Run Time Type Identification in AIPS++

DRAFT Version

Darrell Schiebel

Last modified: $Date: 2003/07/16 03:48:57 $ By: $Author: aips2adm $

1 Introduction

This document discusses the run time type identification system (RTTI). This system was implemented using the TI C preprocessor. It implements the standard proposed by B. Stroustrup and D. Lenkov[1][2]. 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.

[1] Stroustrup, Lenkov, "Runtime Type Identification for C++", The C++ Report, March-April, 1992, pp.32-42

[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++**:

```
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:

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

A *a = new B;
B *b = (checked (B *) a);
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 `#define`d 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 TI 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.

---

[1] Usenix C++ Proceedings, Aug. 1992

[2] In the general case, this is nontrivial to determine (expound latter)

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 TI 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)
typeid(b) == typeid(a)
typeid(c) == typeid(a)
typeid(c) == typeid(B)
```

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:

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

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

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 **typesig()`. 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:

```
rtti class Base {virtual void dummy();};
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).typeid() << endl;
    for (int i = 1; ((*type).typesig())[i]; i++) {
      for (int j = 0; j < count; j++)
        cout << "\t";
      _printTree(((*type).typesig())[i],count+1);
    }
  }
}

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
```