    float r,i;
\};

inline Complex operator+(const Complex \&a, const Complex \&b)\{
  return(Complex(a.r + b.r,a.i + b.i));
\}
```

Both of these classes would work with expressions like `c = a + b` or `c = a - b` where a, b, and c are instances of class `Complex`, but in the case of, `c = 100 + a;` and `c = 100 - a`, only the `operator+` would succeed because the constructor with defaults would be applied to `100` to produce a temporary of type `Complex`. In the case of `operator-` the compiler has no way of knowing to which type the `100` should be converted (See Meyers, 66-71).

3.3.3 const and functions

Use const parameters for functions whenever possible, use const pointers and references as return values to avoid the cost of object creation, and use const member functions whenever the function does not modify the object. Using the `const` specifier allows for further specification of the type signatures of functions. By declaring the parameters as `const`, the function accepts a wider number of arguments, i.e. both const and non-const arguments. By declaring the function as const, the potential users of the class are told that they can call the function without worrying about modifications to the object. By returning const pointers and references, the integrity of the object is maintained without the cost of copying a portion of the object.

3.4 Parameterized Types/Templates

Parameterized types should be used to express a family of types. The most obvious use is for parameterized container classes which will contain heterogeneous data, e.g. a linked list. "Do Not use polymorphism (derived classes with virtual functions) to implement parameterized types."[5] Thus heterogeneous objects should not be derived from a common class simply for the purpose of making a container class or whatever from the common class. The primary reason this is problematic is that the heterogeneous objects are free to mix in the container class, and the only way to maintain a homogeneous collection is by runtime type checks. Of course, sometimes this is the desired behavior.

[5] Plum, p.60

3.5 Object Interface

The object interface should be a minimal interface, particularly the public interface. Members should be as protected as possible, and only the members designed for users of the class should be public. In addition, if possible the members should be private, and only when they are intended for use by derived classes and not for object users should they be made protected.

3.6 Casts

Although some times type casts are necessary, they should, in general, be avoided. The presence of casts undermines the *C++* type system. Each cast should be clearly documented. Also, the casting away of const variables and objects should be avoided. In fact, attempted modifications to a const object can either cause an addressing exception or modify the object as specified. The choice is implementation dependent[6]. Downcasts from a virtual base class are always illegal. Sometimes, casts can be eliminated by using virtual functions in the base class to perform the necessary operations. Whenever possible the burden of performing operations specific to derived classes should be shifted from a case statement on "object type" to virtual functions and inheritance.

3.7 Nested Types

Advantage can be gained by using nested type to avoid name conflicts in the global name space. The type names can be made local to the class which utilizes them. For example:

```
class A \{
 public:
  enum data \{ IN, OUT, UP, DOWN \};
  data getData()\{return x;\}
  void setData(data z)\{ x = z;\}
  A() : x(IN)\{\};
 private:
  data x;
\};

main()\{
  A a;
```

[6]ARM p.71

```
  A::data t;
    t = a.getData();
    a.setData(A::UP);
\}
```

Here `data` is a type local to the class `A`, but since it is public it can still be used as a type name and the elements of the enumeration can be accessed by prefacing them with the class specifier, `A::`. The elements of the enumeration, `IN, OUT,` etc. must be unique, i.e. there could not be another enumeration with any of the same elements as the `data` enumeration within `A`.

3.8 Error Handling

Exceptions will provide the primary means of flaging and handling errors. An exception class, will be built for general use. It will be based upon the OpenSys Inc., *Exc C++ exception enabling library* and the 1988 Usenix paper on exceptions [7].

3.9 IO Streams

It is often advantageous to use the `<iostream.h>` package because it is extensible. That is the designer of a package can design functions to print out his/her classes.

3.10 Library Names

To identify members of libraries, e.g. math, system, etc., the classes or stand alone public functions belonging to these libraries should be prefixed with one ore two letters to denote their membership to the library. Thus, for example:

```
class mSin \{
  ...etc...
\};
```

would belong to the AIPS++ math library. In general, mixed case will be used to distinguish words within an identifier, e.g. `MyLongIdentifier`.

[7]Miller, Exception Handling Without Language Extensions, 1988 Usenix Proceedings, C++ Conference, pp.327-341

3.11 Format

Many feel that it is important to have a format guidelines. These are standards which make the code "more readable" by having a indicating how control structures and code blocks should look. There are three primary standards [8]. Of these, the Kernighan and Ritchie format is the control structure format for AIPS++. So for example:

```
main(int argc, char *argv[])\{
  char *x;
  for (int i=1; i < argc; i++) \{
    int j = i+1;
    while (j < argc) \{
      if (strcmp(argv[j],argv[i]) < 0) \{
        x = argv[i];
        argv[i] = argv[j];
        argv[j] = x;
      \} else \{
//      argv[i] = argv[i];                       // Format of else
      \}
      j += 1;
    \}
  \}

  for (i = 0; i < argc; i++) \{
    cout << argv[i] << "\n";
  \}
\}
```

4 Warnings

4.1 Machine Size and Ordering Dependencies

4.1.1 Size of Builtin Types

Concerning the sizes of the standard integral types the following are guaranteed [9]:

- `1 == sizeof(char) <= sizeof(int) <= sizeof(long)`

[8]Plum p.181
[9]Stroustrup 2ed p50

- `sizeof(float) <= sizeof(double) <= sizeof(longdouble)`
- `sizeof(`*t*`) == sizeof(signed `*t*`) == sizeof(unsigned `*t*`)` where t is one of the basic types

Additionally, the following are the minimum sizes guaranteed for the integral types [10] and the floating point types [11]:

- `char` — 8 bits
- `short` — 16 bits
- `long` — 32 bits
- `float` — 32 bits
- `double` — 64 bits
- `long double` — 96 bits

The size of an `int` is typically 16 to 32 bits, but this size is not guaranteed (32 bits in all probability for machines running AIPS++). The size of the integral and floating point types are available in the header files `<limits.h>` and `<float.h>` respectively. (see p.72 Plum)

In addition, dependencies on the byte ordering of numeric types should be avoided. For example, the following is a bad idea:

```
short i = 25;
char *cptr;

cptr = (char *) \&i;
```

One cannot anticipate the byte to which `cptr` points, given the possible difference in architecture byte orderings (p.74 Plum). In addition, when passing integral values between machines, the values should be converted to/from network byte ordering via the routines in `<netinet/in.h>`, e.g. htonl, etc.

[10]Stroustrup 2ed p50
[11]ARM p.24

4.1.2 Structure Size and Member Offset

Another possible source of alignment errors, is hard-coding offsets into structures because the packing of structure members into the storage is implementation dependent. The proper way to determine the offset is to use the `offsetof` macro in `<stddef.h>`(see p. 84 Plum). So for a struct like:

```
struct mystruct \{
  int x;
  int y;
\};
```

`offsetof(mystruct, y)` might yield 4.

4.1.3 Character Constants and String Literals

Character constants should contain only one character because differences in machine byte order may lead to values which differ in numeric value of character sequence, for example:

```
short crlf = '\r\n';
```

is a non-portable use of character constants. A literal string can be used if appropriate, or a left shift can be used to portably generate a integral value out of characters, e.g.

```
\#define CHAR2(a, b) (((a) << CHAR\_BIT) + (b))
```

portably combines two characters [12].

String literals could also be a source of problems. They should not be modified instead named arrays should be used. In the following example, the first method is the portable way to obtain a modifiable string; the second is the wrong way [13]:

1. `static char fname[] = "/tmp/edXXXXXX"; mktemp(fname);`
2. `mktemp("/tmp/edXXXXXX")`

[12]Plum, p.76
[13]Plum, p.86

4.2 Creation and Deletion

4.2.1 Initialization

It is important to remember that global static objects which have a constructor or are initialized with a constant expression are initialized in the order in which they are encountered [14]. There are ways to ensure a certain initialization order [15].

Another possible source of problems is with the initialization of member variables in a constructor definition. The member variables are initialized in the order in which they occur in the class, regardless of their order in the constructor definition. So for example:

```
class A\{
  private:
    int x;
    int y;
    int z;
  public:
    A() : z(12), y(9), x(z+y) \{\};
    A(int t) : x(8), y(t), z(0) \{\};
\};
```

will cause problems, because its order of initialization will always be — x, y, then z. This choice was made to have consistent initialization orders. Otherwise, if the initialization order was dependent upon the order in the constructor definition, the two constructors in this example would have different initialization orders [16].

In general, it is best to avoid designs which depend on the initialization order of static globals or static members. Also, it is important to remember the initialization order of member variables to avoid problems.

4.2.2 Deletion of Arrays

When deleting an array of objects, it is important to call the delete operation with the array specifier. Otherwise, the destructors for each of the elements in the array will not be called. Only the destructor for the first one will be

[14] Plum, p.138

[15] Schwarz, "Initializing Static Variables in C++ Libraries", C++ Report. Vol1 No2, Feb. 89, pp1-4.

[16] Meyers, p.42

called. So for example, if we have a string class which allocates memory to hold the string:

```
class String\{
  private:
    char *rep;
  public:
    String(char *);
    String()\{ rep = 0;\}
    const char *operator*()\{ return rep;\}
    String \&operator=(char *);
    ˜String()\{ delete rep;\}
\};

String::String(char *v)\{
  if (v != 0)\{
    rep = new char[strlen(v)+1];
    strcpy(rep,v);
  \} else
    rep = 0;
\}

String \&String::operator=(char *newval)\{
  if (rep != 0)
    delete rep;
  rep = new char[strlen(newval)+1];
  strcpy(rep,newval);
  return(*this);
\}

main() \{
  String *s = new String[3];

    s[0] = "Hello";
    s[1] = "there";
    s[2] = "friend";

    delete s;
\}
```

the call to `delete s` only causes the destructor for the first `s[0]` to be called. The space allocated for `there` and `friend` is lost forever. The correct way to delete `s` is `delete [] s` [17].

Another interesting point is that for an array of objects to be allocated via `new` a *default constructor* has to be defined, i.e. a constructor without parameters [18]. Specifying a constructor with all default parameters is not sufficient.

4.2.3 Virtual Destructors

Sometimes it is important for destructors to be declared as virtual in a base class. This is important when all of the following hold [19]:

- There are classes derived from the base class.
- The destructor in the derived class is different form the base class destructor.
- Derived class objects may be deleted via a pointer or reference to base class objects.

It is important for the destructor to be virtual when these conditions hold because when deleting a derived object via a pointer to the base class, only the destructor for the base class will be called. The derived class' destructor will not be called, thus possibly causing memory leaks.

4.2.4 References

Another cause of memory problems is references. It is generally a bad idea to initialize a reference to memory which can be deleted, e.g. free store, automatic variables. Typically this problem will result in segmentation violations or the like, and not in memory leaks. However, a substantial amount of time could be expended tracking down the source of error (see p.38 Plum).

4.2.5 New and Delete

Some problems can arise with classes which define the operators `new` and `delete` with more than one argument because these definitions hide the global or base class definitions. For example [20]:

[17] Meyers, p.19
[18] ARM p.61
[19] Plum, p.208
[20] Meyers, p.25

```
typedef void (*PEHF)();

class X \{
  public:
    void *operator new(size\_t, PEHF pehf);
\}

main()\{
  void specialErrorHandler();

  X *px1 = new (specialErrorHandler) X;
  X *px2 = new X;
\}
```

The first `new` succeeds with no problems. The second however causes an error because the global `new` which takes one parameter, `size_t`, is hidden by the local `new` in class `X`. This problem can be corrected by either calling the global new explicitly, `X *px2 = ::new X`, or by providing a `new` operator in `X` which takes only one parameter. It is also a good idea to supply both `new` and `delete` if one is needed to avoid future maintenance problems.

4.3 Copy Operator

Be aware that if you don't explicitly disallow the copy operator, ***Type***`::operator=(`***Type***`&)`, one will be supplied for you, i.e. a bit copy. One way to ensure that there is no copy operator is to make its declaration private, preventing users from accessing it, and not providing a definition, preventing `friends` from using it.