Exception Handling in AIPS++

DRAFT Version

Darrell Schiebel

Last modified: $Date: 2003/07/16 03:48:58 $ By: $Author: aips2adm $

1 Introduction

The main intent of this paper is to provide a user level introduction to the AIPS++ exception mechanism with a basic introduction to the implementation. This should allow developers to use the mechanism.

This package provides a method to handle exceptional conditions. When such conditions arise, control is transferred to a code block which can handle the exception, if one exists. If no handler exists which is capable of handling the exception, the program is aborted. This package handles cleaning up the call stack which may have been created between when the handler was *installed* and when the exception was thrown. In addition, any objects[1] which were created during this period will be deleted.

This provides a mechanism for transferring control from the point where an exceptional condition occurs to the code to handle the exception without memory leaks. Some of the advantages of this approach are:

- The code to display the error message can be moved out to the appropriate level. Otherwise, the code to display an error message might have to be deep within an application in a place which could otherwise be independent of the graphical user interface. Without exceptions error messages must be carefully propagated out via return values until the appropriate level for their display is reached.
- The exception mechanism allows for an incremental graceful exit. The same exception can be thrown a number of times so that each layer of the program can have a chance to respond to the exception as appropriate. This can be important for maintaining the integrity of portions of the program, e.g. the user interface or database portions.
- Using exceptions a program can continue after an error even though the error may have been fatal at the point where it occurs. In a more traditional approach, this would require careful checks of return codes.

So while everything which can be accomplished with exceptions can be accomplished with returned error codes or other traditional approaches, the use of exceptions is cleaner.

[1] The objects must be derived from the `Cleanup` object somewhere in it inheritance hierarchy to be automatically deleted

2 Interface

This section describes the AIPS++ exception mechanism from the perspective of a "user" of the system. It discusses the ARM interface, the mechanisms for automatic deletion of dynamically created objects, and the steps necessary to create new exception types whose objects can be *thrown*.

2.1 ARM Interface

The AIPS++ exception interface conforms to the standards outlined in the ARM. The ARM exceptions are based on the idea of `try` and `catch` blocks. The `try` block contains the code which may throw an exception, and the `catch` block, which for the AIPS++ mechanism must follow immediately after the `try` block, contains the handlers for the various exceptions which may be thrown in the `try` block. So in the simple case, one might have:

```
try {
  if (how_many == 1) throw(MajorError(1));
  if (how_many == 2) throw(MinorError(2));
  if (how_many == 3) throw(TypedError(3));
} catch (TypedError xx) {
  cout << "Caught TypedError" << endl;
} catch (MinorError xx) {
  cout << "Caught MinorError" << endl;
} catch (MajorError xx) {
  cout << "Caught MajorError" << endl;
} end_try;
```

The `end_try` is not part of the ARM standard, but it is required for the AIPS++ mechanism. Also, on some platforms catching a *reference* instead of a *pointer* or an *object* can cause errors as a result of the exceptions implementation. In this example, depending on the variable `how_many` a different exception will be thrown, via `throw()`. The exception will then be caught by the first `catch` block that matches the exception. Thus, if `MinorError` was derived from `TypedError`, then the `TypedError` block would be executed before the `MinorError` block for a thrown `MinorError`.

It is also possible to have incremental recovery. A *caught* exception can be *rethrown* with `rethrow()`. The following example illustrates this usage:

```
try {
  if (how_many == 1) throw(MajorError(1));
  if (how_many == 2) throw(MinorError(2));
  if (how_many == 3) throw(TypedError(3));
} catch (MinorError xx) {
  cout << "Caught MinorError" << endl;
} catch (MajorError xx) {
  cout << "Caught MajorError" << endl;
  rethrow;
} catch (TypedError xx) {
  cout << "Caught TypedError" << endl;
} end_try;
```

In this example, the `MajorError` catch block handles a `MajorError` when it occurs and then *rethrows* the exception so that it could be handled by other catch blocks. Thus, if `MajorError` is derived from `TypedError`, then the `TypedError` catch block would have the opportunity to handle its portion of the error after the `MajorError` catch block was finished.

These `try/catch` blocks can be nested to the level necessary as long as the introductory `try` is balanced with a closing `end_try`. The following example contains a nested try block:

```
try {
  if (how_many == 0) throw(ExcpError(1));
  try {
    if (how_many == 1) throw(ExcpError(1));
    if (how_many == 2) throw(ExcpError(2));
    if (how_many == 3) throw(ExcpError(3));
  } catch (ExcpError xx) {
    throw(ExcpError(9));
  } end_try;
} catch (ExcpError xx) {
  cout << "Caught ExcpError" << endl;
} end_try;
```

2.2 Cleanup Objects

Another important portion of the exception handling mechanism is the section which manages the objects created between the point where the `try/catch` blocks are entered and the point where the exception is thrown. These object are automatically cleaned up when an exception occurs. The destructors are called for dynamically created objects, automatically created objects and objects which are *physically* members of another object. Also, the storage space for dynamically created objects is freed.

However, the user must do a few things to allow this to happen. The objects which are to be cleaned up in the event of an exception must be derived from the `Cleanup` class. Any objects belonging to a class derived from `Cleanup` which were created between the point where the `catch` block that handles the exception is entered and the point where the exception was *thrown* will be automatically deleted.

The class designer must also supply a `cleanup()` function for each of the classes to be cleaned up in the event of an exception. This function should call the class' destructor:

```
class foo: public Cleanup {
  public:
    ~foo() {}
    void cleanup() {this->foo::~foo();}
};
```

`this->foo::~foo()` is the best way for a member function to call its class' destructor. Other methods may fail on some platforms. In addition, one should not rely on `cleanup()` being called; the destruction code should reside in the destructor. If a class which is derived from `Cleanup` fails to provide a `cleanup()` member, then the destructor for that class may not be called.

One important member function which one inherits from `Cleanup` is `makePermanent()`. This function removes the object from the control of the *cleanup* mechanism, thus the object can *survive* future exceptions.

2.3 Creating Exception Types

One of the advantages of the AIPS++ exception mechanism is that one can design a class hierarchy of exceptions. This results in an a great amount of flexibility. The reason for this is that one can *catch* whole groups of exceptions with one catch block, and then handle them as appropriate. One can also catch an exception with a specific handler, and the *rethrow* the exception so that it can be caught with a more general handler.

The root of the exception hierarchy is the class `ExcpError`. This class must be the base class for all new exception hierarchies. All of the exceptions which are part of the AIPS++ kernel, `IndexError`, `AllocError`, etc., are derived from the `AipsError`. Thus, if one wished, all exceptions resulting from the AIPS++ kernel could be caught by one catch block which catches `AipsErrors`, or all exceptions generated could be caught by a catch block which catches `ExcpErrors`. Generally, it is a good practice to derive all exceptions for a given library from a library specific exception class, e.g. `AipsError`, which is in turn derived from `ExcpError`.

To add an exception class, a particular constructor must be supplied, a constructor which takes an `ExcpError` pointer as a parameter. For example, the following is the definition of the `AipsError` class:

```
class AipsError : public ExcpError {
  protected:
    String message;
  public:
    const String &getMesg() const {return(message);}
    AipsError(const Char *str=0) : message(str) {message.makePermanent();}
    AipsError(const String &str) : message(str) {message.makePermanent();}
    AipsError(ExcpError *excp) : ExcpError(excp){
      AipsError *tmp;
      if (tmp = (checked (AipsError *) excp)) {
        _equal = True;
        message = tmp->message;
        message.makePermanent();
      } else {
        _equal = False;
      }
    }
    ~AipsError(){}
};
```

The reason why the constructor which takes an `ExcpError` pointer is a bit opaque is because this constructor is the only hook which is available to the exception handling mechanism to perform a *type comparison* to determine if the exception being caught matches the type of the thrown exception. This comparison is taking place on the line where the *checked* cast is applied[1]. If the cast is successful, then the `_equal` boolean member variable is set to `True`, otherwise it is set to `False`. This flag indicates if the catch clause to which this object belongs should be executed.

One of the **most** important things to note about this example it that it contains a `String` member variable. `String` is derived from the `Cleanup` class, so ordinarily the `message` mem-

[1] See AIPS++ note 150 for more information about the run time type system.

ber variable would be destructed whenever an exception occurred. However, in this case that would be disastrous because the whole point is for the `String` to persist from the point where the exception is thrown to the point where it is caught. The `makePermanent()` function calls removes the management of `message` from the control of the exception mechanism. If this call were taken out, the `message` would be created as part of the `throw()`, but the `message`'s destructor would later be called as part of the exception process, destroying the message before the `catch`.

There are two cases where one must worry about unintended destruction as the result of an exception:

1. objects stored in an error class, e.g. `AipsError` above
2. object pointers stored by another object, where either object is derived from `Cleanup`

In the the first case, the stored object is deleted as a result of the exception process when it *must* survive to be returned for a catch block as discussed above. In the second case, a pointer to an object is saved and if not made permanent the stored object is open to unintentional deletion. For example,

```
Slist<String*> list;
try {
  list.addRight(new String("hello"));
  throw(ExcpError());
} catch (ExcpError x) {
} end_try;
```

In this case, `list` is allocated outside of the `try` block so it will survive any exceptions thrown within the try block. However, the `String` object which is created and stored inside of list in the `try` block will not survive the thrown `ExcpError`; it will be deleted unless steps are taken to prevent its deletion. Unfortunately, after the `try` block `list` still retains a pointer to the now deleted `String` object. The `String` object needs to be made permanent to prevent this from happening.

The `makePermanent` member function is not always sufficient. Take the `IndexError` class as an example,

```
template<class t> class IndexError<t> : public IndexError {
  protected:
    FreezeCleanup f;
    t oIndex;                 // Offending Index
    UnfreezeCleanup u;
  public:
    IndexError<t>(t oI, const Char *str=0) : oIndex(oI), IndexError(str) {}
    IndexError<t>(t oI, const String &str) : oIndex(oI), IndexError(str) {}
    IndexError<t>(ExcpError *excp) : IndexError(excp) {
      IndexError<t> *tmp;
      if (tmp = (checked (IndexError<t> *) excp)) {
        oIndex = (*tmp).oIndex;
        _equal = True;
      } else {
        _equal = False;}}
};
```

There the offending index, `oIndex`, is stored as a value. There are no problems if the type of `oIndex` happens to be `Int`, but if its type is `String` we have problems once again. In this case, `makePermanent` cannot be called because the type, `t`, is not known, and thus it may or may not have a `makePermanent` member function. The `FreezeCleanup` and `UnfreezeCleanup` classes solve the problem in this case. The `FreezeCleanup` object suspends `Cleanup` management until either the `FreezeCleanup` object, here `f`, is deleted or an `UnfreezeCleanup` object is created. In the above example, `f` is initialized suspending `Cleanup` management in the process, `oIndex` is initialized unmanaged by `Cleanup`, and finally `u` is initialized re-enabling `Cleanup` management. Extreme care, must be taken to avoid suspending Cleanup management for the rest of the program. Either a `UnfreezeCleanup` object must be created or the `Cleanup::enable()` function must be called when `Cleanup` management is to be resumed.

If the methods outlined above are not followed when applicable, insidious bugs can result. A common symptom of these bugs is multiple deletion of objects, i.e. once by `Cleanup` and again by the the destructor of a class which survived the exception.

3 Implementation

This section briefly discuses the implementation of the AIPS++ exception handling system. The basic underlying system is based on the design discussed in a paper by William Miller from the 1988 Usenix Conference[1]. In addition, an interface is built on top of this exception handling mechanism which is compliant with ARM style exception handling mechanism described in the previous section.

3.1 Supporting System

The underlying system supports both resumptive[2], and non-resumptive exceptions. The semantics of this system are that exception handlers are *installed* at various points, and these handlers can process exceptions which occur in their block scope. These handlers when presented with an error can:

- (abort) – handle the exception and continue with the block where the exception handler was installed (ARM style)
- (retry) – handle the exception, correct the error, continue from the point where the exception was thrown.
- (abdicate) – handle the exception if necessary, and then allow it to propagate to other handlers (ARM's `rethrow()`).

This mechanism extends the ARM proposal by the ability to continue from the point where the exception occurred.

All of the context switches which take place are performed using *setjmp* and `longjmp`. These standard C functions allow for saving and restoring the call stack and registers[3].

The object deletions are managed by maintaining two stacks. The first stack contains all of the installed exception handlers; pushed on as they were created. The second stack contains all of the `Cleanup` objects which are created, including the exception handlers. Each exception handler is given the chance to handle the exception, until a handler chooses to *abort* or *retry*. If a handler chooses to abort, objects are popped from the stack of `Cleanup` objects and the objects deleted until the handler which threw the exception is encountered. At this point, `longjmp()` returns control to the point where the handler was installed. This is the process which takes place to service ARM style exceptions.

3.2 ARM Interface

The ARM interface is a thin layer atop the exceptions described previously. Much use is made of the standard *cpp* `#define`, to implement the `catch` and try blocks. Beyond this, it is mainly bookkeeping. The important pieces of information which are maintained are:

- The *thrown* exception.

[1] W. Miller, "Exception Handling Without Language Extensions", 1988 Usenix C++ Conference Proceedings

[2] Here *resumptive* implies the ability to continue from the point where the exception was thrown, rather than continuing from the point where the exception was caught.

[3] Floating point registers are not typically saved.

- Which `catch` blocks have been executed. This is important for a `rethrow()`. The *cpp* standard macros `__FILE__` and `__LINE__` are useful here.
- The exception which was last thrown, again for `rethrow()`.
- An indicator to signal if an exception is *uncaught* to allow exit.
- The file and line where the exception was thrown for a descriptive error message when an exception goes *uncaught*.
- The *true* type of the uncaught exception, obtained via RTTI
- An indicator to signal if an exception has been rethrown to allow special processing in the exception handler.

This is most of the information which is maintained to provide an ARM compatible exception interface with descriptive error messages when an exception goes unhandled. The following is a typical error message:

```
Uncaught Exception(aips_MinorError_):
    File           - test.C
    Line           - 44
    Throwing Class - trying
```

All of this information is collected "automatically" through the use of preprocessor macros. The *Throwing Class* information is collected via an extra RTTI member function, `aips_typeName__()`, which was added by the RTTI expander for use by the exception mechanism.