Non-Intrusive Object Introspection in C++ — Architecture and Application*

Tyng-Ruey Chuang **Y. S. Kuo** **Chien-Min Wang**

Institute of Information Science
Academia Sinica
Nankang, Taipei 115, Taiwan
+886 2 788 3799 ext. 1608, 1820, 1703
{trc, yskuo, cmwang}@iis.sinica.edu.tw

ABSTRACT

We describe the design and implementation of system architecture to support object introspection in C++. An introspective object permits observation and change to its own state by a general mechanism that is applicable to objects of all classes. This general mechanism allows the construction of applications where objects are late-binding and the interactions between them are highly dynamic.

Unlike Java, which provides full support for object introspection, C++ has limited built-in introspective capability via its Run-Time Type Information (RTTI) and related facilities. For example, in C++ one cannot query an object for methods that can be applied to it. We show how such introspective information can be collected at compile-time by parsing class declarations, and be used to build a supporting environment for object introspection at run-time.

Our approach is non-intrusive because it requires no change to the original class declarations and libraries, and it guarantees compatibility between objects before and after the addition of introspective capability. This is important if one wants to integrate third-party class libraries, which are often supplied as black boxes and allow no modification, into highly dynamic applications. We show two applications that are built on top of our introspective environment. The first is a generic facility for automatic I/O support of complex C++ objects. The other is a class exerciser that allows interactive execution of dynamically loaded C++ class libraries.

*This work has been supported, in part, by National Science Council, Taiwan, under contracts NSC 85-2221-E-001-009, NSC 85-2221-E-001-010, NSC 86-2213-E-001-003, and NSC 86-2213-E-001-005. This paper is available on-line as technical report TR-IIS-97-015 from the Institute of Information Science, Academia Sinica, via `http://www.iis.sinica.edu.tw`.

KEYWORDS

Object Introspection, Software Reuse and Integration, Object-Oriented Software Development.

1 MOTIVATION

Many object-oriented programming languages, such as CLOS [18], Java [5, 10], Objective C [17], and Smalltalk [4], provide introspective language features that allow the state of an object to be observed and altered by means of a general mechanism that is equally applicable to objects of all classes. In these languages, the binding between a method and the object to be applied with can be delayed until run-time, and the binding requires no static type-checking of the object and the method. This is often called dynamic-binding and it makes easy the construction of applications of which classes are dynamically loaded and executed. Several kinds of applications need dynamic-loading of classes, as we will show later in this paper. For now, let's consider an object-oriented development environment to be such an application since it will need to compile, link, execute, and debug class definitions on-line.

The C++ programming language does not support object introspection [20]. It does provide Run-Time Type Information (RTTI), a run-time class identification mechanism, and virtual function, a mechanism for run-time resolution of method implementation for polymorphic objects. However, these mechanisms are limited in their functionalities since they do not allow full access to an object. For example, one cannot query an object for applicable methods, nor can one gain full access to data members of the object.

These problems are usually solved by using a meta object framework, such as the System Object Model (SOM) from IBM [8], the Common Object Model (COM) from Microsoft [19], or other similar framework. Framework of this kind requires introspective objects to belong to classes which are either derived from a root "Object" class, or the classes themselves are instances of some meta "Class" class. This creates difficulty when integrating existing class libraries that are developed without using the framework. It becomes worse if the class libraries are provided by third-party vendors and are supplied with no source code. Another disadvantage of the above mentioned framework is that objects with introspective capability are not compatible with ordinary objects without the capability. For example, the memory layout of an object is changed once introspection functionalities are added. It may also respond differently to existing methods.

Our goal is to introduce object introspection to existing C++ classes without intruding the original class library, including derivations of the class declarations and memory layouts of the class instances. If achieved, an object will function the same way whether or not it is capable of introspection. Except that now in addition we can invoke methods or access states upon introspective objects using a general mechanism that bypasses the C++ static type-checking mechanism.

This paper is organized as the following. We first discuss in Section 2 background and related work in bring object introspection and reflection to C++. We then outline the system architecture of our non-intrusive scheme of C++ object introspection in Section 3. Section 4 discusses important implementation issues. Section 5 describes two applications that are developed upon the introspective C++ environment. We then conclude this paper with Section 6.

2 BACKGROUND AND RELATED WORK

In [14], *reflection* is defined as the integral ability for a program to observe or change its own code as well aspects of its programming language (syntax, semantics, or implementations) at run-time. A programming language is said to be reflective if it provides its programs with reflection. An important concept in reflective programming languages is *reification*, the process by which aspects of an executing program are brought up using a representation that is expressed in the language and made available to the program. Furthermore, the reification data are causally connected to the related reified aspects such that a modification to one of them affects the other. Few programming languages provide the full power of reflection since reflection is a very powerful concept and its true implication often is not clear. However, several languages, such as Lisp and Prolog, do have limited reflective language features and are able to treat programs as data and to evaluate reification data at run-time.

Object introspection, in the context of object-oriented programming languages, is the ability to observe and change the state of an object using reflection. The concept of introspection is more restricted than reflection because it adheres to the original syntactical, semantical, and implementational aspects of the source language. It just provides a window to the object states of the current execution of a program, and allows changes to them by means of a general gateway to existing legitimate interfaces. For example, using object introspection, one can query, and execute if it exists, an object for a particular method. However, object introspection does not mean the ability to add new methods or modify existing ones for a class (though such ability often can be simulated, if with difficult, in some introspective languages). Therefore, object introspection maintains the semantic integrity of a programming language but open up its programs for general access. Object introspection allows one to construct applications that are more dynamic, and provides avenues for integration of diverse applications. Open implementations [11, 12] of class libraries, for instance, will be most natural when using object introspection.

Some object packaging frameworks, most notable SOM [3, 7, 8] and COM [19], add object introspection and reflection to the C++ programming language. However, they require applications using these frameworks to follow their respective class hierarchy. For example, SOM requires all classes with SOM capability to derive from the SOMObject class, and COM all components with COM capability to equip with the IUnkown interface. Hence object introspection cannot be used for applications or class libraries that are developed without using these frameworks. We aim to provide object introspection to C++ classes without requiring the classes to be derived from or augmented with extra declarations.

There are several proposals and projects on meta object protocols for C++, see for example [1, 6, 9]. They aim to bring full power of reflection to C++ and often require special implementations of the C++ compiler and run-time system (since they either add language extensions or change the language semantics). We seek to develop a framework of object introspection that adhere to the semantics and implementation of C++ [13, 20] and can be used with existing C++ compilers. Application developers should be able to add object introspection to their applications without requiring changes to their existing class declarations and definitions. Nor should they worry about the new C++ semantics and implementations altering the integrity of their applications.

3 SYSTEM ARCHITECTURE

Providing object introspection for C++ is difficulty because C++ objects carry no type information during run-time. (The only exception is the virtual function mechanism for polymorphic objects and the associated RTTI facility, which are limited in functionalities.) A non-intrusive object introspection facility for C++ is even more challenging because one is not allowed to augment the existing class declarations so that type information will be automatically attached to each instance to help introspective operations. The approach we adopt is to define for each class a separate meta object that completely captures information of the class for introspection purposes. Introspective operations on instances of the class are then conducted via going through to the corresponding meta object, which has all the necessary information at run-time.

Before further discussion on the architecture of such an introspective system, let's see what a typical introspective operation in the system looks like, and compare it to the usual C++ method invocation. Let `BSTree` be a class whose instances are binary search trees of which each non-null tree node stores a character string. Suppose class `BSTree` provides a `new` constructor for building an empty tree and an `insert` public method for inserting a new character string into the tree. Then the following C++ code segment builds a new tree `p` and inserts a character string "Sinica" to it.

```
BSTree *p = new BSTree;
p->insert("Sinica");
```

Note that in the above program segment, at compile-time, `p` is known to be an instance of `BSTree`, and the `insert` method is applicable to it.

For introspective operations, however, we cannot assume the binding between `p` (the object) and `BSTree` (the class) is available at compile-time. It may be the case that the class declaration for `BSTree` is not even available at compile-time, and that only the name of the class (a character string "BSTree") is known at run-time. By using our introspective environment, we can achieve the same effect of the above program segment by the following.

```
void *p;
Klass bstree = getClass("BSTree");
p = bstree.new();

Method insert = getMethod(bstree, "insert");
void* argv[] = { "Sinica", 0 };
bstree.invoke(p, insert, argv);
```

Note that the static type of `p` is now `(void *)`. The fact that it points to an instance of class `"BSTree"` is revealed only at run-time. Furthermore, the invocation of method `insert` upon it is via the meta object `bstree`, which will contain all necessary information of class `"BSTree"`.

It is now clear the a non-intrusive introspective environment for C++ has two parts. One is the meta object mechanism which includes declarations of meta classes (such as `Klass` and `Method` above) and the associated supporting libraries (implementations of `getClass`, `invoke` and so on). The other part is the generation of meta objects for classes in need of introspective operations. The part about meta class declarations and libraries is class-neutral and is available at application development time. If all introspective classes are known at application development time, then the code for constructing meta objects can be prepared at compile-time, though meta objects themselves will not materialize until run-time. The generation of the code to produce meta objects can either be manual or automatic. This situation is described in Figure 1 where each stand-alone executable includes a self-contained introspective run-time environment. We call applications of this kind the tightly-coupled ones. Object introspection here complements the usual C++ data access and method invocation mechanism. On the other hand, classes can be dynamically loaded at run-time for their functionalities. In such situations, generation of the corresponding meta objects occurs at run-time, and the generation process has to be automatic. We call applications of this kind the loosely-coupled ones. Here, binding between an object and its associated class is dynamic, and introspection is the normal way of interacting with objects. See Figure 2 for an illustration.

In both the tightly-coupled and loosely-coupled models, the generation of meta objects will need access to the original class declarations but must not modify them. Also note that in both cases, applications are developed using the *original* class declarations, with the addition of the meta class declarations which are fixed. Currently we generate the meta object code automatically by using a parser-based analyzer that extract needed information from application class declarations.

4 IMPLEMENTATION ISSUES

As we have shown above, our design of non-intrusive introspective environment consists of two parts. One is the meta object mechanism which includes declarations of meta classes and their implementations. The other part is the automatic generation of meta objects for classes in need of introspective operations. We discuss in this section several important implementation issues and the solutions we have adopted.

4.1 Meta Class Interface and Library

An application interacts with the introspective environment by using methods defined in the meta classes. However, the interfaces for interaction often carry less type information than what is desirable. As an example, in Section 3 we show how to get the class information for `BSTree` by passing a *character string* `"BSTree"` to the introspective environment. A character string certainly does not say much about the class it is associated with (except its name). Again, to access method `insert` from the meta object for class `BSTree`, we use a *charac-*

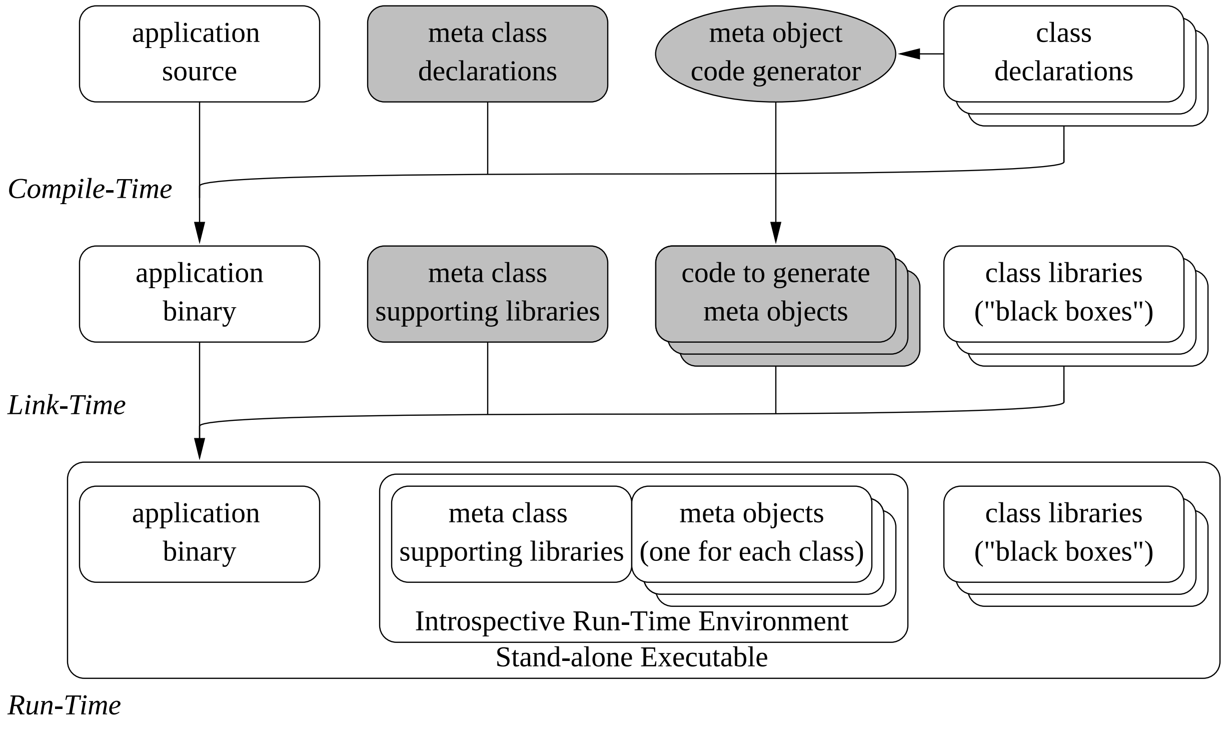

application source
meta class declarations
meta object code generator
class declarations
Compile-Time
application binary
meta class supporting libraries
code to generate meta objects
class libraries ("black boxes")
Link-Time
application binary
meta class supporting libraries
meta objects (one for each class)
class libraries ("black boxes")
Introspective Run-Time Environment
Stand-alone Executable
Run-Time

Figure 1: The tightly-coupled model of introspective applications.

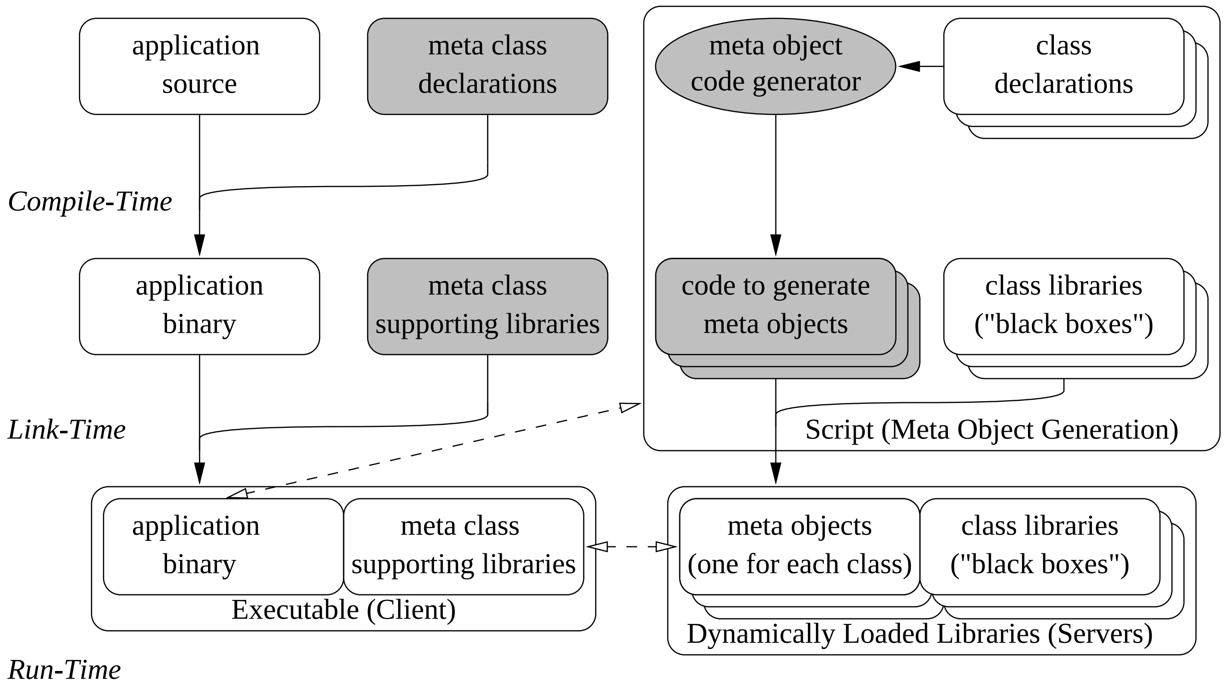

application source
meta class declarations
meta object code generator
class declarations
Compile-Time
application binary
meta class supporting libraries
code to generate meta objects
class libraries ("black boxes")
Link-Time
Script (Meta Object Generation)
application binary
meta class supporting libraries
Executable (Client)
meta objects (one for each class)
class libraries ("black boxes")
Dynamically Loaded Libraries (Servers)
Run-Time

Figure 2: The loosely-coupled model of introspective applications.

ter string "insert" as an argument. The interfaces are typeless because they must serve requests to all kinds of user-defined classes whose properties are not known at compile-time. With this understanding, we now list several issues in the implementation.

Meta classes for class and method. The meta class for class (Klass, as shown above in Section 3) must store the name of the class, pointers to meta objects of its base classes, methods to get names and memory offsets of all its data members, methods to get names and implementations of all its member functions (*i. e.*, getMethod), and several instance conversion routines between this class and its superclasses and subclasses. Several member functions of class Klass are virtual because they require different implementations for different classes. For example, getMethod is a virtual function, and each meta object corresponding to an introspective class will be an instance of a class derived from Klass which actually defines the implementation of getMethod. Again, the invoke method in class Method is virtual as well, and each derived class of Method defines its own implementation. Making these methods virtual helps to reduce work when generating code for meta objects.

Polymorphic object pointers. To correctly access an object, the introspective environment must have the dynamic type information of the object. However, note that all object pointers are treated as pointers of type (void *) when interfacing with the meta objects. Hence, a function is needed for each class to get the dynamic class names for objects of its class. This function is stored in the meta object of the class. Suppose we have a class B, then the meta object for class B will contain a function dynamicType(pObj) defined by

```
virtual const char const *dynamicType
  (void const *self) {
  return typeid(*(B *) self).name(); }
```

that returns the dynamic type name (a character string) of its instance. Function typeid above is from the standard C++ RTTI facility.

Base and derived classes. The content of an object consists of its data members, and those of its bases as well. In order to access the base's data members, the introspective environment has to perform a "up cast" operation that adjusts the object pointer. The up cast function is stored in the meta object of the class. For example, the following function will cast an instance of a derived class D of B to an instance of B.

```
static void *fromDtoB(void const *self)
  { return (B *)(D *) self; }
```

If a class has several bases (*i. e.*, multiple inheritance), then all the up cast functions will be stored in the meta object of B. The cast functions are stored in a table and indexed by names of the superclasses.

Similarly, the following "down cast" routine is stored in the meta object for B as well.

```
static void *fromBtoD(void const *self)
 { return dynamic_cast<D*>((B*) self);}
```

It is used to adjust a polymorphic object pointer with known static type (B *) to its dynamic type (D *). The adjustment occurs, for example, after the dynamic type name "D" is resolved by a call to dynamicType. All accesses to data members pointed to by an polymorphic object pointer starts with this adjusted pointer.

Other details. C++ provides abstract and virtual base class. Objects of an abstract class cannot be constructed directly, and base objects of virtual classes share their storage. Information regarding whether a class is abstract or virtual or not will be recorded in its meta object. This means that the introspective environment has to check first with this information when performing data access or object construction.

4.2 Meta Object Generation and Management

For each class in need of introspective operations, the run-time environment needs a meta object of the class. The meta object can be constructed manually, or produced automatically from the program text where the class is declared. We use a parser-based scanner that produces program code from class declarations such that, when the code is executed at run-time, will generate meta objects. Note that code generation and meta object generation do not happen at the same time for the tightly-coupled applications illustrated in Figure 1. The programming language used for code generation may not even be C++. If it uses C++, it may use a different compiler for the one used by the application. Hence, one must take care in generating code to generate meta objects correctly, especially when regarding memory layout of their instances.

Memory layout. The memory layout of an object is calculated by employing a method that relies on a C++ compiler's ability to treat a suitable aligned absolute address as the starting address of any C++ object [15]. For example, the following code computes the offset (in units of char) of the data member a within an object of class A:

```
offset_of_a_in_A = (char *) &(((A *) 64)->a)
                 - (char *)   ((A *) 64)
```

where 64 can be any well-aligned absolute address. Note that, in order to calculate offsets of private members, the generator must disable access control employed by the class. This can be done by altering the original class declaration in several ways. One can delete all the `private` specifiers in the class declaration, hence explores all data members to outside world. A better way is to insert a friend function to the class declaration and gain access to all data members without compromising access control of the class too much [21]. The generator inserts as a friend function an initialization routine to the class to calculate data member offsets and to produce a meta object for the class. Note that this friend function will need to be compiled with the augmented class declaration. We list additional issues when augmenting class declarations.

Nested class declarations. If the class declarations are nested, the initialization routine for the enclosing class, though declared as a friend function, will not be able to access data members of the enclosed classes. (They are out of scope.) For these cases, we have the generator inserts initialization routines to the enclosed classes as well, and have them declared as friend functions both in the enclosed and enclosing classes.

Class templates. Since the generator analyzes only class declarations, it cannot know how a class template will be instantiated in a user's program. We require the user to give hints on how a class template will be instantiated. As an example, in Section 3, if `BSTree` is a class template, then one must explicitly specify that `BSTree` will be instantiated with type `int`, in order for the generator to produce the meta object for class `BSTree<int>`.

Object compatibility. Note that we may augment a class declaration in several ways in order for the code generator to work. (For example, we insert a friend function to the class declaration, and we may add additional non-virtual member functions for other purposes.) However, we never add data members or virtual functions to the class declaration. This ensures that objects produced by the original class declaration and the augmented one will always have the same memory layout, at least for the usual C++ object model [13]. Furthermore, the augmented class declaration is used only by the generated code and is inaccessible otherwise to the application developers. Developers continue to use the original class declarations and will not aware of the augmented copies.

5 APPLICATIONS

We describe two applications based on non-intrusive introspective C++ environments. The first is a system called *ObjectStream* that provides automatic I/O support for complex C++ objects. The other is a class exerciser called *RunClass* that allows interactive execution of dynamically loaded C++ class libraries. ObjectStream is a system for building applications that are tightly-coupled with their introspective classes. While RunClass itself is a loosely-couple introspective application. Work on ObjectStream and RunClass has been reported elsewhere, although not from the viewpoint of object introspection [2, 22].

5.1 ObjectStream

The C++ standard I/O stream library, by overloading the `<<` and `>>` operators, provides a convenient and type-safe interface for I/O. However, it is only applicable to fundamental types (such as `char`, `int`, *etc.*). If one wants to input/output objects of user-defined classes, one either must extend the stream I/O library by overloading the `<<` and `>>` operators, or define I/O operations as member functions of those classes. Most C++ development environments (such as the Microsoft Foundation Classes) in use today take the latter approach. Their input/output facilities are parts of a pre-defined application framework. Programmers are required to define I/O operations for user-defined classes by following some prescribed procedures — such as deriving user-defined classes from a "persistent" base class and redefining some virtual functions for the purpose of object I/O. This practice takes considerable learning time. Furthermore, programmers have to write code to construct/traverse an object's internal structure in order to perform I/O correctly. When complex objects are involved, this can be tedious and error-prone. By requiring all user-defined classes to derive from a persistent base class, it also prevents pre-built class libraries from performing object I/O. Pre-built class libraries, such as those provided by library vendors but supplied with no source code, often have pre-set class hierarchy and cannot be re-derived from the persistent base class.

In ObjectStream, we provide a generic I/O library that interacts with the introspective run-time environment to automatically traverse an object's internal structure for I/O purpose. The meta class declarations include the follow template I/O operators:

```
template <class T>
Uostream &operator<<(Uostream &os, T &obj) {
```

```
#include <ustream.h>    // Declarations of I/O operators and universal streams.
#include "bstree.h"     // Declarations of BSTree (for Binary Search Tree).

main () {
 AsciiOut  ascout;      // Use ASCII format for output.
 ascout.precision(16);  // Precision is 16 digits wide.
 Uofstream ofile("tree.dat", ascout);  // Create an output stream "tree.dat"
                                       //   on file medium with ASCII format.
 BSTree    myTree;      // The place to hold a BSTree object.
 ...                    // Insertion of elements into myTree omitted here.
 ofile << myTree;       // Store myTree to ofile.
 ofile.close();         // Close the file stream.

 AsciiIn   ascin;       // Use ASCII format for input.
 Uifstream ifile("tree.dat", ascin);   // Use file "tree.dat" as the input
                                       //   stream. The format is ASCII.
 BSTree   *pTree = 0;   // The pointer to hold a restored BSTree object.
 try {
   ifile >> pTree;      // Restore a BSTree object from ifile. After this,
                        //   *pTree and myTree contain the same tree.
 } catch (IOError error) {  error.printMessage(); }  // Catch errors, if any.
 ifile.close();         // Close the input file stream.
}
```

Figure 3: ObjectStream: A working example.

```
  Uwrite(os, (void*) &obj, typeid(T).name());
  return os; }

template <class T>
Uistream &operator>>(Uistream &is, T *&obj){
  Uread (is, (void *) obj, typeid(T).name());
  return is; }
```

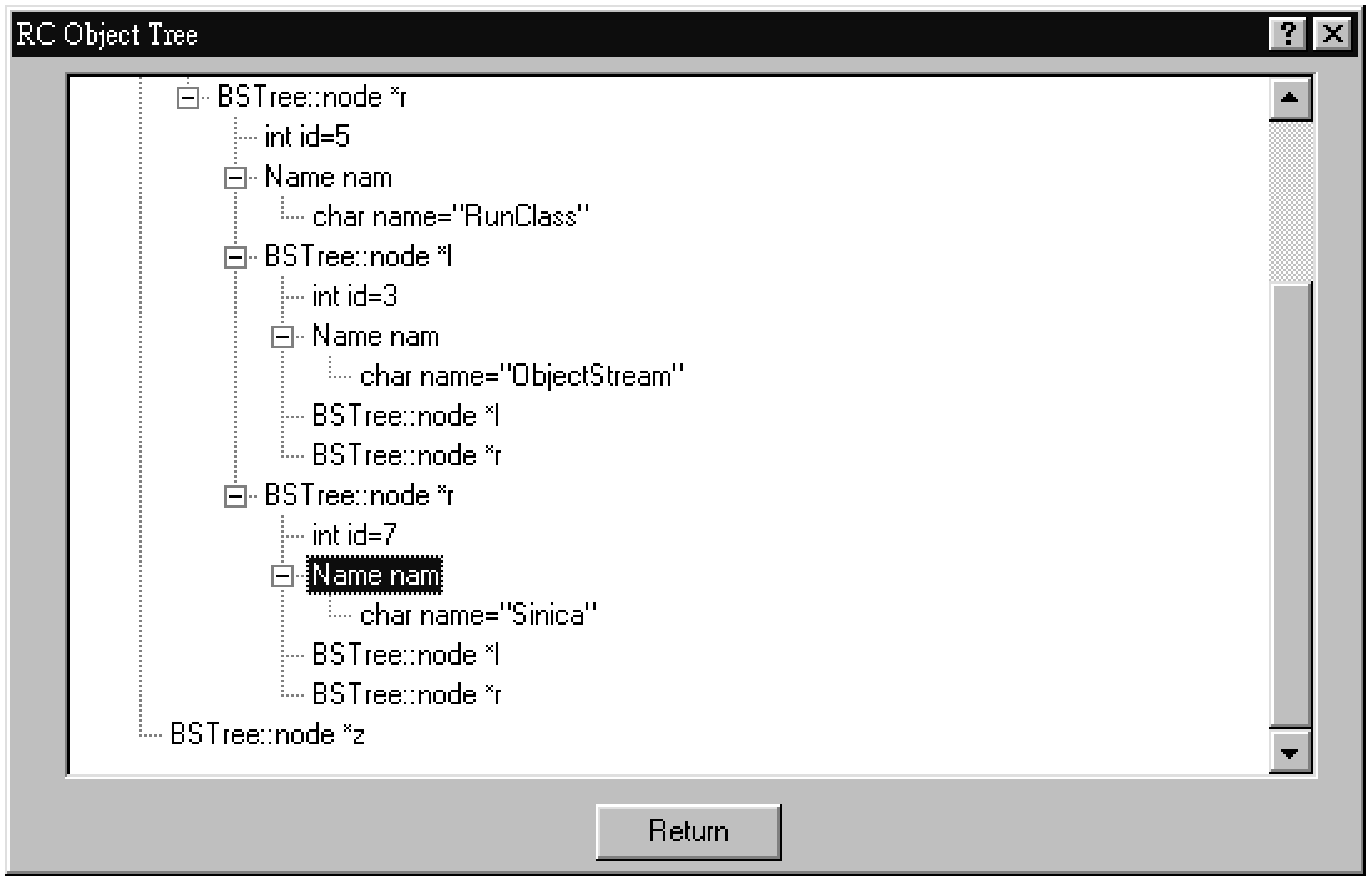
where `Uistream` and `Uostream` are the class names for the universal streams that are capable of input/output objects of all classes. The programmers can then use `>>` and `<<` to input/output objects of user-defined classes.

Note that the object reference is passed as a void pointer, and the class information is passed as a string containing the static class name of the object. `Uwrite` and `Uread` are generic read/write functions that interact with the introspective environment to traverse objects for I/O. The object traversal algorithm used by the library is a depth-first search that looks inside an object for its bases and data members. The traversal algorithm is not that different from those used for garbage collection. Since bases and data members are objects as well, the search is recursive in nature. When the search encounters objects of fundamental types it calls the corresponding primitive routines for I/O. A major issue in the search is to avoid duplication of I/O for objects that have already been visited. For this purpose, we maintain a dictionary of object reference that maps between an object's internal memory address and its external object ID. When output, the library first checks with the object reference dictionary using the object's memory address as a key to see whether the object has been output or not. If so, only the object's ID is output. Otherwise a new ID is assigned to the object and a new entry of (address, ID) pair is entered in the dictionary. Similarly, when performing input operation, if the library see only an object ID, then it uses it as a key to look for the object's memory address in the dictionary.

In Figure 3, we show a program fragment to illustrate how to restore/store user-defined objects from/to universal streams. The program constructs a binary search tree, outputs the tree to a file using the `<<` operator, and read the same tree back from the file using the `>>` operator. ObjectStream also comes with an "object stream browser" that can be used to open an object stream and display objects in the stream. See Figure 4 for a screen shot.

5.2 RunClass

Object-oriented development is often characterized by the development and use of a large set of reusable classes. Current object-oriented CASE environments facilitate software reuse by providing tools to inspect and access libraries. Using these CASE tools, one can easily



```
RC Object Tree
    ⊟ BSTree::node *r
        int id=5
        ⊟ Name nam
            char name="RunClass"
        ⊟ BSTree::node *l
            int id=3
            ⊟ Name nam
                char name="ObjectStream"
            BSTree::node *l
            BSTree::node *r
        ⊟ BSTree::node *r
            int id=7
            ⊟ Name nam
                char name="Sinica"
            BSTree::node *l
            BSTree::node *r
BSTree::node *z
```

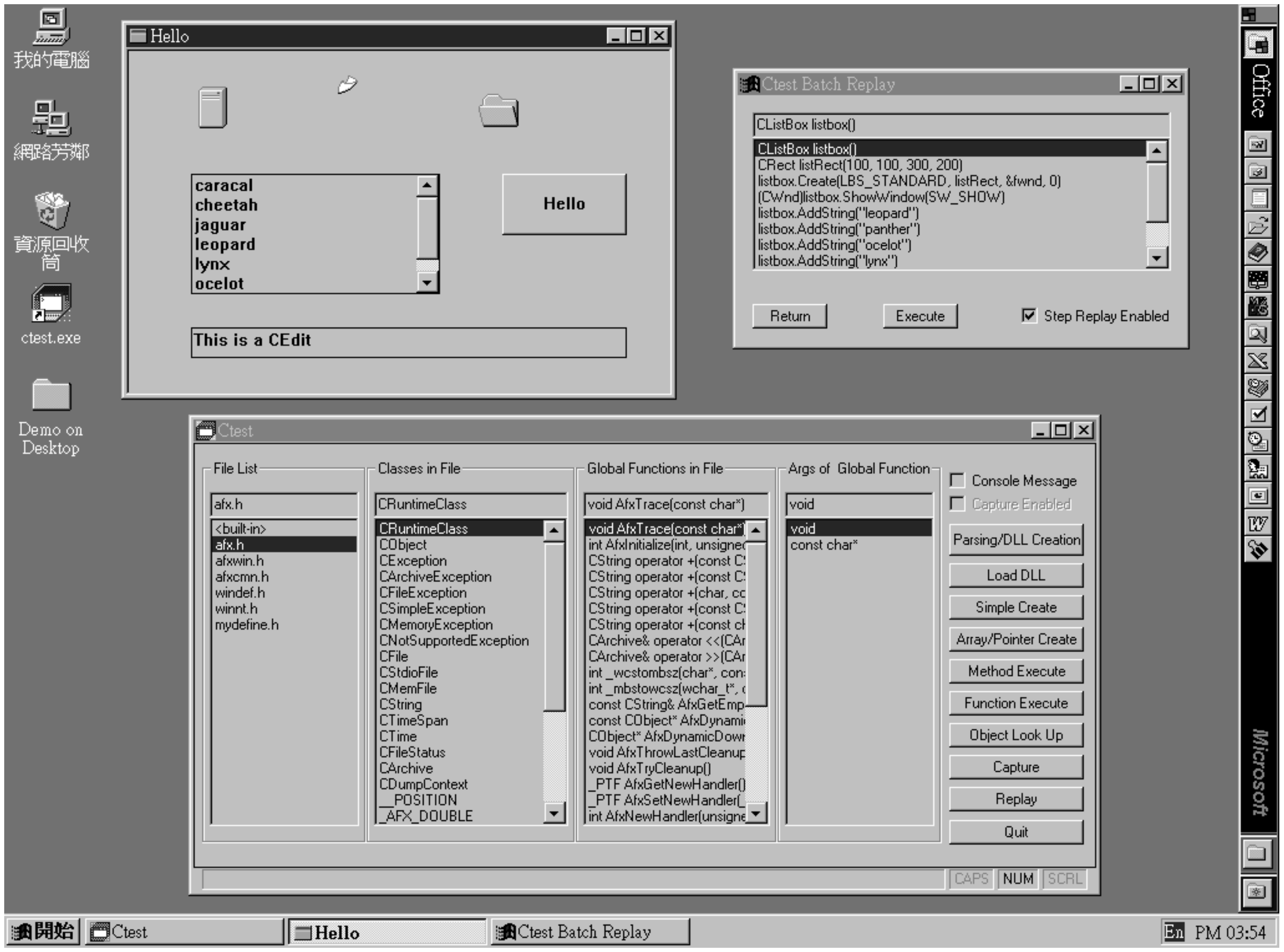
Figure 4: Browsing an object stream.

locate desired classes and retrieve information of interest. However, invocation of classes (including instantiation of objects from classes and invocation of methods on objects) can only be made in the form of traditional programming. In other words, in general one has to write an application program just in order to test a class. This is fine if the application programmer really wants to use the class. However, in many situations, we may just want to invoke a class to understand its functionality, to see an undocumented feature, or too verify its correctness. For these situations, an interactive, easy-to-use environment for "exercising" classes appears to be more convenient than the traditional edit-compile-run programming environment.

RunClass is a class exerciser that, when taking a set of classes as input, allows a user to create objects for given classes, execute methods on specified objects, and examine their contents interactively. RunClass presents an easy-to-use graphical user interface to users with class names and method names displayed on the screen. Users can then select desired classes and methods with a pointing device without memorizing their names. RunClass also manages objects that are created. Objects are grouped according to classes for users' convenience. There are several applications of RunClass. It can be used as a demonstration tool, interactively showing the functionalities of a class library. It can also be used for testing and maintenance purposes because it allows easy access to classes that are unfamiliar to the programmers.

The implementation of RunClass is straightforward upon an introspective C++ environment. It needs a graphical user interface to take commands from the users and passing them to the underlining introspective environment. Results from the command are then displayed for users to interpret. It also needs a command manager that can capture a sequence of commands and log their effects. The command sequence can then be replayed later for regression testing on newer version of the class library.

Figure 5 show a snapshot of using RunClass to exercise the Microsoft Foundation Classes (MFC) [16]. On the right-hand side is the reply dialog, which shows some



caracal
cheetah
jaguar
leopard
lynx
ocelot

This is a CEdit

```
CListBox listbox()
CListBox listbox()
CRect listRect(100, 100, 300, 200)
listbox.Create(LBS_STANDARD, listRect, &fwnd, 0)
(CWnd)listbox.ShowWindow(SW_SHOW)
listbox.AddString("leopard")
listbox.AddString("panther")
listbox.AddString("ocelot")
listbox.AddString("lynx")
```

File List: afx.h
<built-in>
afx.h
afxwin.h
afxcmn.h
windef.h
winnt.h
mydefine.h

Classes in File: CRuntimeClass
CRuntimeClass
CObject
CException
CArchiveException
CFileException
CSimpleException
CMemoryException
CNotSupportedException
CFile
CStdioFile
CMemFile
CString
CTimeSpan
CTime
CFileStatus
CArchive
CDumpContext
__POSITION
_AFX_DOUBLE

Global Functions in File: void AfxTrace(const char*)
```
void AfxTrace(const char*)
int AfxInitialize(int, unsigne
CString operator +(const C
CString operator +(const C
CString operator +(char, cc
CString operator +(const C
CString operator +(const ch
CArchive& operator <<(CAr
CArchive& operator >>(CAr
int _wcstombsz(char*, con
int _mbstowcsz(wchar_t*, c
const CString& AfxGetEmp
const CObject* AfxDynami
CObject* AfxDynamicDowr
void AfxThrowLastCleanup
void AfxTryCleanup()
_PTF AfxGetNewHandler()
_PTF AfxSetNewHandler(_
int AfxNewHandler(unsigne
```

Args of Global Function: void
void
const char*

Figure 5: RunClass: exercising the Microsoft Foundation Classes.

operations captured previously. They appear in the form of C++ statements. On the left-hand side are a window and four MFC "controls" (animate control, list box, push button, and edit control) inside the window, which are all constructed by replaying the captured operations. By using the dialog at the bottom, users can manipulate the windows and controls, like pause and play the animate control, add items to the list box, and so on via interactive method invocations. This MFC exerciser using RunClass appears to be as an attractive CASE tool for training novice MFC programmers.

6 CONCLUSION

We present the concept of object introspection and a framework of non-intrusive implementation in C++ that works with standard compilers and existing class libraries. Several important implementation issues are discussed and two substantial applications are demonstrated. It is shown that object introspection can be added to C++ in a straightforward way and it makes easy application reuse and integration.

ACKNOWLEDGEMENTS

We thank Chuan-Chieh Jung and Wen-Min Kuan for their effort in implementing ObjectStream, and Chin-Chuan Hsu and Wei-Hsueh Lai for their effort in implementing RunClass.

REFERENCES

[1] Shigeru Chiba. A metaobject protocol for C++. In *Conference proceedings of Object-Oriented Programming Systems, Languages and Applications*, pages 285–299. Austin, Texas, USA, October 1995. ACM Press.

[2] Tyng–Ruey Chuang, Chuan–Chieh Jung, Wen–Min Kuan, and Y. S. Kuo. ObjectStream: Generating stream–based object I/O for C++. In *24th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Asia '97)*. Beijin, China, September 1997.

[3] Scott Danforth and Ira R. Forman. Reflection on metaclass programming in SOM. In *Conference proceedings of Object-Oriented Programming Systems, Languages and Applications*, pages 440–452. Portland, Oregon, USA, October 1994. ACM Press.

[4] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Welsey, 1993.

[5] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison–Wesley, August 1996.

[6] Brendan Gowing and Vinny Cahill. Meta–object protocols for C++: The Iguana approach. In Gregor Kiczales, editor, *Proceedings of the Reflection 96 Conference*, pages 137–152. San Francisco, California, USA, April 1996. Proceedings available from Xerox Palo Alto research Center.

[7] Jennifer Hamilton, Robert Klarer, Mark Mendell, and Brian Thomson. Using SOM with C++. *C++ Report*, 7(6):40–45, 65, July–August 1995.

[8] International Business Machines. *SOMobjects Developer's Toolkit, Programmer's Guide, Volume 1-2*, 2nd edition, December 1996.

[9] Yutaka Ishikawa, Atsushi Hori, Mitsuhisa Sato, Motohiko Matsuda, Jórg Nolte, Hiroshi Tezuka, Hiroki Konaka, Munenori Maeda, and Kazuto Kubota. Design and implementation of metalevel architecture in C++ — MPC++ approach. In Gregor Kiczales, editor, *Proceedings of the Reflection 96 Conference*, pages 153–166. San Francisco, California, USA, April 1996. Proceedings available from Xerox Palo Alto research Center.

[10] JavaSoft. *Java Core Reflection — API and Specification*, January 1997.

[11] Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–10, January 1996.

[12] Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering*, pages 481–490. Boston, Massachusetts, USA, May 1997. IEEE Press.

[13] Stanley B. Lippman. *Inside The C++ Object Model.* Addison–Wesley, 1996.

[14] J. Malenfant, M. Jacques, and F.-N. Demers. A tutorial on behavioral reflection and its implementation. In Gregor Kiczales, editor, *Proceedings of the Reflection 96 Conference*, pages 1–20. San Francisco, California, USA, April 1996. Proceedings available from Xerox Palo Alto research Center.

[15] Robert Mecklenburg, Charles Clark, Gary Lindstrom, and Benny Yih. A dossier driven persistent objects facility. In *USENIX 6th C++ Technical Conference*, pages 265–281. Cambridge, MA, USA, April 1994. USENIX Association.

[16] Microsoft. *Microsoft Foundation Classes*, version 4.2.1, 1997.

[17] NeXTSTEP. *Object-Oriented Programming and the Objective C Language.* Addison-Wesley, November 1993.

[18] Andreas Paepcke, editor. *Object-Oriented Programming: The CLOS Perspective.* MIT Press, 1993.

[19] Dale Rogerson. *Inside COM.* Microsoft Press, 1997.

[20] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, 2nd edition, June 1993.

[21] Walter F. Tichy, Jörg Heilig, and Frances Newbery Paulisch. A generative and generic approach to persistence. *C++ Report*, 6(1):22–33, January 1994.

[22] Chien-Min Wang and Y. S. Kuo. Class exerciser: a basic tool for object–oriented development. In *Proceedings of the 1995 Asia Pacific Software Engineering Conference*, pages 108–116. Brisbane, Australia, December 1995.