Runtime Access to Type Information in C++

John A. Interrante and Mark A. Linton
Stanford University

Abstract

The C++ language currently does not provide a mechanism for an object to determine its type at runtime. We propose the *Dossier* class as a standard interface for accessing type information from within a C++ program. We have implemented a tool called *mkdossier* that automatically generates type information in a form that can be compiled and linked with an application. In the prototype implementation, a class must have a virtual function to access an object's dossier given the object. We propose this access be provided implicitly by the language through a predefined member in all classes.

1 Introduction

Some applications need to know the names of classes, their inheritance structure, and other information at runtime. For example, the X Toolkit Intrinsics [3] define a customization mechanism based on class and instance names. With this mechanism, a user can pass a string to an application that is matched against instances of a named class. To applications, the string "*Button*font:courier14" means that the default font for all instances of Button is "courier14". InterViews [2] is a C++ toolkit that supports the X Toolkit customization mechanism. Because the C++ language currently does not support access to any type information at runtime, programmers must write code in every InterViews class that defines the class's name.

Unfortunately, every class writer who needs runtime access to type information must invent their own conventions. These conventions make the exchange of user-defined data types between programmers difficult. For instance, both OOPS[1] and ET++[5] use macros in class definitions to provide a class's name and other information about class types. Neither library can reuse a class type from the other library without modification. Even if all libraries followed a standard set of conventions, these conventions still make writing classes more tedious.

Class writers must take two steps to eliminate these problems: (1) define a standard interface for accessing type information at runtime, and (2) define a way to generate the type information automatically. In this paper, we propose the class *Dossier* as the standard interface to type information, and we describe the implementation of a tool called *mkdossier* that generates a C++ source file containing dossiers. A programmer can then compile and link this file with an application.

A type declaration might not reflect the type of an object at runtime—the object could be a subclass of the declared class. Our prototype implementation therefore requires a class to have a virtual function which simply returns the dossier for its class. Adding a virtual function presents a problem because this change requires recompilation of source code defining or using the class. We propose that the C++ language be extended to provide a predefined (not reserved) member

Research supported by a gift from Digital Equipment Corporation, by a grant from the Charles Lee Powell Foundation, and by an equipment loan from Fujitsu America, Inc.

```
class Dossier;

class DossierItr {
public:
    DossierItr();
    virtual ~DossierItr();

    boolean more();
    void next();

    // dereference through current element
    Dossier* operator ->();

    // coerce to current element
    operator Dossier*();
};

class Dossier {
public:
    Dossier(
        const char* name,
        const char* fileName,
        unsigned int lineNumber,
        Dossier** parents,
        Dossier** children
    );
    virtual ~Dossier();

    const char* name() const;
    const char* fileName() const;
    unsigned int lineNumber() const;
    DossierItr parents() const;
    DossierItr children() const;
    boolean isA(const Dossier*) const;

    // return array of dossier pointers for all classes
    static Dossier*const* classes() const;
};
```

Figure 1: Interface to dossiers

containing a pointer to a dossier. This extension would make type information available for existing classes without requiring changes to their source code.

2 Dossier interface

The name "Dossier" connotes detailed information about a subject. In this case, we want a dossier to be the repository for information about a type. Although a dossier could represent information about any type, in this paper we will only consider class types. Accessing information

```
void traverse(Dossier* d) {
    for (DossierItr i = d->parents(); i.more(); i.next()) {
        cout << "traversing " << i->name() << endl;
        traverse(i);
    }
    cout << "back to " << d->name() << endl;
}
```

Figure 2: Traversing ancestors using iterator

for non-class types requires compiler and language support, which we did not wish to undertake before defining an interface for classes.

Figure 1 shows the Dossier interface. The current interface provides limited information about a class: its name, the file and line number where the class is defined, and iterators to visit parent and children classes. The "isA" function determines whether a class type is a subclass of a given class type. The "classes" function allows the application to access all defined dossiers. Figure 2 shows a sample function that uses an iterator to print the names of all the ancestors of a class.

We anticipate extending the interface to include size information, names of members, and member functions. We concentrated on the minimal functionality so that we could investigate the mechanism without getting bogged down in too many details. When we settle on the final details, we can extend the functionality by replacing the standard "dossier.h" header file where Dossier is defined, updating the library implementation of Dossier, and providing a version of mkdossier that generates the additional information.

3 Dossier implementation

The Dossier interface describes what information is available for a class type; it does not say how the type information is generated. We expect a compiler or a special tool to automatically generate the representation of dossiers so that programmers need not manually define or update dossiers. Like the virtual function table used by most C++ compilers to implement virtual function calls, only one dossier representation should exist for each class in an application. We have implemented a tool called mkdossier that we use to generate dossiers just like we use makedepend, a tool developed at MIT, to generate Makefile dependencies.

Figure 3 shows the role of mkdossier in building an application. The build process calls mkdossier to scan all the source files and generate "_dossier.h" and "_dossier.c". Then the build process compiles the source files, including "_dossier.c", and links them into an executable image.

Figure 4 shows a sample "_dossier.h", the header file that declares the generated dossiers' names. The dossiers represent information about the sample classes "App," "ArgVec," and "CPP"; mkdossier defines each dossier's name by concatenating the prefix string "_D_" and the class type's name. The application can include this header file to import dossiers into application code by name as well as use the "Dossier::classes" function to import dossiers by address.

Figure 5 shows a sample "_dossier.c", the source file that initializes dossiers with information about class types. Programmers can avoid having to manually define dossiers by compiling and linking this file into the application's executable image. At the end of the file, mkdossier generates a static array of pointers to all the dossiers defined in that file. The "Dossier::merge" function

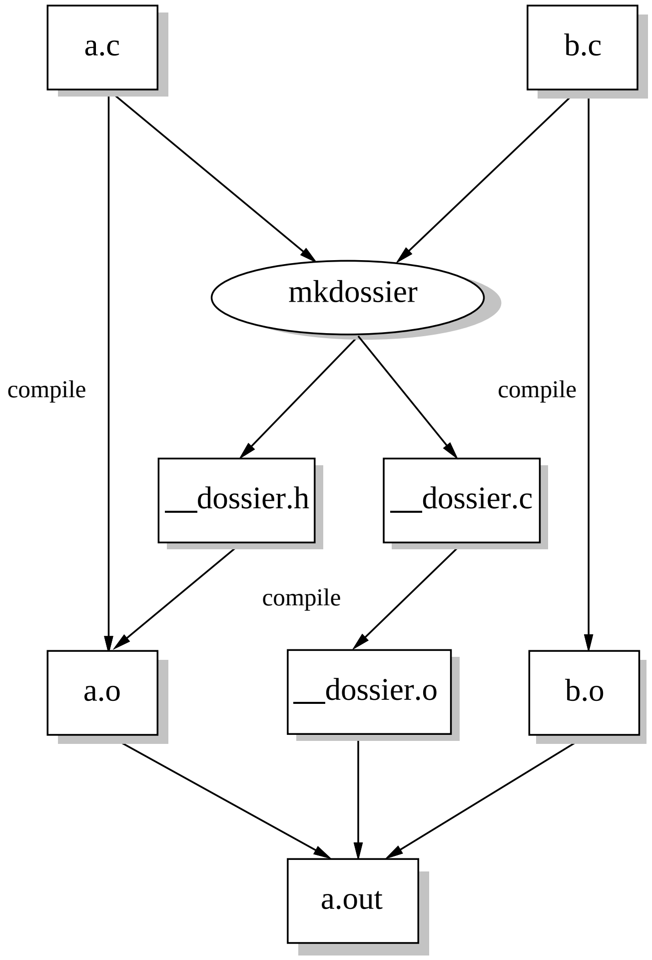

Figure 3: Generating dossiers with mkdossier

```
class Dossier;

extern Dossier __D_App;
extern Dossier __D_ArgVec;
extern Dossier __D_CPP;
```

Figure 4: Declaration of dossiers

```
#include "__dossier.h"
#include <dossier.h>

static Dossier* __D_App_parents[] = { 0 };
static Dossier* __D_App_children[] = { &__D_CPP, 0 };

Dossier __D_App = Dossier(
    "App",
    "/master/iv/src/bin/mkdossier/App.h", 9,
    __D_App_parents,
    __D_App_children
);

static Dossier* __D_ArgVec_parents[] = { 0 };
static Dossier* __D_ArgVec_children[] = { 0 };

Dossier __D_ArgVec = Dossier(
    "ArgVec",
    "/master/iv/src/bin/mkdossier/ArgVec.h", 4,
    __D_ArgVec_parents,
    __D_ArgVec_children
);

static Dossier* __D_CPP_parents[] = { &__D_App, 0 };
static Dossier* __D_CPP_children[] = { 0 };

Dossier __D_CPP = Dossier(
    "CPP",
    "/master/iv/src/bin/mkdossier/CPP.h", 9,
    __D_CPP_parents,
    __D_CPP_children
);

static Dossier* exportedClasses[] = {
    &__D_App,
    &__D_ArgVec,
    &__D_CPP,
    0
};

static int dummy = Dossier::merge(exportedClasses);
```

Figure 5: Initialization of dossiers

merges the static array into a global array containing pointers to all known dossiers so that the "Dossier::classes" function will work correctly even if the programmer compiles and links multiple "_dossier.c" files into the executable image.

Our current implementation generates the file "_dossier.h" because the compiler does not know about dossier declarations. In our proposed language extension, the compiler would automatically import dossier declarations so that the application would not have to include "_dossier.h".

The current implementation of mkdossier calls the C preprocessor to put preprocessed copies of all the source files passed to mkdossier in a temporary directory. The preprocessor strips comments, includes header files, and expands macros so mkdossier can see the same code the C++ compiler sees when it compiles the source files. When mkdossier scans a preprocessed file, it conducts a regular expression search for the keyword "class" followed by text that looks like a class definition. When mkdossier finds a match, it extracts information about that class from its definition. Once mkdossier has scanned all of the preprocessed files, it writes the collected class information to the files "_dossier.h" and "_dossier.c".

Currently we explicitly tell mkdossier which dossiers it should define so we can run mkdossier on a library's source files and include the compiled dossiers in the library. We can then run mkdossier on an application's source files without mkdossier generating duplicates of the library's dossiers. Specifying which dossiers to generate is inconvenient; we are therefore modifying mkdossier to output a dossier for a class only if at least one of the source files defines a non-inlined member function of that class. AT&T cfront 2.0 already uses a similar heuristic to decide when to generate a virtual function table.

4 Experience

We used the Unidraw library [4] as a test bed because it already defines symbolic class identifiers for most of the classes in the library. We changed the typedef ClassId from "unsigned int" to "class Dossier*" and the symbolic class identifiers from integer constants to variables. This change allowed us to use most of the library code unchanged, including all the virtual "GetClassId" member functions. We only had to replace a couple of switch statements by if statements. We replaced the virtual "IsA" member functions, which were redefined in every class, with non-virtual "IsA" member functions defined only in the base classes. The application "drawing" that uses the Unidraw library ran without error after we made these changes, thereby demonstrating that our approach is a practical way to give an application runtime access to type information. Figure 6 shows example code fragments from Unidraw that illustrate how we changed Unidraw classes to use dossiers.

We considered automatically generating a "GetClassId"-like function for existing class types. We could have written a tool to produce new header files with the function declaration added and modified mkdossier to produce the function definitions along with the dossiers. However, we decided that manually adding a single function definition in every class was not a significant problem. What is critical is that mkdossier automatically generate the type information so that the programmer does not have to define it.

5 Language extension

If a programmer wants to use class types developed externally, it is inconvenient to modify someone else's header files. A simple language extension would eliminate the need to change class definitions and also make it possible to obtain type information for typedef names in addition to class types.

```
typedef class Dossier* ClassId;

extern ClassId ELLIPSE_COMP;
extern ClassId ELLIPSE_VIEW;

ClassId ELLIPSE_COMP = &__D_EllipseComp;
ClassId ELLIPSE_VIEW = &__D_EllipseView;

class EllipseComp : public GraphicComp {
public:
    EllipseComp(SF_Ellipse* = nil);

    virtual ClassId GetClassId();
//  boolean IsA(ClassId);   -- actually inherited from Component
};

boolean Component::IsA (ClassId id) {
    return GetClassId()->isA(id);
}

ClassId EllipseComp::GetClassId () { return ELLIPSE_COMP; }

ellipse.c:116:    if (tool->IsA(GRAPHIC_COMP_TOOL)) {
```

Figure 6: Examples of dossiers' use in Unidraw

We propose that a read-only member "dossier" of type Dossier* be predefined for all user-defined types. By making "dossier" predefined instead of reserved, we avoid disturbing the behavior of any existing code. The programmer would use the syntax "*typename*::dossier" to access information for a class type or typedef and the syntax "*object*.dossier" or "*object*->dossier" to access information for a class object.

Applications need not import the Dossier interface to use a class object's "dossier" member. An application could compare the value of "*object*->dossier" with the value of "*typename*::dossier" to identify a class object's data type. Including "dossier.h" makes additional information about the class object's data type available.

Additionally, we propose that the compiler treat "*object*->dossier" as equivalent to "*object*.dossier" if the class does not have a virtual table and equivalent to a virtual function call if the class has a virtual table. For such classes, "*object*->dossier" will return the dossier associated with the class object's dynamic data type even if the compiler cannot determine the type statically. We call the "dossier" member of such classes a "virtual member variable." One possible implementation would generate a unique virtual table for every class and place the address of the class's dossier in the first slot of the virtual table.

An analysis of the Unidraw library revealed that only two out of 115 Unidraw classes would have shared their parent's virtual table if they had not defined a virtual "GetClassId" member function. We expect that requiring every class to have a unique virtual table whether or not it could have shared its parent's virtual table will cause only a small increase in the size of executables.

6 Future Work

Mkdossier must rescan all the source files whenever class information changes. If mkdossier could rescan only the source files that the compiler has to recompile, mkdossier would run faster. What we need is a way for mkdossier to remember the type information it collected last time.

We can make the previously collected type information available to mkdossier by compiling and linking the file "_dossier.c" into the mkdossier executable after each run of mkdossier. To be practical, this approach requires the availability of shared libraries so that we can minimize the disk space occupied by many instances of mkdossier. Shared libraries allow us to store only the information that actually differs among all of the instances.

Alternatively, we could store the type information outside of mkdossier in an external file or in a database server. When mkdossier starts up, it reads the previous type information from the external file or the database server. When mkdossier shuts down, it writes the updated type information to the external file or database server. We plan to investigate which method would be the best method to let mkdossier scan files incrementally.

7 Summary

We have defined Dossier, an interface for accessing type information at runtime. We have implemented mkdossier, a tool that generates a C++ source file containing the type information. To demonstrate the practicality of our approach, we modified the Unidraw library and a Unidraw-based application to use dossier information. Finally, we proposed a simple extension to C++ that would provide a uniform method to access information for classes and typedefs.

References

[1] Keith E. Gorlen. An object-oriented class library for C++ programs. In *Proceedings of the USENIX C++ Workshop*, pages 181–207, Santa Fe, NM, November 1987.

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

[3] Joel McCormack, Paul Asente, and Ralph R. Swick. *X Toolkit Intrinsics—C Language Interface*. Digital Equipment Corporation, March 1988. Part of the documentation provided with the X Window System.

[4] John M. Vlissides and Mark A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[5] Andre Weinand, Erich Gamma, and Rudolf Marty. ET++—An object-oriented application framework in C++. In *ACM OOPSLA '88 Conference Proceedings*, pages 46–57, San Diego, CA, September 1988.