1 Introduction

This document discusses the run time type identification system (RTTI). This system was implemented using the T1 C preprocessor. It implements the standard proposed by B. Stroustrup and D. Lenkov\textsuperscript{12}. However, the implemented system also provides “hooks” for much more extensive type information.

We will first discuss the “sanctioned” interface. This is the basic system proposed by Stroustrup et. al. with minor syntactic changes to ease implementation. Stroustrup proposes the concept of a “checked cast”, and a type comparison operation. The last section will discuss the functionality beyond what was proposed by Stroustrup.

\textsuperscript{1} Stroustrup, Lenkov, "Runtime Type Identification for C++", The C++ Report, March-April, 1992, pp.32-42

\textsuperscript{2} Stroustrup, Lenkov, "Run-Time Type Identification for C++ (Revised)", Usenix C++ Proceedings, Aug. 1992, pp. 313-339
2 Sanctioned Interface

This interface provides a mechanism which allows the user to safely cast a pointer. If the pointer is coercible to the indicated type then a valid pointer is "returned" otherwise a NULL pointer is "returned". Stroustrup's proposed syntax\(^1\) is similar to the syntax used for any cast in C++:

```cpp
class A {virtual void dummy();};
class B : public A {};
class C : public A {};

A *a = new B;
B *b = (? B *)a;
C *c = (? B *)a;
```

With respect to Stroustrup's system, \(b == a\) and \(c == 0\). The virtual function \(A::dummy()\) is necessary because run time type information is only generated for classes if they have a virtual function table\(^2\). This is necessary because the virtual table and the virtual mechanism provides the ingredients necessary for the construction of a run time type system:

1. A place to store a function pointer as a hook for each object to return its type information.
2. A mechanism to access the object at the "end" of the inheritance chain.

So in this manner, one can perform a "checked" cast on a pointer, and perform the necessary comparisons to downcast a pointer from a base object to a pointer to the desired derived object.

The implementation for AIPS++ follows this standard closely. The only difference is the syntax. The above example would look like:

```cpp
rtti class A {virtual void dummy();};
class B : public A {};
class C : public A {};

A *a = new B;
B *b = (checked (B *) a);
```

\(^1\) Usenix C++ Proceedings, Aug. 1992

\(^2\) In the general case, this is nontrivial to determine (expound latter)
C *c = (checked (B *) a);

The rtti qualifier was added to underscore the work that was going on behind the scenes. The run
time type information will be added only if the class is prefixed with rtti and if the class
has a virtual function. The rtti qualifier can simply be #defined to nothing if run time type
information should become available as a language feature. The "?" syntax would be difficult
to support in our implementation mainly because it is implemented using a preprocessor. The T1
preprocessor operates by pulling out sections of the code between a starting delimiter, character
string, and an ending character. Thus, the closing paren is necessary to allow the preprocessor to
process both the casting type and the variable which needs to be cast. In any event, these checked
casts can easily be converted to the ARM style with a simple sed script.

If a checked cast is applied to an object pointer which does not have run time type information
a compile time error will result. In the case of templates implemented with the T1 C preprocessor,
the syntax for declaring an object with run time type information is as follows:

    template<class t> class X<t> {  
        virtual void dummy();  
        t *variable;  
    }

    template<class t> class Y<t> : public A {t *variable;};

    RTTI_DECLARE_ONCE X<int>;  
    DECLARE_ONCE Y<int>;

In this case, (assuming that the A which Y<t> is inherited from is the A in the above examples) both
X<int> and Y<int> would both have run time type information. X<int> would have it because it is
specified in the RTTI_DECLARE_ONCE statement, and Y<int> would have it because this information
is inherited from A. This syntax will change when templates become available as language features,
i.e. the keywords DECLARE, DECLARE_ONCE, RTTI_DECLARE_ONCE, etc. will no longer exist.

In addition to a checked cast, Stroustrup's proposal provides for a method of comparing two
types to see if one type matches another either directly or through possible casts. The format for
this type of comparison is as follows (were A, B, and C are from the above examples):

    A *b = new B;  
    A a;  
    A *c = new C;  

    typeid(b) == typeid(A)  
    typeid(a) == typeid(A)  
    typeid(c) == typeid(A)
All of the comparisons would be true except for the last comparison. The AIPS RTTI system supports this syntax of `typeid()` applied to both variables and types which have run time type information.
3 Un-sanctioned Interface

The use of the interface described here is discouraged, because it may not be available if an RTTI should become available as a language feature in C++. A string representing the name of a type can be obtained by using the typeid() function which is a member of the Typeid objects. A reference to a Typeid object is returned by the typeid() global function previously discussed. This type name can be obtained as follows:

```cpp
A *a = new B;
A *a2 = new A;
A *a3 = new A;
B *b = new B;

const char *a_type = typeid(a).typeidO;
const char *a2_type = typeid(a2).typeidO;
const char *a3_type = typeid(a3).typeidO;
const char *b_type = typeid(b).typeidO;
```

In this example, strcmp(a_type, b_type) == 0, and strcmp(a2_type, a3_type) == 0, but strcmp(a_type, a2_type) != 0. This is the case because the typeid(a) returns the most specific type available, i.e. the type of B.

The final useful member function of the object returned by typeid(), Typeid, is const Typeid **typesigO. This function returns an array of Typeid object pointers. The first element of the array is a pointer to the Typeid of the current object. The subsequent entries are pointers to the Typeids of each of this objects parents, if it has any. The end of the array is denoted by a NULL entry. So for example, the following code will print out the inheritance tree:

```cpp
rtti class Base {virtual void dummyO();};
class Derived1 : virtual public Base {};
class Derived2 : virtual public Base {};
class Derived3 : public Derived1, public Derived2 {};

void _printTree(const Typeid *type, int count) {
    if (type) {
        for (int j = 0; j < count; j++)
            cout << "\t";
        cout << (*type).typeidO << endl;
        for (int i = 1; (**type).typesigO)[i]; i++) {
            for (int j = 0; j < count; j++)
                cout << "\t";
            _printTree((**type).typesigO)[i],count+1);
    }
}
Chapter 3: Un-sanctioned Interface

```cpp
void printTree(const Typeid &type) {
    cout << type.typeid() << endl;
    for (int i = 1; (type.typesig())[i]; i++)
        printTree((type.typesig())[i],1);
}

void Base::dummy() {};
main() {
    Derived3 d;
    printTree(typeid(d));
}
```

This piece of code will result in output resembling the following:

```
Derived3
    Derived1 Base
    Derived2 Base
```

```
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.
- 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.