A RUNTIME TYPE INFORMATION SYSTEM FOR C++

Frank Buschmann, Konrad Kiefer, Frances Paulisch, Michael Stal

SIEMENS AG, ZFE BT SE 31
Otto - Hahn - Ring 6, 8000 München 83, GERMANY

Tel: +49/89/636-49380, Fax: +49/89/636/40898
Email: mop@ztivax.siemens.com

Position

In the discussion about meta-level architectures our position is as follows: There are two major alternatives how a programming language can support such facilities:

- full language support
- language support for the basic concepts plus a mechanism to extend this minimum

Both alternatives have pros and cons:

- full language support:

  pros: direct support of all features by the language and its compilers

  efficiency

  standardization

  cons: danger of overloading the language (not all features are needed for all kinds of applications)

  lack of features (there will always be applications which need more support than that provided by the language)

- language support for the basic concepts plus a mechanism to extend this minimum

  The pros of the previous alternative are the cons of this one, the cons of the previous alternative its pros

Our point of view in this discussion is that we prefer the second alternative because of its higher flexibility. We also believe that meta-level architectures are useful for programming languages based on compile time information like C++. Independent from the chosen approach, we consider the following points as important for providing meta-level architectures:

- is a feature useful?
- is it necessary?
- is it technically sound?

Only if all three questions can be answered with “yes”, should a meta-level facility be integrated in a programming language or a standard library.

The following article is based upon a paper which was presented at the TOOLS ‘92 conference in Dortmund.

Abstract

In the area of C++ programming a sophisticated mechanism for runtime type identification and access to runtime type information is required to support many applications and libraries. This paper describes an adequate runtime type information system for the C++ programming language: the **Meta-Information-Protocol** (MIP). An integration strategy is presented to demonstrate how type information can be generated and how the MIP can be included and used within C++ applications. The MIP itself is implemented in a library class hierarchy with a base class called **Meta**. It includes member functions to access type information about classes, data and function members of classes and their types. The MIP is based on the object-oriented approach. It supports typesafe downcast for the C++ language. In addition development of libraries and toolkits are supported as well as integration of different libraries.

1 Introduction

Various attempts have been made in C++ to implement a method of type identification, access to type information [STROUSTRUP92] [LENKOV91] [INTERRANTE90] [GORLEN87] [WEINAND88] and a mechanism to access additional application specific type information. Such methods and mechanisms are needed for the following reasons:

- Support for typesafe downcast
- Support for integration of libraries
- Support for libraries and toolkits

In this paper we examine the problem of type identification and access to type information in C++. We introduce a runtime type information system for the C++ programming language which consists of extensions of classes and a set of library classes. An integration concept is presented to demonstrate how the type information is generated and used within C++ applications.

Runtime type information and the MIP are currently discussed within the ANSI committee. The ANSI X3J16 subgroup is responsible for standardizing the C++ programming language.

2 The Concept of MIP

At present, type identification and access to type information are not integrated in the C++ programming language. Since many C++ applications need access to type information, a Siemens in-house project is launched to provide a type information system with a full set of member functions required by the operating divisions of the company. It is called the **Meta-Information-Protocol** (MIP). This runtime type information system is realized with the help of extensions of user defined classes which allow type identification and access to type information. In addition, a library class **Meta** is implemented which includes the member functions of MIP.

The functionality of MIP is divided into three layers which are developed stepwise. The three layers consist of member functions with access to:

Layer 1: Type information about classes. The layer provides basic functionality to support typesafe downcast.

Layer 2: Type information about data and function members of classes and their types. This layer provides additional functionality to access type information. It is required for applications which need comprehensive type information.

Layer 3: Information about relative addresses of data members. In addition functions which support object I/O are implemented. The goal of this layer is to support the implementation of persistency.

The MIP approach allows for direct access to type information about classes, rather than access to type information about other types (e.g. int). Type information about a class is bound to the class itself. Type information about data and function members of a class is bound to the type information about the class and can be accessed through this type information. Type information about types of members of classes is bound to the type information about these members.

2.1 The Meta Class Hierarchy

The runtime type information system supports access to type information about classes, members of classes and their types. To implement the system we identified the following different kinds of type information:

- Type information about *classes*
- Type information about *data members* of classes
- Type information about *function members* of classes
- Type information about *union* declarations
- Type information about *enum* declarations

In addition, type information about C++ types is necessary to cover the whole type concept of the C++ programming language, e.g. type information about arrays.

The functionality of the runtime type information system is implemented in a class hierarchy (see fig.1). Its base class is called **Meta**. For each kind of type information a separate class is derived from Meta. These classes implement that part of the functionality of MIP that is applicable to the kind of type information they represent.

2.2 The Methods of MIP

The Meta class hierarchy comprises the following methods:

name(...) returns the name of a class, data member, function member, or type, with scope hierarchy if necessary.

size(...) returns the size of a class, data member or type in byte.

baseClassIter(...) iterates through the direct base classes of the given class. **classListIter(...)** iterates through all classes.

findClass(...) looks for type information about a class specified by its name.

nbMembers(...) returns the number of data and function members of a class.

dataMemberIter(...) iterates through the data members of the given class.

fctMemberIter(...) iterates through the function members of the given class.

nbParams(...) returns the number of parameters of a function member.

paramIter(...) iterates through the parameters of a function.

friendIter(...) iterates through all friends of a class.

nbIndices(...) returns the number of indices of an array.

nbElems(...) returns the number of elements of an enum type.

enumIter(...) iterates through all elements of an enum type.

typeCompIter(...) returns the address of the type information about type components of member, typedef and function type declarations. It iterates through all components of a type declaration.

typeDecl(...) returns the type declaration of class, struct, union, typedef, enum and data member declarations as well as the return type of function members and global functions. This method allows type identification and type comparison.

kindOf(...) identifies the kind of the current type information, e.g. “classKind” for type information about classes.

3 Open Issues

Some issues in the MIP approach for type identification require further discussion

- Support of exception handling

  Exception handling uses implicitly the type identification mechanism. It is necessary that the mechanism has access to information about inheritance hierarchies. If the C++ exception handling mechanism should use the MIP, there will be a need for a further concept which integrates the MIP into the exception handling mechanism. Since the C++ exception handling mechanism allows to handle any type the MIP must be extended to directly support type inquiry for all types.

- Extension of the generated type information with application specific information

  Sometimes it is necessary to extend the type information generated by the compiler or a tool with application specific information. For example such information is required for a flexible object-oriented inter-process-communication. A suitable mechanism to handle application specific type information must be developed for MIP.

- Initialization of type information about global types

  The implementation of MIP layer 2 requires that type information about other types than classes must be stored. Type information about classes is initialized statically through invoking the constructors of the class Meta directly. A problem is the initialization of type information about user defined global types which are used within classes, e.g. type information about types declared using the **typedef** construct of C++. The reason for this problem is that the MIP approach to access type information is to extend classes, but types can not be extended.

A possible solution to this problem is to introduce a member function for all classes called **initMeta()**. This member function initializes the type information for all types used within the class. Since the method is directly invoked at the beginning of an application, the correct initialization of global types is guaranteed. The disadvantage, however, is that the initMeta functions may do some redundant work. Consider the following example: If two classes are using the same global type, code for initializing of this type is implemented in the initMeta member functions of both classes. The initialization mechanism of class Meta has to avoid that type information for this type is declared twice.

- How to access type information

  Every class must have access to information about its type and it must be possible to access this type information from outside the class. One solution would be to extend classes with additional members:

  private:

  static Meta classInf;

  public:

  virtual const Meta getMeta() const;*

  static const Meta getInfo() const;*

  The data member **classInf** contains the type information about the class. This variable must be initialized properly before starting the main routine of C++ applications. One drawback of this approach is that for “external C” classes and classes without virtual functions, the addition of the virtual function getMeta() will cause their storage layout to change.

  An alternate solution is to integrate type identification into the language. Through the type identifier of a type it is possible to access runtime type information which is supported by a library class. The ANSI discussion about runtime type identification and information follows this direction (see chapter 6).

4 Integration of the Concept

Without any support of MIP through the C++ programming language and its compilers there is a need for a tool called **MIP-generator** to support the runtime type information system. Tasks of the MIP-generator are:

- Extension of classes

  According to the MIP concept described in section 2 additional members must be implemented in each class.

- Generation of type information

  The MIP-generator must add new code to the source files of applications which initializes the static data member classInf.

- Inclusion of the class Meta

  The class Meta must be included into every source file of an application to have access to the member functions of MIP.

The MIP-generator has to be included into the translation process and must be invoked right after the C++ preprocessor. The MIP-generator includes a lexical analysis created by the UNIX™ standard program LEX and a parser for the C++ language created by the Berkeley YACC (using the C++ grammar published by James A. Roskind).

5 Related Work

Many class libraries and toolkits provide their own concept of runtime type identification and access to runtime type information. Relevant for our work are the Dossier approach [INTERRANTE90] and the Hewlett Packard approach [LENKOV91] which are summarized in the following subsections. Other interesting ideas on runtime type information can be found in [STROUSTRUP91][STROUTSTRUP92].

6 Runtime Type Information as part of C++

At present it is discussed within the ANSI X3J16 subgroup to include runtime type information in the C++ language [STROUSTRUP92]. One of the alternatives presented there for discussion is as follows.

- Type identification will be integrated in the language standard. A class typeid allows the user to identify the dynamic type of pointers. A template class Typeid, which is derived from typeid, allows the user to get the type identifier of a static type.
- The typeid objects contain a pointer to an object of a library class Type_info.
- The library class Type_info provides methods to access type information about a type.
- The standard type information may be extended by deriving classes with additional type information from the class Type_info.

If runtime type information is supported by the C++ language, some of the open issues enumerated in section 3 are solved by the compiler and linker.

7 Current Status and Future Work

The MIP layer 1 and those parts of MIP layer 2 that deal with type information about classes and data members are already implemented. The implementation of the full functionality of MIP is currently in progress. Work on a generic solution to persistence is also in progress. The goal here is to demonstrate the usefulness of runtime type information based on MIP with a concrete example. A MIP-generator that modifies and extends the source files of applications to make available type identification and access to type information at runtime is planned. A mechanism for extending the generated type information with application specific information will follow after the completion of the whole MIP functionality.

8 References

[GORLEN87] Keith E. Gorlen, An Object-Oriented Class Library for C++ Programs, Proceedings of the USENIX C++ Workshop, 1987

[INTERRANTE90] John A. Interrante, Mark A. Linton, Runtime Access to Type Information in C++, USENIX C++ Conference, 1990

[LENKOV91] Dmitry Lenkov, Michey Mehta, Shankar Unni, Type Identification in C++, USENIX C++ Conference, 1991

[ROSKIND91] James A. Roskind, A YACCable C++ grammar, 1991

[STROUSTRUP92] Bjarne Stroustrup, Dmitry Lenkov, Runtime type identification for C++, C++ Report, March/April 1992

[STROUSTRUP91] Bjarne Stroustrup, The C++ Programming Language, 2nd edition, 1991

[WEINAND88] Andre Weinand, Erich Gamma, Rudolf Marty, ET++ - An Object-Oriented Application Framework in C++, ACM OOPSLA 1988, Conference Proceedings, 1988

