Revision 4.0 1994/08/26 05:07:17 aips2adm # exhale: Base release 04.000.00 # # Revision 3.0 1993/11/01 04:20:54 aips2mgr # exhale: Base release 03.000.00 # # Revision 2.0 1993/03/16 08:17:14 aips2mgr # exhale: Base release 02.000.00 # # Revision 1.1 1992/04/09 19:53:50 bglenden # Initial revision

Class Documentation Standards

for the AIPS++ Project
(Pre-release Version)

Chris Flatters

This page deliberately not left blank.

Class Documentation Standards

This document contains standards and recommendations for writing reference documentation for AIPS++ classes. The authoritative documentation for a class is assumed to be a commented C++ header file. The standards set out in this document support the use of automated tools which can extract the documentation from the header file and transform it to a form suitable for a typesetting program such as TEX.

These standards comprise mandatory *requirements* and advisory *recommendations*. Requirements and recommendations are written as follows.

Requirement:
you must do this.

Recommendation:
it is advisable to this.

This pre-release edition is being distributed to solicit comments. Please send any comments to `cflatter@nrao.edu`.

1 Basic Principles

Class documentation should provide programmers with all of the detailed information they need in order to be able to use instances of a given class. It should never be necessary for a programmer to refer to the actual implementation of an object; examining implementation code is a tedious and error prone business. If this is to be avoided then the class documentation must be accurate, precise and (most importantly) complete as well as comprehensible.

A C++ class declaration contains a great deal of information about a class that is guaranteed to be reliable. For example, it lists the names of the member functions, their arguments and any exceptions that they may raise[1]. The C++ code alone does not, however supply complete information about a class and must be supplemented by a commentary.

1.1 Contractual Obligations

Programming by contract is a common paradigm in object-oriented programming. The behaviour of any class can be defined by a set of contracts between the class and its clients that take this form.

> I have a member function called X taking formal arguments Y and returning Z. If you ensure that condition P is true and then call member function X then I guarantee to do Q.

The class documentation should be a statement of the relevant contracts: it should say exactly what the class guarantees to do and what the obligations of the class' clients are. There should be no loopholes.

Class invariants, preconditions and postconditions are useful in documenting contractual obligations. A class invariant is a condition that must hold for all objects of a given class, a postcondition is a condition that will hold after a function is executed (representing a responsibility of the class) and a precondition is a condition that must hold before a function is called (representing an obligation on the client). All three are mathematical in nature and are best written in mathematical notation. Unfortunately, many AIPS++ programmers are not comfortable with the mathematics required. This leads to the first recommendation.

Recommendation:

you should use class invariants, preconditions and postconditions and write them using mathematical notation if you are comfortable with these concepts.

However, if these concepts are used, we can not guarantee that all AIPS++ programmers will understand them.

Recommendation:

if you document a class using mathematical conditions that are not immediately obvious to a reader with a working knwoledge of C++ then you should either describe the same conditions informally in plain English or provide them with a descriptive label.

[1] Current versions of C++ do not support exceptions

An example of a mathematical condition with a descriptive label follows.

```
all array elements are zero:
    forall( int i; 0 <= i && i < current.length(); current[i] = 0 )
```

1.2 Writing Mathematics

If mathematical notation is used it must be written in a form that can be easily read and understood in a header file. This rules out the use of mathematical mark-up languages such a TEX. A programmer may assume, however, that the readers of his documentation understand C++ notation.

Recommendation:
write mathematical conditions in a form that is as close to C++ as possible.

C++ notation is rather too limited to be useful on its own and some conventions must be adopted to extend it for documentation purposes. The following notations are recommended.

`result` Use the pseudovariable `result` to represent the return value of a function.

`current` Use the name `current` to refer to the current object; `current` is less clumsy than the `*this` notation that must be used in actual C++ code.

`forall` Write universal quentifications using the template `forall(` *declarations*`;` *restrictions*`;` *condition* `)`.

`exists` Write existential quantifications using the template `exists(` *declarations*`;` *restrictions*`;` *condition* `)`

1.3 Comment Delimiters

Comments that contain documentation must be written in a form that can be easily recognized and extracted by a mechanical processor. All documentation comments have the following form.

```
/*X*
 *
 * Some text
 *
 *X*/
```

X is a single letter code that is used to denote what is being documented in the comment. Valid codes are

`H` file headers

`C` class headers

`F` function headers

`T` constants

`E` named and unnamed enumerations

A documentation extraction program uses the `/*`X`*` and `*`X`*/` delimiters to recognize comments that should be extracted. Comments that are delimited by plain `/*` and `*/` are discarded as are comments that begin with `//`. The leading column of asterisks makes it easy for a human reader to distinguish text in the comment from C++ code.

2 Header File Layout

A header file contains the definition of one or more C++ classes.

Requirement:
header files must conform to the template shown below.

```
/*H* -*- C++ -*-
 *
 * header file description
 *
 * copyright statement
 *
 *H*/

#ifndef identifier
#define identifier

class definition list

inline functions

#endif
```

GNU emacs uses the `-*- C++ -*-` string to distinguish files containing C++ code from those containing C code.

The *header file description* should say what publicly available classes are defined in the file and briefly describe how they are related to one another. The form of the *copyright statement* has yet to be decided at the time of writing.

The *class definition list* is a list of class definitions. Definitions of all *inline functions* must follow the last class definition and will be ignored by documentation extraction tools.

2.1 Class Definition Layout

Requirement:
class definitions must conform to the template shown below.

```
/*C*
 *
 * class description
 *
 * INVARIANTS
 *
 * label: condition
 * label: condition
 *
 *C*/
class name : ancestors
friend class declarations
{
public:
    constant definitions
    enum type definitions
    function declarations
protected:
    constant definitions
    enum type definitions
    function declarations
private:
    constant definitions
    enum type definitions
    function declarations
};
```

Recommendation:

put the `public`, `protected` and `private` parts of a class definition in that order so that a human reader can find those parts of the class definition that are most relevant to him near the top of the definition.

The *class description* should be an informal description of the class in plain English. It should state what the class represents and give a good idea of the responsibilities of the class. If the class has particular memory requirements then they should be described here.

The `INVARIANTS` section is optional. Each *label* should be a short phrase describing the invariant condition and the *condition* should be a mathematical description of the same condition (see Section 1.2 [Writing Mathematics], page 5). The label may be omitted if the meaning of the *condition* is self-evident or both the *label* and *condition* may be replaced by an English phrase if the condition can not easily be written in mathematical form.

2.1.1 Constant Definitions

A class may define one or more constants.

Requirement

constant definitions should conform to the following template.

```
/*T*
 *
 * constant description
 *
 *T*/
C++ constant definition
```

The *constant description* should state what the constant represents and, if the value of the constant does not appear in the C++ definition (ie. if the constant is a `static` class member), then the value of the constant should be included in the description as in the following example.

```
/*T*
 *
 * The mathematical constant pi (3.14159265358978).
 *
 *T*/
 static const double pi;
```

2.1.2 Enum Type Definitions

Classes often define `enum` types to represent such things as states.

Requirement:
Definitions of enumerated types must conform to the following template.

```
/*E*
 *
 * enum type description
 *
 *E*/
C++ enum type declaration
```

The *enum type description* should state, in plain English, what the enumerated type represents. It should also list the symbolic values of the enumeration and what they represent as in the following example.

```
/*E*
 *
 * This enumeration represents the possible status returns from a
 * Stack operation.
 *
 *    ok             operation successful
 *    overflow       stack would have overflowed: stack unchanged
 *    underflow      stack would have underflowed: stack unchanged
 *
 *E*/
enum stackState {ok, overflow, underflow};
```

2.1.3 Function Declarations

Member functions and friend functions require extensive documentation.

Requirement:
Function declarations must conform to the following template.

```
/*F*
 *
 * function description
 *
 * PRECONDITIONS
 *
 * label: condition
 *
 * POSTCONDITIONS
 *
 * label: condition
 *
 *F*/
C++ function declaration
```

The *function description* should describe what the function does, what conditions it will work under, what it returns and what its arguments are. The option `PRECONDITIONS` and `POSTCONDITIONS` clauses repeat parts of this description in a more precise mathematical form. They may be written as a mathematical statement with an optional label (see Section 1.2 [Writing Mathematics], page 5) or in plain English. Here is an example of a function declaration[1].

[1] The `true` precondition indicates that the behaviour of `Stack::pop()` is defined under all circumstances

```
/*F*
 *
 * Pop an item from the stack.  If the operation is completed
 * successfully then the value returned is ok. If the stack is
 * already empty the the value returned is underflow.
 *
 * PRECONDITIONS
 *    true
 *
 * POSTCONDITIONS
 *    (!current.empty() && result == ok)
 *    || (current.empty() && result == underflow)
 *
 *F*/
stackState pop();
```

3 An Example Header File

```
/*H* -*- C++ -*-
 *
 * This file contains the definitions for the Stack class.
 *
 * << Copyright statement belongs here >>
 *
 *H*/

/*C*
 *
 * The BoundedStack class represents a Stack (LIFO) of an arbitrary
 * type and of fixed size.  The number of entries that can be placed
 * on a Stack is given as a template argument when the Stack is
 * instantiated.
 *
 * Objects of type T may be pushed onto or popped from the top of
 * the Stack using push() and pop().  The topmost element on the
 * Stack may be examined using top().  A program can test whether
 * a Stack is empty of full using empty() or full().
 *
 *C*/

#ifndef STACK_H
#define STACK_H

class Stack<class T, int size>
{
public:

    /*F*
     *
     * The default constructor.
     *
     * PRECONDITIONS
     *     true
     *
     * POSTCONDITIONS
     *    stack is empty: current.empty()
     *
     *F*/
    Stack();

    /*F*
     *
     * Return true (a non-zero integer) if the Stack is empty or
     * false (zero) if the Stack is not empty.
     *
```

```
     *F*/
    int empty() const;

    /*F*
     *
     * Return true (a non-zero integer) if the Stack is full or
     * false (zero) if the Stack is not full.
     *
     *F*/
    int full() const;

    /*F*
     *
     * Return the value of the item at the top of the Stack.  The
     * The value returned is the same as the last value pushed onto
     * the stack.  The Stack may not be empty.
     *
     * PRECONDITIONS
     *     stack not empty: !current.empty()
     *
     *F*/
    T top() const;

    /*F*
     *
     * Push the value ton to the Stack.  The Stack must not be
     * full when this function is called.
     *
     * PRECONDITIONS
     *     stack not full: !current.full()
     *
     *F*/
    void push(T t);

    /*F*
     *
     * Remove (pop) the topmost value from the Stack.  The Stack
     * must not be empty when this function is called.
     *
     * PRECONDITION
     *     stack not full: !current.full()
     *
     *F*/
     void pop();

private:
     T implementation[size];    // Array of values
     int max_size;              // Max. number of values on stack
     int cur_size;              // Current number of values on stack
```

```
};

#endif
```

Index

Table of Contents