X3J16/91-0063
91-05-10

# Type Identification in C++

*Dmitry Lenkov*

California Language Laboratory
Hewlett-Packard Company
19447 Pruneridge Avenue, MS: 47LE
Cupertino, CA 95014

E-Mail: dmitry%hpda@hplabs.hp.com

*ABSTRACT*

Many applications and class libraries require a mechanism for run-time type identification and access to type information. This proposal describes a general type identification mechanism consisting of language extensions and library support. It is based in the paper [9] presented at the USENIX C++'91 conference, feedback provided by many C++ experts, and several ideas suggested by Bjarne Stroustrup.

## 1. Introduction

There have been various attempts made in C++ to implement a method of type identification for objects and a mechanism to access additional type information [3],[4],[5],[8],[9]. There are several reasons why such identification is needed.

- Support for accessing derived class functionality

  Many of the commonly available C++ class libraries (such as NIH[4], InterViews[6], ET++[5], COOL[8]) consist of an inheritance hierarchy with a root class (such as the Object class in NIH). When dealing with pointers to this root class, a common operation in these toolkits is to determine if a pointer points to an object of a derived class. If so, the pointer is cast down to the derived class so that a derived class member function may be invoked. Since C++ performs its type checking at compile time, type information is not available at runtime, and each toolkit uses different mechanisms for determining the actual type of the object being dereferenced. When the root class is a virtual base class (as in NIH), since the cast down to a derived class is not permitted by C++, the library must invent mechanisms to circumvent this restriction.

- Support for Exception Handling.

  The exception handling mechanism[1][2] requires type identification at run time, in order to match the thrown object with the correct catch clause. Since a catch clause can catch a type which is a base class of the thrown object, it is necessary for the compiler to generate information about inheritance hierarchies that is used to do correct matching. The exception handling mechanism is an example of an implicit use of the type identification mechanism.

- Support for Accessing Type Information.

  There are various class-specific actions that are difficult to achieve using the normal virtual function mechanism. For example, consider the following task: count (or do some similar task) for all nodes of a particular type in a tree of polymorphic objects.

- Support for Libraries and Toolkits.
  Once the type of an object has been determined at runtime, it may often be necessary to get further information about the type. For example, as described in [3], applications may need to know the names of classes and their inheritance hierarchy, if a customization mechanism uses class and instance names. Our proposal describes library support for getting additional information about a type.

This proposal presents language and library extensions that will support type identification and access to type information at run-time. The goal of this proposal is to create a uniform mechanism for the creation of and access to type information. The implementation strategy for the type identification mechanisms descibed in this proposal is outlined in [9].

## 2. The ptr_cast Operator

Applications often require the ability to determine dynamically if a pointer points to an object which is a subtype of a given type and cast the pointer down to the given type. The ptr_cast operator [7] is a conditional cast operator that allows the programmer to perform these two operations at once. It is a predefined operator that takes a type name as the first parameter and a pointer as the second parameter. For example,

```
ptr_cast( A, p)
```

determines if the actual type of the object pointed to by "p" is a subtype of type A (note that a type is a subtype of itself). If so, it converts "p" to a pointer to class A pointing to the class A instance which is a subobject of the object originally pointed to by "p". Otherwise it returns a NULL pointer.

Consider three classes:

```
class List {...};

class SortedList: public List {
   ...
   Key least_key();
}

class LenSortedList: public SortedList {
   ...
   int length();
}
```

Here are some examples of how the ptr_cast operator can be used. It is assumed here that a declaration of a variable is allowed in the expression of the conditional statement.

Example 1:

```
List* l_p = // initialize
...
l_p = // point to some other list
...
if( SortedList* s_p = ptr_cast( SortedList, l_p)) {
   Key k = s_p -> least_key();
   ...
}
...
if( LenSortedList* ls_p = subtype( LenSortedList, l_p))
   cout << (ls_p -> length());
```

An important characteristic of two conditional statements above is that "s_p" and "ls_p" exist within conditional statement scope only. It assures that these pointers can be used safely.

Another example is calling a function that requires an actual parameter which is a derived class.

Example 2:

```
void func( LenSortedList*);
...
if( LenSortedList* ls_p = ptr_cast( LenSortedList, l_p))
   func(ls_p);
```

In this example "l_p" is a pointer. ptr_cast can also take a reference as a parameter:

Example 3:

```
void func( LenSortedList*);
...
void foo( LenSortedList& l_r) {
...
   if( LenSortedList* ls_p = ptr_cast( LenSortedList, l_r))
      func(ls_p);
...
}
```

However ptr_cast cannot be used to assign value to a reference. If "func" takes a reference as a parameter instead of a pointer, the above two examples cannot be restructured so that "func" could be invoked with a guaranteed correct parameter. This is because ptr_cast is a conditional operator and may return NULL pointer which cannot be converted to a legal reference.

## 3. The ref_cast Operator

The ref_cast operator [7] is proposed to deal with the cases when a reference supposed to be returned. The ref_cast operator is a predefined conditional cast operator that takes a type name as the first parameter and a reference as the second parameter. For example,

```
ref_cast( A, r)
```

determines if the actual type of the object referenced by "r" is a subtype of type A. If so, it converts "r" to a reference to class A referencing the class A instance which is a subobject of the object originally referenced by "r". Otherwise it raises an exception. Here is an example how ref_cast can be used:

Example 4:

```
void func( LenSortedList&);
...
void foo( LenSortedList& l_r) {
...
   try {
      ...
      func( ref_cast( LenSortedList, l_r));
      ...
   }
   catch (bad_ref_cast) {
      ...
   }
...
}
```

## 4. The subclass operator

In the previous examples two new cast operations were used to allow functionality defined in subclasses to be used. However, there are applications that do not require cast operations to be performed while the subtype relationship between two classes still needs to be established. The subclass operator [9] is proposed to handle these cases. It is a predefined operator that, like ptr_cast, takes a type name as the first parameter and a pointer, or a reference, as the second parameter. It returns a result of type int. The result is 1 if the dynamic type of the object being pointed to, or referenced, is a subtype of the type provided as the first parameter. Otherwise 0 is returned. Consider:

Example 5:

```
void sort( List*);
...
List *l_p = // initialize
...
if( !subclass( SortedList, l_p))
   sort( l_p);
```

Consider another example:

Example 6:

```
void other_func( OtherType *);
...
OtherType p = // initialize
...
if( subclass( SortedList, l_p))
   other_func( p);
```

In this example, the functionality associated with the SortedList subclass is invoked as in example 2. However actual actions take parameters of types other than SortedList. Thus a cast operation is not needed.

It is clear that in the above examples subclass can be replaced with ptr_cast. Thus it does not add any new functionality. However it is a more efficient operator since it does not require to compute the pointer conversion.

## 5. Polymorphic and non-polymorphic types

All examples above assume that participating classes are polymorphic, i.e. contain at least one virtual function. The use of the operators introduced above for non-polymorphic classes is limited because only statically defined types can participate in the operation. If the three classes defined above are non-polymorphic (no virtual functions are declared) then in all examples above results of the introduced operators are useless. It may be desirable to produce a warning if these operators are used with non-polymorphic classes, since the programmer may not be aware that the classes are non-polymorphic.

## 6. The typeid Type

Some of the applications described in the introduction would require a unique identifier to be associated with a type. The predefined type called typeid [9] is proposed to be used. Each unique type in an application has a unique value of the typeid type associated with it. Below two operators which return values of type typeid are defined.

The typeid type can be defined in two different ways. With the first approach [9] the typeid type is a simple predefined type, similar to int or void*, with a few operations defined on it. Expressions evaluating to the typeid type can be compared for equality and inequality. Variables of the typeid type can be assigned or initialized with an expression of the typeid type. No other operations are allowed. Initialization of variables of this type can only be done through the use of the operators stype and dtype defined below.

With the second approach [7] the typeid type is a predefined class. It can be defined basically as the TypeInfo class (see below). However its instances can still be initialized be the operators stype and dtype only.

## 7. The stype and dtype operators

stype returns the type identifier (typeid value) for the static type of an expression. It can also be applied to a type name and returns the type's typeid value. The dtype operator can be applied to any expression that evaluates to a pointer, or reference, to a type. If the pointer points to a polymorphic class, dtype returns the type identifier (typeid value) of the actual type of an object pointed to by this pointer. Note that this type must be determined dynamically. If the pointer does not point to a polymorphic class, dtype returns the typeid value of the static type pointed to by the pointer definition.

Example 7:
```
List* l_p = new SortedList;
int num_Sorted_Lists = 0;
...
typeid t  = dtype(l_p);
if (t == stype(SortedList)) num_Sorted_Lists++;
```

The reason that stype and dtype are not predefined member functions is the same reason that sizeof is not a member function: both identify a fundamental property of types, as opposed to an operation on objects of those types. On the other hand, both can be applied to any types including types such as (int* (*) ()).

## 8. The TypeInfo class

Given a typeid, programmers may wish to get information about the underlying type; programmers may also wish to extend the type information automatically generated by the compiler (for example, they may wish to store the name of type). The implementation of the language features described in the previous sections will require an implementation to store some information about each class.

This information can be accessed using the TypeInfo class interface described below. The C++ library standardization effort will determine the minimum functionality to be provided by all implementations.

```
class TypeInfo {
   void* impl_p;                        // points to the implementation
                                        // dependent data structures
                                        // generated by the compiler and
                                        // link time tools

public:
   int sizeof();                           //size of type
   const char* name() const;          //class name
   int get_num_base_classes();             //number of base classes
   typeid get_base_class( int pos);        //typeid of specified base class
   int is_virtual_base_class( int pos);    //is specified base virtual?
   int visibility_of_base_class( int pos); //public(==2), protected(==1)
                                           //or private(==0) base class?

   //The routines are used to extend the compiler generated type information
   AuxTypeInfo* get_aux_typeinfo( typeid key);
   int add_aux_typeinfo( AuxTypeInfo *info, typeid key);
};
```

If the typeid type is defined as a class it can replace TypeInfo completely. All above functionality will be defined in the typeid class in this case.

## 9. typeid as a Simple Type

A standard library function called get_type_info is proposed to convert a typeid into a pointer to the TypeInfo object.

The type inquiry function is specified as follows:

```
TypeInfo* get_type_info( typeid)
```

The advantage we gain from separating the typeid type form the TypeInfo class is that we make a clear distinction between the types recognized by the language and the classes recognized by the standard class library. In addition, initialization through predefined operators is more natural for simple types than for class instances.

## 10. Extensibility

Clearly, there needs to be a way of to define and access more than just the minimal type information provided by TypeInfo. The information stored and the association of that information with the class TypeInfo object needs to be examined in detail. This section describes mechanisms [9] whereby a user may extend the type information associated with a class. It is best to allow the class library creators and users to specify what information needs to be associated with each type.

The following mechanism is used to extend the type information associated with a type.

- A member function called "add_aux_typeinfo" is provided in the TypeInfo class. This member function is used to attach additional type information to the minimal type information

```
#define ADD_TYPE_INFO( TYPENAME, INFONAME, INFO_PTR) \
 get_typeinfo(stype(TYPENAME))->add_aux_typeinfo(INFOPTR,stype(INFONAME));

#define GET_TYPE_INFO( INFONAME, OBJECT_PTR) \
 (INFONAME*) (get_typeinfo(dtype(OBJECT_PTR))->get_aux_typeinfo(stype(INFONAME)))
```

These macros can be used to rewrite the example described in the previous section.

```
class NameInfo : AuxTypeInfo {
   char *widget_name;
public:
   NameInfo(char* n): widget_name(n){};
};

NameInfo NameInfoObject = "ClassWidget";
// Attach additional type information for "Widget"
ADD_TYPE_INFO( Widget, NameInfo, &NameInfoObject)

// find out the name of a class.
Widget* w = // initialized to something;
char* widget_name = GET_TYPE_INFO( NameInfo, w) -> widget_name;
```

## 12. References

[1] Margaret A. Ellis, Bjarne Stroustrup, The Annotated C++ Reference Manual, Addison-Wesley, 1990

[2] Andrew Koenig and Bjarne Stroustrup, Exception Handling for C++, USENIX C++ Conference Proceedings, 1990

[3] John A. Interrante, Mark A. Linton, Runtime Access to Type Information in C++, USENIX C++ Conference Proceedings, 1990

[4] Keith E. Gorlen, An Object-Oriented Class Library for C++ Programs, Proceedings of the USENIX C++ Workshop, 1987

[5] Andre Weinand, Erich Gamma, and Rudolf Marty, ET++ - An Object-Oriented Application Framework in C++, ACM OOPSLA'88 Conference Proceedings, 1988

[6] Mark A. Linton, John M. Vlissides, and Paul R. Calder, Composing user interfaces with InterViews, Computer, 22(2):8-22, February 1989

[7] Bjarne Stroustrup, Personal communication

[8] Mary Fontana, Martin Neath, Checked Out And Long Overdue: Experience in the Design of a C++ Class Library, USENIX C++ Conference Proceedings, April, 1991

[9] Dmitry Lenkov, Michey Mehta, Shankar Unni, Type Identification in C++, USENIX C++ Conference Proceedings, April, 1991

generated for a type.

- A member function called "get_aux_typeinfo" is provided in the TypeInfo class. This member function is used to retrieve any additional type information that a user may have attached to a type.
- It is reasonable to expect that multiple users may wish to attach auxiliary type information to the same type. Therefore, the notion of a "key" is required. A "key" is used to distinguish between multiple auxiliary type information objects attached to the same type.

Consider an example:

```
// User wants to add a "name" field to the TypeInfo for class Widget

//See below for an explanation of the AuxTypeInfo class
class NameInfo : AuxTypeInfo {
   char *widget_name;
public:
   NameInfo( char* n): widget_name(n){};
};

NameInfo NameInfoObject = "ClassWidget";
// Attach additional type information for "Widget"
get_typeinfo( stype(Widget)) -> add_aux_typeinfo( &NameInfoObject, stype(NameInfo));

// Assuming the user has installed name information in Widget, and
// all classes derived from it, here is how a user could dynamically
// find out the name of a class.
Widget* w = // initialized to something;
char* widget_name = (NameInfo*) (get_typeinfo( dtype(w)) ->
                  get_aux_typeinfo( stype(NameInfo))) -> widget_name;
```

The extensibility scheme proposed is essentially a convenient method of adding a static member (in fact, a virtual static member) to an existing type, without having to modify the type in any way. Individual users can certainly come up with various methods of accomplishing the same result, but the goal here is to propose a *uniform* method for extending type information.

## 11. The AuxTypeInfo Class

Any additional type information should be defined as a class derived from AuxTypeInfo. Instances of this are used to link the auxiliary type information objects. See section 4.5 on additional information about the implementation of extensibility. The AuxTypeInfo class is defined as follows:

```
class AuxTypeInfo {
   // next auxiliary type info
   AuxTypeInfo* next;
   // The type of the class derived from this class
   typeid key;
};
```

The library header file can define macros so that the additional "key" parameter can be automatically generated. For example: