A General-Purpose Run-Time Type Information System for C++

Alexandru Telea

7th October 1997

Abstract

Since C++ is a statically typed language, operations concerning types have to explicitly precise types at compile-time unless the programmer supplies some 'system' that simulates type variables at run-time. This paper presents such a system which implements type variables (typeids), generalized pointer casting and object construction from run-time supplied typeids. A simple method to add these facilities to any C++ class hierarchy is presented.

1 Introduction

C++ is a widely spread, general-purpose programming language. One of its main advantages is its *strong typing*: any C++ object or variable has a well-defined type, which is declared at compile-time. This offers the great advantage of compile-time type-checking, which can trap many programming errors or inconsistencies and leads to a well-structured program design.

In contrast with this approach, interpreted languages offer the facility to declare new types at run-time and, consequently, run-time objects of these new types. This facility can be useful when an application doesn't know beforehand (i.e. at compile-time) which is the complete set of types and/or objects it will need during its execution. A typical example is an 'open' object-oriented application which consists of a pre-compiled 'kernel' and a set of user-written 'class libraries' which are loaded and used by the kernel. The idea is to leave this set *open*, i.e. to allow the user to write new classes (derived or not from the initial classes) and make them available to the kernel without having to recompile the latter, or to create instances of these classes at run-time. The kernel should be able to become aware of these new types and instantiate them at run-time.

Statically typed languages like C++ must resolve *all* operations involving types (e.g. casting and creation of objects of given types) at compile time. It is however possible to partially simulate the freedom offered by dynamically-typed languages like Objective-C or Java by introducing the concept of type variables or *typeids* (see [**?**] for an introduction to typeids for C++). The first step in this respect is done by some C++ compilers which support some run-time type operations (e.g. pointer casting) However, there are still many compilers which lack such support or provide it only partially. Another approach is taken by application libraries which provide their own run-time type information system for their classes. An example is the Open Inventor C++ library [**?**]. Yet another example is the ROOT data analysis system [**?**]. The main drawback of many

of these software solutions to run-time type information is that they impose strong constraints on the application classes which desire to benefit from it (e.g. inherit from a specific base class).

This paper presents a run-time type information (RTTI) system which offers most of the features already presented: type variables (typeids), run-time type information about any program object, run-time pointer casting in its most general form and run-time object creation from typeids. The presented system comes as a C++ header and a C++ source file which implement a set of simple tools that can be used to transparently add RTTI to any C++ class in an application. The system is written in standard C++ as described by [**?**] and should compile with any C++ compiler which complies to that standard.

2 The Concept of Run-Time Type Information

Run-time type information (RTTI) can be seen as an extension to static (compile-time) type information. It offers to the user information about pointers and references which is similar to the type information a compiler maintains while compiling a C++ program. Basically, RTTI comes in two flavours: getting type information from run-time elements like pointers and references and getting type information from 'static' elements like classes.

Since this information is to be passed to the user (who will manipulate it at run-time), we need a way to encode it in some run-time object. Such an object is called a **typeid** and keeps all information which characterizes a C++ type. In other words, for any C++ type there is a typeid object which encodes its type. In the rest of this paper we shall identify the C++ type concept with the C++ class concept, thus ignoring basic types like int, float, etc.

To precise notions, a typeid for a pointer or reference will represent the type of the actual object pointed by that pointer or referred to by that reference. The typeid for a class will obviously represent the type of that class.

3 Uses of Run-Time Type Information

If we have a means to obtain a typeid from a pointer, reference or class, we can provide a couple of useful operations on typeids:

- **1. Type names:** Given a typeid, which is an abstract encoding of a type, we would like to obtain a textual representation of it. For example, for a C++ class *A*, we would like to obtain a character string *"A"* from its typeid.
- **2. Typeid comparison:** Given two typeids, we should be able to determine if they represent the same type or if they represent types related by inheritance or unrelated types.
- **3. Pointer and reference casting:** Given a pointer and a typeid, we can determine if the actual pointed object is of the type represented by that typeid. Furthermore, we could cast a given pointer *p* to a C++ type encoded by a given typeid *t* at runtime and, if the cast succeeds, return a pointer *p'* of type *t*.
- **4. Run-time object creation:** Given a typeid *t*, we would like to create an object of type *t* and return it as a a pointer of type *t*.

Operation 1 allows us to compare types at run-time and determine their relationship (e.g. subclass to superclass). Operation 2 maps types to a textual representation which can be necessary if the user desires to be given type information in a readable way. Operation 3 gives the possibility to cast pointers and references at run-time, which is mostly used in the form of *downcasts*, i.e. casts from a base type to a derived type and gives the possibility to interpret a pointer in a different way. Operation 4 allows a program to postpone the decision of creating new objects until run-time and then create objects from run-time selected types.

4 RTTI Support for the Application Programmer

In this section we describe the effective tools we offer to the application programmer to implement and use RTTI.

A **typeid** class is introduced to represent the type of a C++ class. Some of the previously presented operations are supported directly by methods of the class typeid (see Section 5.1 for a full description of the **typeid** class). Some other operations, however, had to be implemented as C++ macros, since they return *typed* pointers or have *type* arguments (by type, we mean a C++ class type). As outlined before, C++ does not have 'type variables' which could be passed to or returned from methods, hence the only way to implement such operations was to use macros.

To summarize, the programmer can manipulate type information at run-time by using typeid instances. Such instances can be freely copied, deleted, assigned to and copy constructed, mostly like any 'well-behaving' C++ class instances.

We shall present firstly these methods and macros which support the operations described in Section 3. A simple method to add RTTI capabilities to a C++ class will be then introduced. Finally, we give some examples of how the RTTI system can be used.

4.1 RTTI Basic Tools

In the following, **RTTI-class** stands for a C++ class which has RTTI capabilities, **p** for a pointer to a RTTI-class and **T** for a C++ RTTI-class. The 'return type' of a macro stands for the type of object that macro expands to.

- **typeid STATIC_TYPE_INFO(T):** Returns a typeid encoding the type of the C++ RTTI-class **T**. If **T** is not a RTTI-class, a compile-time error occurs.
- **typeid TYPE_INFO(p):** Returns the typeid for the object **p** is pointing at (regardless of **p**'s own type). If **p** is not pointing at an object of a RTTI-class, a compile-time error occurs. If **p** is NULL, a special typeid for the NULL pointer is returned.
- **T* PTR_CAST(T,p):** Returns the value of pointer **p** cast to the type **T**, i.e. a **T*** or NULL if cast fails. If either ***p** or **T** are not RTTI-classes, a compile-time error occurs.
- **int typeid::operator==(typeid):** Compares two typeids and returns 1 if they represent the same type, else 0. An operator!= with the natural semantics is also provided for typeids.
- **const char* typeid::getname():** Returns the textual name of a typeid.

- **T* TYPE_NEW(T,typeid t):** Given a C++ base type **T**, an object of type **t**, where **t** represents a type identical to or derived from **T**, is created and returned as a **T***. If **T** is not a RTTI-class a compile-time error occurs. If **t** is not a type derived from **T** or not an instantiable type having a default constructor, NULL is returned. The implicit reasons for needing to supply the type **T** of a baseclass of *t* are described in detail in Section 5.2.6.

4.2 Adding RTTI to a C++ Class

This section describes the steps to be taken to add RTTI capability to a C++ application class. These consist of two main operations. The first one amounts to adding one line to the class declaration. For the following example of a declaration for a C++ class called **C**:

```
class C {
            .....

        };
```

We only need to add the text **TYPE_DATA** to the declaration. This is best done by adding it right before the closing brace (although it is legal to add it anywhere in the class declaration):

```
class C {
            .....

            TYPE_DATA
        };
```

The second operation consists of adding a macro to the file containing the class definition (i.e. the **.cc** file). The macro has the general form:

RTTI_DEFn(*classname*,*"classname"*,b_1,..., b_n)

Here *classname* is the C++ class we add RTTI to, *n* is the number of direct base classes of *classname* which have RTTI (omit *n* if *classname* has no bases with RTTI), *"classname"* is the textual name we want to have in the RTTI system for the C++ class *classname* and b_1,...,b_n are the C++ direct bases of *classname* which have RTTI.

For example: for a class declared as follows:

```
class C : public A , public B
```

We should add the following line to its definition file:

```
RTTI_DEF2(C,"C",A,B)
```

A last remark: if we plan to use the run-time object creation feature of the system, we have to declare which RTTI-classes are instantiable by a default constructor and which not. In order to make a C++ class run-time instantiable, one should use exactly the same procedure as above but replace the macro name RTTI_DEF byRTTI_DEF_INST. Of course, that class should have a public default constructor, otherwise it is impossible firstly to create an instance of it and secondly not to supply any constructor parameters to the TYPE_NEW macro previously described.

4.3 Additional RTTI Tools

In Section 4.1 we have introduced the most important tools that the application programmer can use to exploit RTTI. This section introduces some additional, less important tools which refine the features made available by the first set of tools or are more convenient to use (e.g. less verbose). Assume the same notations as in Section 4.1.

- **const char* STATIC_TYPE_NAME(T):** Returns the textual name of class T (shorthand for **STATIC_TYPE_INFO(T).getname()**).
- **const char* TYPE_NAME(p):** Returns the textual class name of the class of ***p** (shorthand for **TYPE_INFO(p).getname()**).
- **int DYN_CAST(typeid t,p):** Given a pointer *p*, returns 1 if it can be cast to the type described by *t*, else 0. This macro is similar to **PTR_CAST** but offers more freedom. While **PTR_CAST** lets you vary the pointer you cast (but still keeps the type you cast to fixed, since given at compile time), **DYN_CAST** lets you vary both the pointer and the type (represented now by a typeid) at run-time. **DYN_CAST** is therefore useful when you want to check that a pointer selected at run-time casts to a type selected at run-time. Of course, it has a drawback: while **PTR_CAST** returns a typed C++ pointer, **DYN_CAST** can only return a flag (1/0) since one can not obtain *typed* pointers unless the type is precised at compile-time (see also Section 4).
- **int typeid::can_cast(typeid):** Similar to **DYN_CAST**, this method returns 1 if the typeid argument can be cast to 'this', else 0. This method can be useful to check casting directly on typeids rather than on pointers and C++ types.
- **typeid::typeid(const typeid&),... :** The typeid class provides all the usual copy constructor and assignment operator semantics. In other words, typeids can be copy constructed, passed and returned by value or by reference, assigned to and so on. This makes the typeid class as simple to use as a basic C++ data type.

4.4 Creating typeids

So far, the programmer can retrieve typeids from existing C++ classes or pointers to objects of these classes. There are however situations when the user would like to 'create' a typeid at run-time and treat it as a typeid obtained from a compile-time type. Such a situation can occur if the user of an application wants to supply a 'type name' at

run-time and require the creation of an object of that type. We already have presented the **TYPE_NEW** macro which can create an object given a typeid for the desired type. However, the user can not give such a typeid at run-time since he has no way to access a C++ class to retrieve its typeid.

One solution is to create a new kind of typeid which will be called a **dyn_typeid** (**dyn** stands for dynamic). Such a typeid can be created from a textual description of a type name (e.g. a char*) and should be used *only* for the **TYPE_NEW** macro, since it doesn't contain actually any information but a textual type name. See Section 4.5 for an example on how to use a dyn_typeid.

4.5 Examples:

Following are some examples of how the previously presented tools can be used to perform several run-time operations:

- **Example 1: Pointer casting** Assume we have a class **C** derived form a class **B** which is further derived from **A**. We shall create a B, reference it via an A* and down-cast it to B (which should work) and then to C (which should fail).

```
A* pa = new B;          //construct a B and reference it
                         //via a base-class pointer

B* pb = PTR_CAST(B,pa); //run-time cast the A* to a B*.
                         //Will succeed since
                         //pc is in fact pointing to a B

if (pb) pb->b_method(); //check if cast ok, then call a method of B

C* pc = PTR_CAST(C,pa); //cast the A* to a C*. Will fail since
                         //pa is pointing to a B but not a C

if (pc) pc->c_method(); //pc will be NULL, so nothing will be done
```

- **Example 2: Using textual information** Assume the same class hierarchy as in Example 1. This example shows how the user can interrogate types and pointer and retrieve their textual names.

```
f(A* ptr)
{
    cout<<"This pointer is of type<<STATIC_TYPE_NAME(A)
    <<"but actually points to a "<<TYPE_NAME(ptr)<<endl;
}
```

Assuming ptr points to a C object and the RTTI information has been defined as in Section 4.2, the above code will print:

```
This pointer points is of type A but actually points to a C
```

- **Example 3: Using typeids** Assume the same classes as in in Example 1. We shall check casting with typeids now instead of C++ types.

```
typeid At = STATIC_TYPE_INFO(A);
typeid Bt = STATIC_TYPE_INFO(B);
typeid Ct = STATIC_TYPE_INFO(C);

A* pa = new B;          //construct a B and reference
                         //it via a base-class pointer

int cast_to_B = DYN_CAST(Bt,pa); //cast_to_B will be 1
                                 //since pa is pointing to a
int cast_to_C = DYN_CAST(Ct,pa); //cast_to_C will be 0
                                 //since a B is not a C
```

- **Example 4: More typeids** Assume the same code as in in Example 3. We shall check casting without macros now.

```
typeid t = TYPE_INFO(pa);  //get typeid of the pa pointer

int cast_to_B  = Bt.can_cast(t); //trying to cast t to
                                 //a Bt will return
int cast_to_C  = Ct.can_cast(t); //trying to cast t to
                                 //a Ct will return 0
```

- **Example 5: Run-time object creation** Given a textual type name at run-time, we shall create an object of that type. This example uses dyn_typeids which have been introduced in Section 4.4.

```
char name[10];

cout<<"Give name of type to instantiate (subclass of BASE)";
cin>>name;                    //get a textual type name
                              //from the user

dyn_typeid t(name);           //create a typeid for
                              //the given type name

BASE* obj = TYPE_NEW(BASE,t); //try to create an object
                              //of type t
```

In the above example, we assume that the types the user wishes to instantiate are all derived from some base class BASE. The reasons the types supplied by the user have to be derived from a known type are described in Section 5.2.6.

5 Implementation of the RTTI Mechanism

The previous sections have presented the facilities offered by the RTTI mechanism to the application programmer but haven't described the implementation of this mechanism. In fact, the user of the RTTI system should not be concerned about the way the features he uses are implemented.

The design of the RTTI system tried to face a set of requirements including the palette of operations offered to its user (described in Section 4) and shielding the user from the RTTI implementation. This means that there could be several possible implementations of the RTTI mechanism which should ultimately offer the same API, basically the one introduced in Section 4.

However, some RTTI implementations would not have been able to support *all* the operations presented in the RTTI system API. Moreover, there exist several similar RTTI systems which provide a very similar API but with different semantics. Although not obvious, such differences can be essential once RTTI is used for larger, more complex C++ class hierarchies.

The intent of this section is to present our implementation for the RTTI mechanism. The important constraints which arose during the systems' design will be outlined and the advantages and drawbacks of the chosen solutions will be commented, often by comparison to other possible solutions.

5.1 Interface and Implementations of Typeids

The user's perspective on a typeid is of a class which encodes 'all type information' about an application C++ class. The definition of a C++ class is unique in a program. However, we can have more typeid objects representing the type of the same class. This fact, added to the fact that the interface of the typeid and its implementation should be separated, leads to the idea that a typeid should only be a 'handle' to the real type information representation. In this way, we a) insulate the typeid interface from its implementation and b) can easily create and destroy typeids but keep only one copy of the type information per C++ class (the same idea is used by [**?**]).

We shall introduce here a new class **Type_info** which represents the real implementation of the type information for a C++ class. In other words, each C++ class will have a unique Type_info object which will encode its type information, but there can be any number of typeids which refer to this Type_info. In terms of object-oriented design patterns, the typeid is a Handle and the Type_info is a Body. typeid delegates all its requests to its Type_info object. typeid can be also seen as a Proxy for a Type_info (it acts as a Type_info, but is lighter and simpler). Type_info can be seen as a (sort of) Singleton, since there is only one instance per C++ class. See Section 5.2 for a complete description of the Type_info implementation.

One design problem is related to the fact that for each C++ class which desires to use RTTI there has to be exactly one Type_info object. Since this object is however related to the C++ class rather than to its instances, the best way is to implement it as a static class member. Besides unicity, this will also ensure that any C++ class will automatically

create its (static) Type_info instance. The relationship between typeids, C++ classes and their Type_infos is as depicted in Figure 1:

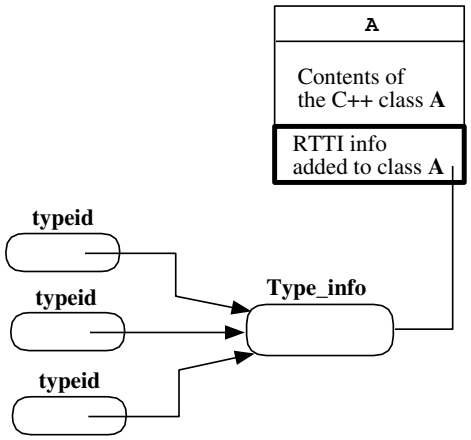

Figure 1: Relationship between typeids, C++ classes and their Type_infos. A C++ class **A**, its Type_info static member and three typeid objects which refer to it are shown

The second main design point relates to type information retrieval: how do we retrieve a typeid from a C++ class or pointer ? The first question is trivial, since we can retrieve the static Type_info C++ class member having a C++ class (this is exactly what the macro STATIC_TYPE_INFO does). The second question however needs some additional work, since we do not want to relate the information with the pointer type but to the pointed object actual type. Basically, we can do this in C++ by inserting the information in the class object and adding a virtual method to this object.

This is exactly the way we implemented the PTR_CAST, TYPE_NAME and TYPE_NEW macros: they use some virtual functions of the pointed object (one of them, for example, returns the static Type_info of its class).

To summarize, we implemented the RTTI mechanism by two basic constructions: virtual functions and a static Type_info object added to each C++ class. This leads us to the important observation that we need to be able to add virtuals to a C++ type which wishes to support RTTI. This explains why this approach can work only for classes but not for basic C++ types (e.g. integers). This is however not a limitation since anyway one can't derive types from, say, integers, so run-time casting and other similar operations are actually meaningless for integer pointers, for example.

5.2 The Type_info Class

In this section we shall present the structure of the Type_info class and how that information, in conjunction with the virtual functions previously mentioned, can perform the RTTI operations we need.

Section 5.1 described the typeids as being proxies for Type_infos. This implies that the four main RTTI operations we want to support (see 3) must be implemented at Type_info level. As previously said, one Type_info is automatically constructed for each C++ class using RTTI. The access to these Type_infos is done only via Type_info pointers stored in typeids, so Type_infos are never copied or passed by value.

In the following we shall take each RTTI operation and describe in which way it is implemented inside Type_info and gradually build the representation of Type_info.

5.2.1 RTTI Operation: Retrieving Type Information From Classes and Pointers

The first operation we need is to get a Type_info from a C++ class pointer or a C++ class. As already explained in the previous section, the Type_info is a static member called, for example, $RTTI\_obj$ of the C++ class. Given a C++ class T as argument, the STATIC_TYPE_INFO macro will simply retrieve the RTTI_obj member of T. Given a C++ class pointer p (which, say, points to an actual T object), the TYPE_INFO macro will call a virtual method via p which will return the same RTTI_obj object (Figure 2.

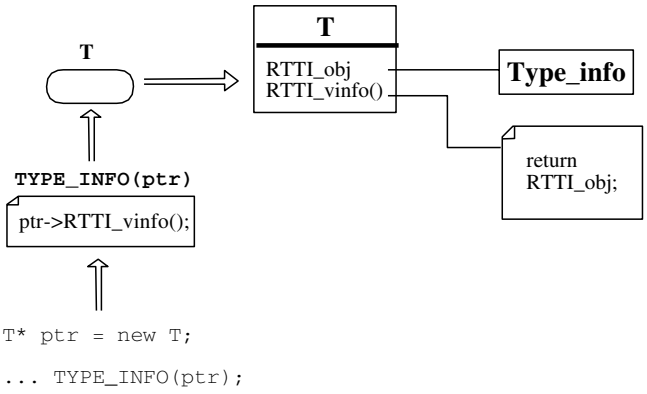

Figure 2: Retrieving Type_info from C++ pointers. The TYPE_INFO macro calls a virtual function of the pointed object which returns the static Type_info class member of its class.

In order to do this, we need to add a static Type_info member and a virtual method $RTTI_v info()$ to each C++ class T which wishes to use RTTI operations.

5.2.2 RTTI Operation: Type Names

The simplest information is a textual type name associated with each Type_info. In order to implement this, Type_info will internally store a char* which encodes the textual name of the type it represents. The name information is passed to Type_info at construction. When a typeid is asked the type name it will delegate this reauest to its Type_name 'body' (Figure 3). This is, for example, the way the TYPE_NAME macro is implemented.

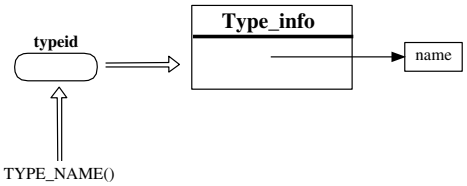

Figure 3: Implementation of type names in the Type_info class. The double arrow shows the message passing sense.

5.2.3 RTTI Operation: Typeid Comparison and Casting

As previously described, typeids can be compared for equality by their **operator==()** or for inheritance relationship by the **can_cast()** method.

The comparison operator for typeids will compare their Type_info pointers. If the comparison fails, it will compare 'deeper', i.e. the textual type names stored in the typeids. One could wonder why should a deep compare be necessary if there is exactly one Type_info instance per C++ class and the names of the C++ classes are unique in a program. The answer is that there is an additional case when Type_infos are created (see Section 4.4 and therefore there *may* be cases when there are several Type_info objects with the same name.

The **can_cast()** method basically finds out if a given Type_info $t1$ represents a base class of another given Type_info $t2$. In order to implement this, Type_infos have to store ancestor relationships. Similarly to C++ class inheritance where a class declares its direct bases, a Type_info will store pointers to the Type_infos representing the direct base types of its type. This information is passed to Type_info at construction time. The Type_info objects will therefore create a graph isomorphic to the inheritance graph created by their C++ classes (Figure 4). Having this data, **can_cast()** simply reduces to a recursive search for $t1$ in the Type_info graph rooted at $t2$.

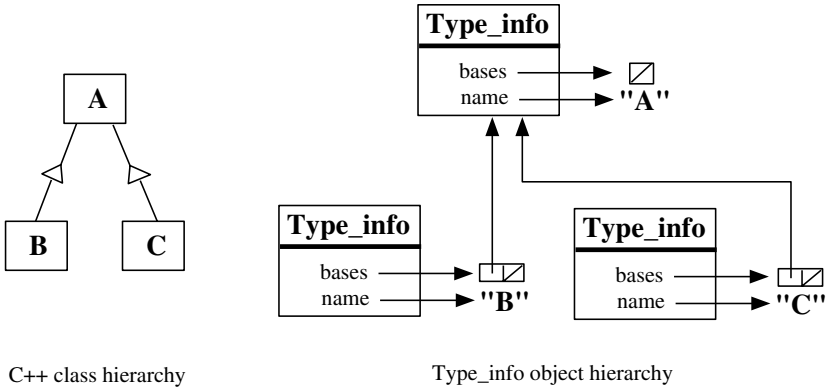

Figure 4: Implementation of type comparison and casting in the Type_info class.

5.2.4 RTTI Operation: Pointer Casting, First Attempt

At a first glance, it seems that one could use the inheritance graph of Type_infos described in the previous section to implement the PTR_CAST operation on C++ pointers: in order to cast a pointer p to a C++ type T, retrieve the Type_info $t1$ of T and $t2$ of p and check if $t2$ can be cast to $t1$ with the previous method. If so, then return the cast pointer $(T*)p$, else return NULL.

This simple implementation, described also by [**?**], fails to work in two major situations. The first situation is when the type A of p is a virtual base class of B (Figure 5 a). In this case, C++ will be unable to perform the cast $(B*)p$ even though p is in fact pointing to a B. The reason is that there is no compile-time information that the C++ compiler could use to perform a $(B*)p$ in case p points formally to a virtual base of p. Attempting to perform such a cast will hence be rejected at compile-time.

The second situation is more dangerous: suppose we have a C++ class A derived from two classes B and C and a $B*$ pointer p which points to an A (Figure 5 b). We

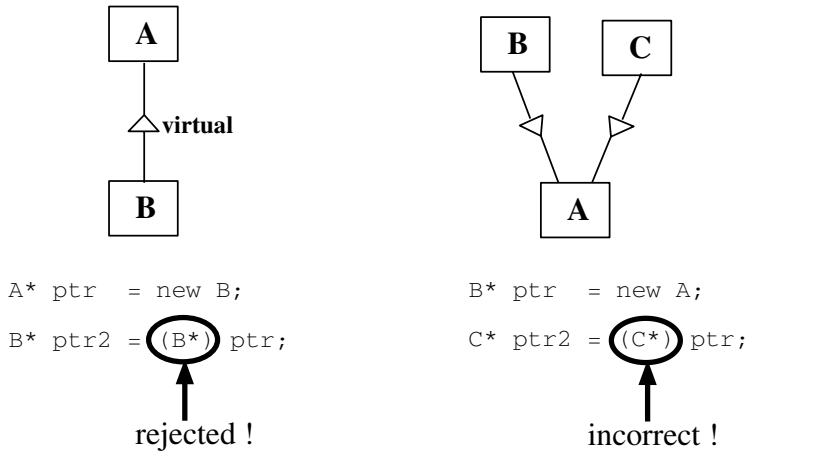

Figure 5: Cases when C++ pointer casting will fail to work correctly. **a.** Virtual base to derived class cast. **b.** Cast between types on different inheritance paths.

should be able to cast p to a C, since it actually points to an A which is a C also. If we however directly cast the $B*$ to a $C*$, the compiler will simply interpret p as pointing to a C without doing any offset calculations for finding the correct position of the C subobject in the A subobject. The reason of this is that C++ performs offset calculations when casting pointers *only* if the types of those pointers happen to be on the same path in an inheritance graph. In the previous example however, C and B are on two different inheritance paths (which both start at A). In this case, the compiler will not refuse the cast but simply reinterpret the $B*$ as pointing to a $C*$, which is obviously wrong. Worse, there is no warning issued when such a cast is performed.

To summarize, direct C++ pointer casting can not be reliably used to implement the PTR_CAST operation, since it works correctly only for types which belong to the same inheritance path and which are not related by virtual inheritance.

5.2.5 RTTI Operation: Pointer Casting, Correct Attempt

The solution to performing a correct and general C++ pointer cast is to introduce a virtual method in the C++ application class whose instance pointers are cast. This method (called *RTTI_cast*) receives a typeid argument t and returns the object itself (i.e. its *this* pointer) correctly cast to the C++ type T corresponding to t, if *this* can be cast to T i.e. t is a base of *this* or NULL otherwise.

The implementation of *RTTI_cast* is as follows (in this example C is a C++ class having direct bases A and B):

```
void* C::RTTI_cast(typeid t)
{
   if (t == &RTTI_obj) return this;
   void* ptr;
   if (ptr=A::RTTI_cast(t)) return ptr;
   if (ptr=B::RTTI_cast(t)) return ptr;
```

```
    return 0;
}
```

The idea behind *RTTI_cast* is to bypass C++'s pointer casting mechanism and to use a custom casting. For this, *RTTI_cast* of a class is recursively calling the *RTTI_cast* methods of its bases. Moreover, all pointers are transmitted *as void*'s and NOT as typed C++ pointers*.

Having the above, it is now very simple to implement PTR_CAST:

```
#define PTR_CAST(T,ptr) (T*)p->RTTI_cast(STATIC_TYPE_INFO(T))
```

The difference between this implementation of PTR_CAST and the one presented in the previous section is obvious: now all the 'real' casting work is done inside *RTTI_cast*. There is just one explicit C++ cast from void* to T* which is perfectly safe since we're guaranteed that *RTTI_cast* returns a pointer of type T. Moreover, this version of *RTTI_cast* can obviously solve the virtual base problem and the casting across inheritance paths problem. Finally, the run-time cost of the new *RTTI_cast* is basically identical to the old version. The only extra price to pay is the declaration and definition of a new virtual in each C++ class (fortunately, this process can be automated).

5.2.6 RTTI Operation: Run-Time Object Creation

The last operation to be implemented is run-time object creation. As described, this operation creates an object of a type specified at run-time by a typeid. In a certain sense, it can be seen as a run-time equivalent of the C++ **new** operator.

The macro TYPE_NEW which implements run-time object creation needs two parameters. One of them is a typeid t for the type T we want to instantiate. The other one is a C++ class name of a base class B of T. The second parameter is necessary since we have to return the newly created object as a typed object. In other words, TYPE_NEW(B,T) will create a new T and return it as a B*.

The implementation of the TYPE_NEW RTTI operation which performs this task has to add one extra piece of information. Namely, the RTTI system has to maintain a repository of all application classes (more exactly, of all Type_infos of all application classes). When a TYPE_NEW operation is requested, the type to be instantiated is looked up in this repository and, if found, is instantiated.

The proposed solution is to implement this repository in a 'distributed' fashion. Rather than keeping a centralized list of all types of an application, the idea is to let each Type_info know about all types directly derived from it (similarly to the way a Type_info knows about its bases)(Figure 6). This solution fits well the previous description of the TYPE_NEW operation: when TYPE_NEW(T,t) is invoked, the typeid t will be searched in the inheritance graph rooted at T downwards

Keeping information about derived types inside each Type_info object offers a faster way to locate the existence of a type to be instantiated at run-time than the centralized repository version. Indeed, in the case the information about subclasses is kept locally in each Type_info, the system must only scan the inheritance path starting at the root type specified in the TYPE_NEW operation. If the system kept a 'flat' list with all Type_infos of an application, the worst case search would mean traversing the whole list.

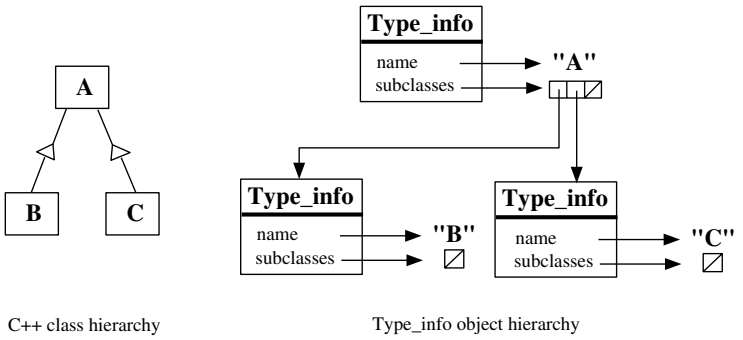

Figure 6: Type_info implementation support for run-time object creation

After finding the right Type_info, the system should create a new instance of its corresponding C++ class. In ordr to do this, it has to call the C++ **new** operator passing the C++ class as argument. This can not be done generically for the same reason which forbade us casting C++ pointers generically, namely the lack of 'real' type variables in C++. Moreover, we can't use virtual constructors since the C++ classes the RTTI system should manage do not always have a common base. The only available solution is to declare a static method in each C++ application class which returns a new object of that type. However this is only half of the solution since we should return the new object as a pointer cast to the typeid argument supplied to TYPE_NEW. The solution is to use the method *RTTI_cast* introduced in Section 5.2.5 on the newly created object, passing it the typeid to cast to.

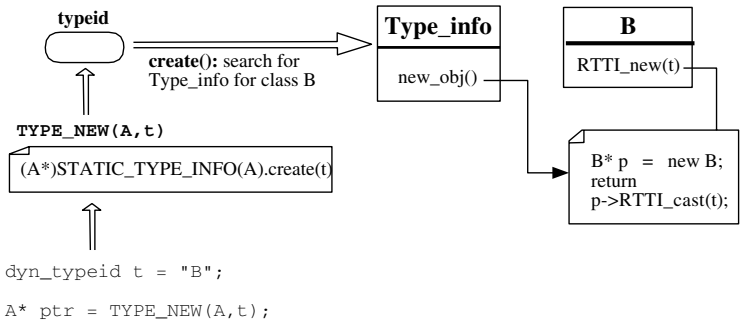

Figure 7: Example of run-time object creation

In the example presented in Figure 7, class B inherits from A. The steps of the user request to create a B and return in as an A* are outlined. First, the Type_info for B is searched in the graph rooted at A (by a method of Type_info called **create()**). When this Type_info is found, the creation static method of class B (i.e. B::RTTI_new())is retrieved from the Type_info. This implies that a Type_info for a class has to store a pointer (called here *new_obj*) to the object creation method RTTI_new of the class it represents. When this method is called, a B is allocated on the heap and it is returned as a pointer cast to the typeid of A. Finally, TYPE_NEW will properly cast the void* returned by *create*() to an A* (which is safe since RTTI_cast produced an A*).

Marginal cases include trying to instantiate non-existing classes (in which case *create*() will fail to find the respective Type_info and will return NULL) or trying to instantiate

classes which have no default constructors like abstract classes (in which case the respective Type_info will have a NULL $new\_obj()$ member so will simply return NULL when asked to instantiate its class).

5.3 Automatic Creation of Type_infos for C++ Classes

The previous sections presented the implementation of the RTTI operations in terms of the Type_info class. We assumed that for each C++ application class, there exists exactly one Type_info object which is correctly created to represent all type information for that class. Moreover, we assumed that the application C++ classes contain a number of virtual methods which are called back by the RTTI operations.

In Section 4.2 we presented the two steps the user has to take in order to add RTTI to a C++ class: the insertion of the TYPE_DATA keyword in the class declaration and of the RTTI_DEF and RTTI_DEF_INST keywords in the file containing the class definition. This section describes the actions which are automatically taken by the RTTI system when those keywords are inserted in a C++ source and describe how the Type_info objects are initialized.

As one might have already guessed, TYPE_DATA and RTTI_DEF are C++ macros which expand into code for declaration and initialization of some features of the RTTI system. They are described in detail in the following.

5.3.1 The TYPE_DATA Macro

The TYPE_DATA macro expands into a set of declarations of static and virtual methods and members which have to be added to an application C++ class that wishes to use RTTI. More precisely, the following members are added:

```
protected:
   static  const  Type_info RTTI_obj;
   static  void*  RTTI_new(const Type_info*);
public:
   static  typeid RTTI_sinfo();
   virtual void*  RTTI_cast(typeid);
   virtual typeid RTTI_vinfo() const;
```

The $RTTI\_obj$ member is the static Type_info which represents this class. $RTTI\_new()$ has been described in the previous section. It creates a new object of this type and returns it as a pointer cast to the Type_info parameter. Both $RTTI\_sinfo()$ and $RTTI\_vinfo()$ return the $RTTI\_obj$ member of this (the static method is used for the STATIC_TYPE_INFO operation while the other one by the TYPE_INFO operation). The last method, $RTTI\_cast$, returns a pointer to an object of this class cast to the typeid parameter and is used by the PTR_CAST operation.

5.3.2 The RTTI_DEF and RTTI_DEF_INST Macros

The RTTI_DEF and RTTI_DEF_INST expand into the definition of the class members declared in each application class by the TYPE_DATA macro. The sematics of the methods having been presented in the previous sections, we shall only describe the initialization of the Type_info member.

The Type_info member contains the following members to be initialized: the type's textual name, the Type_infos for the type's direct bases and for the type's direct subclasses and a pointer to this type's $RTTI\_new()$ static method. All this information can be easily obtained since the RTTI_DEF macros provide it directly or indirectly via their parameters.

One case deserves however particular care: the initialization of the list of direct subclasses of a Type_info. This list is implemented as a Type_info** and is called *subtypes*. In a project having multiple files with multiple classes, it may happen that a Type_info for a superclass tries to add itself to the *subclass* member of one of its subclass Type_infos *which has not been constructed yet*. We can not help this situation since the order of construction of static class members in a C++ program is undefined. One might think that this situation leads to unpredictable or dangerous behaviour. This is however not the case, since the compiler first allocates all static objects and initializes them to zero and only then starts calling their constructors. Since Type_info calls no virtual methods in its constructor and since we're guaranteed that all Type_info static objects are zeroized at the time the first Type_info constructor is called, we can guarantee that our initialization scheme (which relies on finding zeroized Type_infos and calls non-virtual methods on not yet constructed static Type_infos) is safe for any compilation system.

5.4 Implementation of Dynamic Typeids

Section 4.4 introduced the dynamic typeid concept, represented by the **dyn_typeid** class. As explained, this class is useful when the user wishes to create a 'dummy' typeid from a textual name and pass it, for example, to the TYPE_NEW operation.

Class **dyn_typeid** is similar to **typeid** in that it is a 'proxy' to a Type_info. However, a dyn_typeid can be constructed also from a textual type description (i.e. a char*). This will create a 'dummy' Type_info which contains only the name field.

Since dyn_typeid can be only used for passing it as argument to the **typeid::create()** method, we declare this method as accepting dyn_typeids and we derive typeid from dyn_typeid and not conversely. Thus a typeid can be used as a dyn_typeid but not conversely. Moreover, methods using dyn_typeids like **typeid::create()** will be aware that they should only access the type name.

6 Comments

This paper has presented a run-time type information (RTTI) system which lets the programmer add several features to any application class: textual type name information, generic type management via typeids, run-time pointer casting for any combination of pointer type and target type and type instantiation from run-time specified typeids.

The RTTI system adds only a minimal overhead to application classes (a few virtual and static methods and a small static object per application class). The system is guaranteed to work correctly in any C++ application compiled with a C++ compiler which follows the basic language specifications. Moreover, an application planning to use RTTI has to perform only minor modifications (add two keywords, one per class declaration and one per class definition). As any C++ library, the RTTI system introduces a few new indentifiers, in the global scope as well as in each application class which uses RTTI. We have attempted to minimize name clashes by using private declarations where applicable or prefixing the global indentifiers with $RTTI\_$.

The presented system is incremental, i.e. the features it offers can be added separately starting from the simplest to the most complex (i.e. from textual type names, continuing with typeids and typeid operations, pointer casting and ending with run-time object creation).

The features offered by the system are also incremetal, i.e. they can be used independently of each other, starting with the simplest and ending with the most involved ones. Programmers can therefore understand only those features that their particular applications need.

References

[Brun et al., 1995] Brun, R., Buncic, N., and Rademakers, F. (1995). *The ROOT System*. http://root.cern.ch.

[Stroustrup, 1991] Stroustrup, B. (1991). *The C++ Programming Language, Second Edition*. Addison-Wesley.

[Wernecke, 1994] Wernecke, J. (1994). *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor*. Addison-Wesley.