Note 190: Exceptions Changes for AIPS++

B.E. Glendenning
bglenden@nrao.edu

April 24, 1996

1 Exception mechanism

The C++ exception mechanism provides three features:

1. Transfer of *control* between the point in the program where the the exception is *thrown*, and where it is *caught*.
2. Transfoer of *information* between the "thrown" and "caught" locations. The information which is transferred can be arbitrary, since any object can be thrown. Note that a *copy* of the object is thrown.
3. *Destruction* of automatic objects.

The differences between the AIPS++ exception emulation and the standard C++ mechanism are as follows:

1. Only classes that follow certain rules[1] may be thrown using the AIPS++ emulation.
2. Standard exceptions allow "any" exception to be caught with a single catch statement (using `catch(...)`).
3. Only classes derived from a special base class are destroyed by the AIPS++ exception emulation.
4. The syntax of the AIPS++ emulation is somewhat different. For example, catch clauses must end with `end_try`. These differences can be hidden by a macro.
5. AIPS++ cannot rethrow exceptions.
6. AIPS++ does not support exception specifications.
7. AIPS++ exceptions contain the line number and source file of the location from which they were thrown.

[1]Basically, the classes must have the AIPS++ version of RTTI.

2 When to throw an exception

Exceptions are intended to be used in exceptional circumstances — those that the application programmer cannot reasonably be expected to test for — not as an alternative return mechansim. If nothing else, throwing an exception is expensive: Meyers[2] notes that throwing an exception might be three orders or magnitude more expensive than a normal function return.

As a rule of thumb then, a library routine should rarely have a *catch* clause. Anticipated problems should be handled with normal programming constructs (error returns and the like). The AIPS++ library already follows this rule.

Looging through the *aips* package, an exception is thrown[3] in about 1800 places. Examining about half of these reveals the following reasons that exceptions are thrown:

Argument checking	45%
Internal invariant/state verification	40%
Memory allocation	10%
I/O errors	2.5%
Misc.	2.5%

It seems to me that we are generally throwing exceptions appropriately.

The only errors that a running program might plausibly be able to handle (other than via a graceful exit or a next iteration of a high level loop) are the memory allocation and I/O errors.

3 What exception to throw

The draft standard C++ language and library has the following exception hierarchy (indentation indicates inheritance):

```
bad_alloc
bad_cast
bad_exception
bad_typeid
exception
    logic_error
        domain_error
        invalid_argument
        length_error
        out_of_range
    runtime_error
        range_error
        overflow_error
```

[2] *More Effective C++*, Addison-Wesley, 1996.

[3] Either directly, or through the use of an *Assert* statement in one form or another.

Basically, the first four are thrown by various language features (for example if `new` fails, `bad_alloc` is thrown), and standard library components throw exceptions derived from `exception`.

The AIPS++ library, by contrast has 42 exception classes. There are seven classes alone to describe various bad things that can happen to *Array* classes (conformence error, indexing error and the like).

This seems excessive, given that the running program cannot meaningfully deal with most of the errors. All that is usually wanted from the exception is an error message and information about where the exception was thrown.

There are three obvious approaches:

1. Use the standard exceptions (implemented by us for compilers without standard exceptions — the classes are quite simple).
2. Shadow the exception hierarchy with a similar `aips_*` hierarchy.
3. Shadow the exception hierarchy, using multiple inheritance (so that the same catch could get both the `aips_*` and standard exception).

I prefer the first solution. The only reason I can think of to adopt the latter solutions is to embed line number and source file information. However one can intercept exceptions with a debugger, so I don't consider this advantage convenient enough to warrant the added complexity in the exception hierarchy.

We do need to allow for AIPS++ specific exceptions, so I would introduce an `aips_exception` class derived from `exception`, and perhaps an `aips_io_exception` derived from it. I don't think that yet more derived classes should be forbidden, however their creation should be driven by the programmer who could usefully *catch* such a class.

4 The `what()` string

While it is true that debuggers on systems with native exceptions will allow programmers to intercept exceptions, it is also true that programmers will occasionally have to try to recreate a problem only given an error message provided by the `what()` member function. To make it easy to figure out the originator of the exception, I propose that exceptions that originate from `throw()` but not `Assert*` should contain the name of the function to allow the programmer to discover where the error originated.

A format of:

```
"Class::member() - error message"
"Class::member(args) - error message" // if overloading makes it ambiguous
"::func() - error message"            // global function
"::func(args) - error message"        // global overloaded function
```

Should generally be chosen under normal circumstances.

After some reflection, I have decided not to suggest that the function name be embedded in `Assert` statements — we want to continue to make it easy for programmers to put in checks for "impossible" errors.

5 Proposal

I propose the following changes to the AIPS++ treatment of exceptions.

1. That an exception hierarchy like the one I outlined be created.
2. That existing exception classes gradually be removed during maintenance.
3. That the (potential) *catch*er of an exception determines when a new exception class should be created.
4. That a `#define` be created so that all exceptions may be caught both with our exception emulation and "real" exceptions.
5. That error strings in exceptions directly from a `throw` have a standard format that contains the throwing function's name.