Run-time Type Information and Incremental Loading in C++

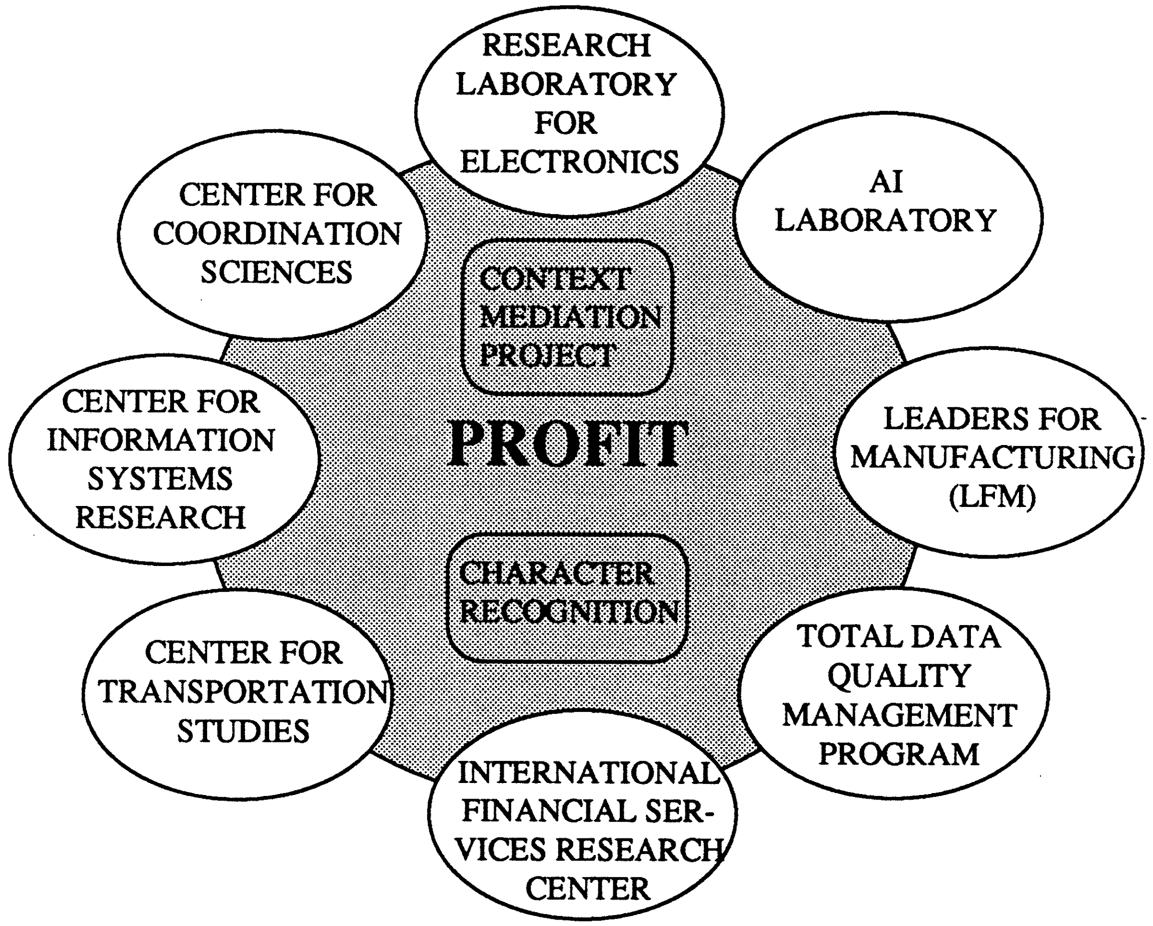
Murali Vemulapati
Sriram Duvvuru
Amar Gupta

WP #3772 September 1993
PROFIT #93-11

Productivity From Information Technology
"PROFIT" Research Initiative
Sloan School of Management
Massachusetts Institute of Technology
Cambridge, MA 02139 USA
(617)253-8584
Fax: (617)258-7579

Productivity From Information Technology (PROFIT)

The Productivity From Information Technology (PROFIT) Initiative was established on October 23, 1992 by MIT President Charles Vest and Provost Mark Wrighton "to study the use of information technology in both the private and public sectors and to enhance productivity in areas ranging from finance to transportation, and from manufacturing to telecommunications." At the time of its inception, PROFIT took over the Composite Information Systems Laboratory and Handwritten Character Recognition Laboratory. These two laboratories are now involved in research related to context mediation and imaging respectively.


RESEARCH LABORATORY FOR ELECTRONICS
AI LABORATORY
LEADERS FOR MANUFACTURING (LFM)
TOTAL DATA QUALITY MANAGEMENT PROGRAM
INTERNATIONAL FINANCIAL SERVICES RESEARCH CENTER
CENTER FOR TRANSPORTATION STUDIES
CENTER FOR INFORMATION SYSTEMS RESEARCH
CENTER FOR COORDINATION SCIENCES
CONTEXT MEDIATION PROJECT
PROFIT
CHARACTER RECOGNITION

In addition, PROFIT has undertaken joint efforts with a number of research centers, laboratories, and programs at MIT, and the results of these efforts are documented in Discussion Papers published by PROFIT and/or the collaborating MIT entity.

Correspondence can be addressed to:

The "PROFIT" Initiative
Room E53-310, MIT
50 Memorial Drive
Cambridge, MA 02142-1247
Tel: (617) 253-8584
Fax: (617) 258-7579
E-Mail: profit@mit.edu

Run-time Type Information and Incremental Loading in C++

September 13, 1993

Abstract

We present the design and implementation strategy for an integrated programming environment which facilitates specification, implementation and execution of persistent C++ programs. Our system is implemented in E, a persistent programming language based on C++. The environment provides type identity and type persistence, i.e., each user-defined class has a unique identity and persistence across compilations. The system provides Run-time Type Information for the user-defined types and it also provides efficient run-time access to the members of an object by generating *maps* of the objects. It also supports incremental linking and loading of new classes or modification of classes existing in the database.

1 Motivation

C++ has proved itself to be an effective programming tool for a wide variety of applications. Many additions to C++ have been proposed to make it more suitable for specific applications. These proposed additions include: Persistence, Exception Handling, Object I/O, Run-time Type Information, Incremental Compilation, Linking and Loading, Environment Support.

E [4] is an extension to C++ designed for writing software systems to support persistent applications. E augments C++ with database types, iterators, collections and transaction support. E mirrors the existing C++ types and type constructors with corresponding database types (db types) and type constructors. Db types are used to describe the types of objects in a database i.e. the database schema. The **persistent** storage class provides the basis for populating the database. If the declaration of a db-type variable specifies that its storage class is **persistent**, then that variable survives across all runs of the program and also across crashes. For transaction support, E provides library calls to begin, commit or abort a transaction. These calls are supported by the EXODUS storage manager [10] which provides basic support for objects, files and transactions.

There are many complex, data-intensive applications such as CAD, CAE and CASE where E is inadequate as the design language and where we need additional facilities such as access to type information of objects at run-time, database querying, incremental linking in of classes etc.

COSMOS [11] is one such application tool for building object-oriented knowledge based systems for engineering applications. It integrates rule-based and object-oriented programming paradigms to provide a uniform framework for processing both procedural and heuristic knowledge. COSMOS provides a framework in which a user can model the domain of his application in terms of classes and their inter-relationships (with relations such as inheritance, composition etc). The domain knowledge is modeled in terms of rules which perform inference over the attributes of a class and also over the classes themselves. The Object Management part of COSMOS, which is built over EXODUS/E platform, provides facilities for dynamic creation of classes and modification of class hierarchy. It also provides persistence for the instances of the classes created. The persistent instances of the classes and the classes themselves (*class objects*) are stored on the EXODUS database.

In the following sections, we present the extensions to E that we have incorporated in the COSMOS object-management system and motivate them.

1.1 Run-time Type Information

So far there have been various attempts to provide a mechanism for accessing type information of objects at run-time. There are several reasons for doing so:

- Support for accessing Inheritance information is often necessary. We would like to know if a pointer to a base class actually points to an instance of a class derived from that base class. If so, we can *cast down* the pointer in order to be able to access a derived class member.
- The exception handling mechanism requires run-time type identification. When an exception is raised, an object of a particular type is *thrown* and in order to match the thrown object with the correct *catch* clause, access to type information is necessary. Also, as it is possible for a *catch* clause to catch a type which is a base class of the thrown object, we need to generate information about inheritance hierarchy.
- Often, the virtual function mechanism is inadequate for type-specific actions.
- In addition to type identification, more information about a type is often necessary. For example, the COSMOS object management system provides information such as the names of the classes and their inheritance relations. It also provides the names of attributes of a class and a map of the class. This is necessary to support the rulebase where the rules can access the attribute values of an object by name.
- Query evaluation needs access to type information of objects.

1.2 Incremental Development

Usually a user changes only a small fraction of class descriptions between successive compilations during the software development phase. If the cost of re-compilation can be made proportional to the size of the change rather than the size of files in which the change occured plus the size of header files included from the files, then the compilation of C++ programs using large libraries would become much faster.

Incremental linking and loading are desirable for systems that are designed to allow a user to extend a running system. For example, in a CAD/CAM system, it can be useful to allow a user to define a new kind of component and add it to a layout currently on the screen. Using traditional compile, load and go techniques, it is difficult to add the new component class without having to create the current layout from scratch after the recompilation. At best, there would be a noticeable disruption of service.

The key problem with incremental linking is to provide a way of adding a new class to a running program. It must be possible to define an interface to a yet-to-be-defined class and it must be possible to invoke operations on objects of the new class.

1.3 Environment Support

An environment can support the programmer in two ways: by supporting the activity of programming (providing tools for designing, specifying, implementing, and debugging programs) and by providing a congenial environment for the final program to execute in.

The standard C/Unix model of creating and running programs is insufficient for supporting a persistent language like E as we shall see below.

Type Persistence In C and C++, type definitions are shared between compilation units via the textual inclusion of header files. Thus there is no notion of a type existing outside of any particular run of the compiler. A given type T used to compile one module may or may not be the same T used to compile another. For E programs, which provides persistence of objects, it is critical that the definition of a type remain consistent across compilations. That is, if a persistent object is created with a type T and later this object is manipulated by a program with a different definition for T, the object will be corrupted. This problem is accentuated in object-oriented systems because each object must carry sufficient information to identify its own type uniquely so as to support virtual function mechanism. Hence object persistence entails *type persistence*. A satisfactory solution to this problem requires an integrated program environment that provides, among other features, a type library in which a type has a unique identity across compilations.

Type Availability Often it is possible for a program to encounter an object whose type was not unknown at the time the program was compiled. In a non-persistent C++ program, the implementation of virtual function dispatch can assume that both the method dispatch table (vtbl) and the method code itself are there in the address space of the program. In the presence of persistence, this assumption breaks down. Suppose a program P1 creates a persistent graph G whose nodes are of type T1 and traverses G to execute a virtual function f() on every node of G. Now another program P2 derives a type T2 from T1 and adds a node of type T2 to G. If P1 is run again, it encounters a 'unknown type' error when it tries to invoke virtual function f() on the newly added node.

The problem here is that the environment does not adequately support persistence. Providing such support would involve storing programs as objects in the database and incremental dynamic linking of the method code. The environment has to keep track of dependencies between programs and the persistent objects created by them.

The remainder of the paper describes our approach to address the above mentioned issues.

2 The Run-time Type Information

C++ supports two kinds of types: fundamental types and derived types [1]. The fundamental types are: **int, char, float, double, long, short**. One can construct an infinite number of derived types from the fundamental types in the following ways:

- *array* of objects of a given type.
- *function* which takes a sequence of arguments of given types and returns an object of given type.
- *pointer* to an object or function of a given type.
- *reference* to an object or function of a given type.
- *class* which contains a sequence of objects of various types, a set of functions for manipulating these objects, and a set of access restrictions.
- *pointer to class member* which identifies members of a given type within an object of a given class.

The C++ type system can be summarised by the following type grammar:

```
T ::=  int | char | float | double | long | short
| T []
| T (T, ..., T)
| T*
| T&
| class T
| T class T::*
where T is any type.
```

Our run-time type support system is based on this type grammar. Since types are to be stored as persistent objects in the database, we need to represent the type information in the form of a class definition. The class *Type* captures the type information as described below.

```
dbclass Type {
public:
  virtual int equal?(Type*)=0;
  virtual int overloadable(Type*){return 0;};
};
```

Class *Type* is a top-level abstract class from which various other types are sub-typed. The method *equal* is a boolean test to test the equality of two types. The method *overloadable* is a test to decide if a type can be overloaded by another type.

The class *Type* is sub-classed into the following classes: *PrimitiveType, ClassType, ArrayType, PointerType, ReferenceType, FunctionType, PointerToMemberType.*

```
dbclass PrimitiveType :public Type {
public:
  dbint ptype; //integer code for each of primitive types
  PrimitiveType (dbint i) {ptype = i;}
  int equal?(Type*);
  int overloadable(Type*);
};
```

```
dbclass ClassType :public Type {
public:
  dbchar *class_name; // name of the class
  Type** base_list; // list of base classes
  MetaClass* mptr; // pointer to a meta_class object
  int equal?(Type*);
  int overloadable(Type*);
  int same (Type*);
  int has_base(Type*, int direct=0);
  int can_cast(Type*);
};

dbclass ArrayType :public Type {
public:
  Type *elem_type;
  dbint size;
  ArrayType(Type* t,dbint sz)
    {
      elem_type = t;
      size = sz;
    };
  int equal?(Type*);
  int overloadable(Type*);
};
```

```
dbclass PointerType :public Type {
public:
  Type *obj_type;
  PointerType(Type* ot){obj_type = ot;}
  ~PointerType();//{delete obj_type;}
  int equal?(Type*);
  int overloadable(Type*);
};

dbclass ReferenceType :public Type {
public:
  Type *obj_type;
  ReferenceType(Type* ot){obj_type = ot;}
  ~ReferenceType(){delete obj_type;}
  int equal?(Type*);
  int overloadable(Type*);
};

dbclass FunctionType :public Type {
public:
  Type *ret_type; // return type
  collection <Type*> arg_type_list; // list of argument types
  FunctionType(Type* rt) {ret_type = rt;}
  ~FunctionType(){delete ret_type;}
  int equal?(Type*);
  int overloadable(Type*);
};
```

```
dbclass PointerToMemberType: public Type {
public:
  MetaClass* mptr; // pointer to the meta_class object
  Type* mem_type; // type of the member
  int equal?(Type*);
  int overloadable(Type*);
}
```

The class *MetaClass* is the class representation of a class type which contains information about the various attributes and methods of a class. This is described in detail in the next section.

2.1 MetaClass

Type Persistence is provided by creating, for each user-defined class, a persistent instance of *MetaClass* which contains information about the user-defined class. The unique ID of this object provides the type identity across compilations. The persistent collection of user-defined types can looked upon as a kind of a *type library* from which an end-user can choose the required classes and build his application.

The class representation for *MetaClass* is described below:

```
dbclass MetaClass {
private:
  MetaClass** public_bases; //list of public base classes
  MetaClass** friends;
  dbchar *class_name;
  Attribute** attr_tbl; //table of member names and types
  collection<root> extent; // the extent of the class
public:
  MetaClass(dbchar* n);
  const dbchar* name() const;
  int add_base(dbchar* base_name,int base);
  delete_base(dbchar*);
  int is_base(dbchar* base_name);
  int add_friend(dbchar* fname);
  Type* get_type(dbchar* an); // get the type of a given attribute
  add_attribute(Attribute*);
  delete_attribute(dbchar *);
  create_instance(dbchar* iname, int pers=1); //create a instance
  delete_instance(dbchar* iname);
  root* find_instance(dbchar* iname);
  generate_code();
}

dbclass Attribute {
  dbchar* name;
  Type* type;
  enum {public, private, protected} access;
  int offset;
}
```

The class *Attribute* describes each member of a class (whether a data or a function member). Each Attribute instance contains an integer offset of that attribute which gives the offset of the attribute from the starting address of the instance. Each *MetaClass* instance contains a pointer to a table of attributes (attr_tbl) which is a map of an object of the class. This facilitates faster access to the attributes of an instance. In the next section we shall see how this map is generated for each user-defined class.

The class *root* is the top-level class in the hierarchy of all the user-defined classes. This class provides the basic scaffolding needed to provide the RTTI and dynamic linking facilities. This class is described in detail in the next section.

Each *MetaClass* class contains an attribute called 'extent' which is the collection of all the instances of the class represented by that *MetaClass* instance. The methods *create_instance* and *delete_instance* manipulate the extent of the class. Each instance is uniquely identified by its name and the method *find_instance* returns a pointer to the appropriate instance (as a root pointer) given the instance name.

3 The Root Class

The key problem with incremental linking is to provide a mechanism for adding a new class to a running program. This reduces to two sub problems. First, it must be possible to define an interface to an hitherto unknown class and second, it must be possible to create instances of these newly linked in classes and invoke methods on them from the original program.

A user can try to invoke any operation on such an object and the object examines the request to decide if it is a valid operation. If so, the operation is simply invoked. Else, a run-time error occurs. That means, static type-checking (i.e., checking the validity of an operation invokation statically) is impossible in that kind of situation. This contradicts with the view that every object in a C++ program must be of a specific type so that the set of valid operations is known at compile time.

One way to solve this problem is to extend the virtual function mechanism of C++. The virtual function dispatch of C++ allows objects to be manipulated in a type-safe manner without actually knowing their exact type. If the new class is derived from an already known class with virtual function members, the interface of the base class can be used to type-check calls to member functions of the derived class. In effect, the virtual function mechanism provides the link between the existing code and the new code.

This forms the motivation for the definition of the *root* class. It sits at the top of the hierarchy of all the user-defined classes.

```
dbclass root
{
  static MetaClass* class_ptr;
public:
  dbchar *instance_name;
  root(dbchar* cname,dbchar* iname);
  ~root();
  Type* get_type(dbchar* aname); //return the type of an attribute
  virtual void copy(root*){}; // make a copy of this object
  virtual Val get_value(dbchar* ){}; // get the value of an attribute
  virtual int put_value(dbchar*,Val ){}; // store a value into an attr
  virtual void invoke_method(char*,ValList){}; //invoke a method
}
```

Each root instance contains a pointer to the appropriate *MetaClass* object in order to get access to the type information at run-time. The get_value() and put_value() methods give access to individual members of an object by making use of the *map* information provided by the *MetaClass* instance.

4 Implementation Strategy

The user defines new classes by means of a class editor. The class definitions are parsed and a *MetaClass* instance is created on the database for each class definition. The parser is generated by LEX/YACC tools and it can captures the grammar for C++ class attributes.

For each class definition, E code is generated in the form of .h and .e files. These files are then compiled into .o files by the E compiler. The entire program is maintained in a sort of 'program data base' of compiled fragments. The fragments are annotated with information such as dependency graphs between classes, cross references etc. Whenever a change is made to the class hierarchy (such as adding a new class, changing the inheritance relation between classes, or changing a class), this information is used to decide what fragments need to be recompiled.

4.1 Dynamic Linker/Loader

We use the dynamic linking and loading facilities provided by the SUN OS [8]. The SUN OS provide a simple programmatic interface to the services of the dynamic link-editor. Operations are provided to add a new shared object to an program's address space, obtain the address bindings of symbols defined by such objects, and to remove such objects when their use is no longer required. These functions are:

- dlopen() provides access to a shared object returning a descriptor (handle) that can be used for later references to the object in calls to dlsym() and dlclose().
- dlsym() returns the address binding of the symbol described in the null-terminated character string symbol as it occurs in the shared object identified by handle.
- dlclose() deletes a reference to the shared object refer- enced by handle. If the reference count drops to 0, then if the object referenced by handle defines a function _fini, that function will be called, the object removed from the address space, and handle destroyed.

The set of all user-defined class definitions is compiled into a shared library. Whenever a change is made to the class hierarchy, the existing shared library in the address space of the running program is deleted from the address space by calling the function dlclose(), and the newly recreated shared library is linked in by the function dlopen(). Whenver the shared library is linked into a running program by dlopen(), map for each class is reloaded by means of the dlsym() function.

5 Conclusions

We have described the design and implementation of an environment which facilitates persistent C++ programming by providing various utilities such as run-time type information etc. We are yet to work out the strategy for incrementally adding new virtual functions to a class. We also have to work out the strategy for handling data layout changes in existing instances. This means that we have to implement a *cutover* member function that converts existing objects of a certain class into a new version whenver that class is altered.

References

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

[2] Dmitry Lenkov, Mitchey Mehta, Shankar Unni, *Type Identification in C++*, USENIX C++ Conference Proceedings, 1991.

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

[4] Joel E. Richardson, Michael J. Carey, Daniel T. Schuh, *The Design of E Programming Language*, ACM Transactions of Programming Languages and Systems, Vol.15, No.3, 1993.

[5] Roy H. Campbell, Nayeem Islam, David Raila, Peter Madany, *Designing and Implementing Choices: An Object-oriented system in C++*, Communications of the ACM, Vol.36, No.9, 1993.

[6] Mark Grossman, *Object I/O and runtime type information via automatic code generation in C++*, Journal of Object-oriented Programming, July-August, 1993.

[7] James O.Coplien, *Advanced C++*, Addison-Wesley, 1992.

[8] *Programmers Overview: Utilities and Libraries* Sun Microsystems Inc, 1990.

[9] Bjarne Stroustrup, *The C++ Programming Language*, Addison Wesley, 1985.

[10] M.J.Carey, D.J. DeWitt, J.E.Richardson, E.J.Shekita, *Storage Management for objects in EXODUS*, Object-oriented Concepts, Databases, and Applications, W. Kim and F.Lochovsky, eds, Addison-Wesley, 1989.

[11] D.Sriram et al, *COSMOS Design Document*, IESL Technical Report, MIT, 1991.