** DRAFT **
A Runtime Type Checking and Query Mechanism for C++
** DRAFT **

Mary Fontana
LaMott Oren

Texas Instruments Incorporated
Computer Science Center
Dallas, TX

Martin Neath

Texas Instruments Incorporated
Information Technology Group
Austin, TX

ABSTRACT

The Texas Instruments C++ Object-Oriented Library (COOL) is a collection of classes, templates and macros for use by C++ programmers writing complex applications. Symbolic computing in COOL is one component of this library that substantially improves the development capabilities available to the programmer by providing symbol and package manipulation and runtime type checking and type query. This paper will focus on the implementation of runtime type checking and query.

1. Introduction

The Texas Instruments C++ Object-Oriented Library (COOL) is a collection of classes, templates and macros for use by C++ programmers writing complex applications. An important feature of this library is the symbolic computing capability which provides symbol and package manipulation and runtime type checking and query. This paper will describe the runtime type checking capability. For details on symbols and packages in COOL, see the paper, *Symbols and Packages in C++* [4]. For an overview of the COOL class library, see the paper, *COOL - A C++ Object-Oriented Library* [1]. For complete details, see the reference document, *COOL User's Guide* [5].

The ability to query an object at runtime to determine its type is commonly used in many symbolic computing and complex knowledge-intensive operations management applications. Such capability is found in highly dynamic languages such as SmallTalk [7] and Common LISP [8], but the C++ language does not support this. COOL facilitates a runtime type checking and query capability for C++ classes with the **Generic** class, the **Symbol** class, the **SYM** macro and the **class** macro. The **Generic** class provides the **type_of** and **is_type_of** member functions for runtime type checking and query. **type_of** returns the class type of the object and **is_type_of** determines whether the class type of the object inherits from a specified class type. The **Symbol** class is used to represent the type information of a class object. The **SYM** macro stores symbol names in the "sym_package.p" file which is used to automatically create symbol objects at program startup. Finally, the COOL **class** macro generates the class hierarchy structure and the **type_list** virtual member function for classes singly or multiply derived at some point from the **Generic** class.

2. Symbols

COOL supports symbolic constants and runtime symbolic objects. The **Symbol** class implements the notion of a symbol object that has a name, value and property list (see Figure 1). **Symbol** objects can be created and interned into a package, which is merely a mechanism for establishing separate name spaces for groups of symbols [4]. **Symbol** objects are also utilized by the **Generic** runtime type checking and query member functions as a way of representing the type information of a class object. A **Symbol** object is automatically created at program starup for each class derived from **Generic** with name set to the class name and value and property list set to **NULL**.

The **SYM** and **class** macros are implemented with the COOL macro facility [2] which gives special directives to the COOL preprocessor. The **class** macro (discussed below) uses **SYM** to create Symbol objects for **Generic**-derived classes. **SYM(***name***)** is replaced in the source code with the address of the Symbol object for *name*. At the same time, "*name*" is added to the file "sym_package.p". The total number of symbols and the names of all the symbols in an application are saved in "sym_package.p". The file "symbols.C" initializes a global array of symbols objects with the names stored in "sym_package.p" and is compiled and linked with the rest of the application files.

The application uses the COOL C++ Control program (CCC), instead of the normal CC procedure, to control the compilation process. This program provides all of the capabilities of the original CC program with additional support for the COOL preprocessor. Figure 2 shows the flow of the COOL compilation process using CCC and "symbols.C".

Example

```
Foo.C                                  Foo.i
  class Generic;                         class Generic;
  Generic* obj;                          Generic* obj;
  if (obj->is_type_of(SYM(foo)) {        if (obj->is_type_of(&SYM_symbols[0])) {
  ...                                    ...
  }                                      }

Bar.C                                  Bar.i
  class Generic;                         class Generic;
  Generic* obj;                          Generic* obj;
  if (obj->is_type_of(SYM(bar)           if (obj->is_type_of(&SYM_symbols[1])
      || obj->is_type_of(SYM(baz))           || obj->is_type_of(&SYM_symbols[2]))
  ...                                    ...
  }                                      }


// from sym_package.p

  #define SYM_count  2
  MACRO SYM_DEFINITIONS(define_name) {
  define_name(0,"foo");
  define_name(1,"bar");
  define_name(2,"baz");
}

#define MAKE_SYM(index, symbol)   SYM_symbols[index].name = symbol;
```

```
// from Symbols.C

 #include "sym_package.p"
 Symbol SYM_symbols[2];    // SYM_count
 void SYM_package_initializer() {
   ...
   // SYM_DEFINITIONS(MAKE_SYM);
   SYM_symbols[0].name = "foo";
   SYM_symbols[1].name = "bar";
   SYM_symbols[2].name = "baz";
 }

static Package SYM_package_s(SYM_package_initializer);
```

3. Generic class

The **Generic** class is inherited by all COOL classes. For a complete definition of this class see Figure 3. It provides runtime type checking capabilities for any class that uses it as a base class. This information is accessed with the following member functions:

Symbol* Generic::type_of ();
Returns the symbol type associated with the object.

Boolean Generic::is_type_of (Symbol* type);
Returns **TRUE** if the object is the specified type or inherits from that type somewhere in its class hierarchy, otherwise returns **FALSE**.

int Generic::select_type_of (Symbol type_vector);**
If the object's symbol type is found in the specified array of pointers to symbols, returns the integer index of the matching symbol type, otherwise returns -1. The **TYPE_CASE** macro (discussed later) uses this member function with the **switch** statement to select an appropriate execution path based upon the symbol type of the object.

virtual Symbol Generic::type_list();**
Returns a **NULL**-terminated array whose first element is a pointer to the symbol for the class being defined. The rest of the array elements are pointers to the **type_list**s of the base classes.

The **type_list** member function represents the class hierarchy of the **Generic**-derived class. Note that the **type_list** member function is virtual. Each class derived from **Generic** has a **type_list** member function which is generated automatically by the **class** macro. The **type_of**, **is_type_of** and **select_type_of** member functions use the virtual **type_list** member function to access the class hierarchy of the class object.

Example

The following example shows runtime type checking operations on the variable **n** which is of type **Long**. **Long** is derived from **Generic**. In this example, **SYM(Long)** is a pointer to a **Symbol** object, **&SYM_entry[13]**.

```
extern Generic* n = Long(123);
static Symbol* sym_list[4] = { SYM(long), SYM(Long), SYM(Integer), NULL };

n->type_of ();                       // returns  &SYM_entry[13]
n->is_type_of (SYM(Long);            // returns  TRUE
n->is_type_of (SYM(long));           // returns  FALSE
n->select_type_of (sym_list);        // returns  1
```

4. Class macro

The keyword **class** is a COOL macro which takes a C++ class description and generates code to support the runtime type checking and query mechanism. The **class** macro adds the following lines into the class definition.

```
protected:
    virtual Symbol** type_list ();
```

The definition of the **type_list** member function and the array of symbol pointers representing the class hierarchy of the class are added after the class definition.

Example

The following example shows how the class **Long** which is derived from class **Generic** is expanded by the COOL **class** macro. The additional code generated is highlighted in bold. Note that **Long_types** is a NULL-terminated array of symbol pointers whose first element is a pointer to the **Long** symbol and second element is a pointer to **Generic_types** which contains the **Generic** symbol pointer.

```
class Long : public Generic {
  private:
    long num;
  public:
    Long (long value) { num = value; };                // Make a long
    long operator long () { return this->num; };       // Get the value out
};
```

expands to:

```
class Long : Generic {
  private:
    long num;
  protected:
    virtual Symbol** type_list ();
  public:
    Long (long value) { num = value; };                // Make a long
    long operator long () { return this->num; };       // Get the value out
};

extern Symbol* Generic_types[];                        // initialized to {SYM (Generic), NULL }

Symbol* Long_types[] = { SYM (Long), (Symbol*) Generic_types, NULL };

Symbol** Long::type_list () { return Long_types; }
```

4.1. Classmac macro

The actual code which is expanded in a class definition and after a class definition is controlled by the **classmac** macro. Each **classmac** macro defines how the **class** macro should expand the class definition. The **class** macro does not actually generate the code itself. This is defined in user-modifiable header files that specify **classmac** macros. Thus no changes to the COOL preprocessor are required.

In the following example, the **type_list** member function for the Generic-derived class is generated by defining two **classmac** macros. The **DECLARE_GENERIC** macro expands the **type_list** member function declaration inside the class definition and the **IMPLEMENT_GENERIC** macro expands after the class the definition for **type_list**. The **GENERATE** macro is used to generate block code for multiple base classes of a class definition.

Example

```
classmac (DECLARE_GENERIC, inside)
classmac (IMPLEMENT_GENERIC)

MACRO DECLARE_Generic(class, REST: bases) {
  WHEN_GENERIC_CLASS(class, bases) {
     protected:
       virtual Symbol** type_list() CONST;
  }
}

MACRO IMPLEMENT_Generic(class, REST: bases, BODY: slots) {
  WHEN_GENERIC_CLASS(class, bases) {
     GENERATE(base, bases)
       { extern Symbol* base##_types[]; }
     Symbol* class##_ types[] =
       { SYM(class), GENERATE(base, bases) { (Symbol*) base##_ types, } NULL };
     Symbol** class##::type_list() CONST { return class##_types; }
  }
}
```

The **classmac** macro provides two hooks as a point of customization by user defined macros. A combination of data members and member functions of a class definition are passed as arguments to the macros that can be changed or customized by the application programmer. There may be more than one **classmac** macro hook specified by the programmer. COOL has several and other user-defined macros are simply chained together in a calling sequence ordered according to the order of definition. For example, in COOL there is the **map_over_slots** virtual member function which calls a specified function on all of the data members in the class. (The describe member function would use **map_over_slots** to display the name, value, and type of each data member). The definition of **map_over_slots** is automatically generated for each **Generic**-derived class by defining **classmac** with the data member hook. The **send_if_handles** member function which supports the run-time query of objects for member function availability is generated by utilizing the member function hook in **classmac**.

5. TYPE_CASE

The **TYPE_CASE** macro is analogous to the C++ **switch** statment. It gathers all possible type cases and allows the user to symbolically dispatch on the type of object represented by the **case** statements. It uses the **Generic::select_type_of** member function as shown in the following macro expansion of **TYPECASE**.

Example

```
Generic* g;
TYPE_CASE (g) {
  case Vector:              // If the object is a vector
    ....                    // Do something for Vector
    break;
  case List:                // If the object is a list
    ....                    // Do something for List
    break;
  default:                  // Else do the rest
    ....
}
```

expands to:

```
Generic* g;
static Symbol* switch_symbols_g[3] = {SYM(Vector), SYM(List), NULL};
switch (g->select_type_of(switch_symbols_g)) {
  case 0:                   // If the object is a vector
    ....                    // Do something for Vector
    break;
  case 1:                   // If the object is a list
    ....                    // Do something for List
    break;
  default:                  // Else do the rest
    ....
}
```

6. Comparison with Dossiers

The COOL runtime type checking and query capability is very similar to the Dossier interface proposed in the paper by Interrante and Linton, *Runtime Access to Type Information in C++* [6]. Both the **Dossier** class and the COOL **Symbol** class represent the type information of a class object. Both the **mkdossier** tool and the COOL preprocessor (running the **class** macro) generate a C++ source file ("__dossier.h" and "sym_package.p") to contain the type information. The Dossier interface generates type information for all classes, but COOL generates runtime type member functions only on classes derived singly or multiply at some point from the **Generic** class. This is done automatically by the COOL preprocessor and the code actually generated can be customization by the user. This next table attempts to show more comparisons between the two runtime type information mechamisms.

```
   DOSSIER                             COOL
------------                         -----------
extern Dossier** classes()           extern Symbol SYM_symbols[]
Boolean Dossier::isA(Dossier*)       Boolean Generic::is_type_of(Symbol*)
Dossier* class::GetClassId()         Symbol* Generic::type_of()
DossierItr Dossier::parents()        Symbol** class::type_list()
DossierItr Dossier::children()       none

Class Foo foo;                       class Foo foo;   // :public Generic
class Bar bar;                       class Bar bar;   // :public Generic
extern Dossier* FOO;                 SYM(Foo);        // &SYM_symbol[0]
extern Dossier* BAR;                 SYM(Bar);        // &SYM_symbol[1]

foo.GetClassId()  returns FOO        foo.type_of() returns SYM(Foo)
foo.GetClassId()->isA(BAR)           foo.is_type_of(SYM(Bar))
```

7. Conclusions

Texas Instruments has been using the symbolic computing capability in COOL for the last two year. Applications have utilized COOL symbols, packages, and runtime type checking and type query of **Generic**-derived class objects. Most COOL classes contain **Generic** base class. A significant benefit of this common base class is the ability to declare heterogenous container classes parameterized over the **Generic*** type. In addition, the COOL exception handling facility [3] takes advantage of the runtime type checking of exception objects. An automated method for generating the symbols objects for type information is accomplished with the COOL macro facility [2].

COOL is currently running on a Sun SPARCstation 1 running SunOS 4.x, a PS/2 model 70 running SCO XENIX® 2.3, a PS/2 model 70 running OS/2 1.2, and a MIPS running RISC/os 4.0. The SPARC and MIPS ports utilize the AT&T C++ translator (cfront) version 2.x and the XENIX and OS/2 ports utilize the Glockenspiel C++ translator with the Microsoft C compiler.

8. References

[1] Mary Fontana, Martin Neath and Lamott Oren, *COOL - A C++ Object-Oriented Library,* Information Technology Group, Austin, TX, Internal Original Issue January 1990.

[2] Mary Fontana, Martin Neath and Lamott Oren, *A Portable Implementation of Parameterized Templates Using A Sophisticated C++ Macro Facility,* Information Technology Group, Austin, TX, Internal Original Issue January 1990.

[3] Mary Fontana, Martin Neath and Lamott Oren, *A Portable Exception Handling Mechanism for C++,* Information Technology Group, Austin, TX, Internal Original Issue January 1990.

[4] Mary Fontana, Martin Neath and Lamott Oren, *Symbols and Packages in C++,* Information Technology Group, Austin, TX, Internal Original Issue January 1990.

[5] Texas Instruments Incorporated, *COOL User's Guide,* Information Technology Group, Austin, TX, Internal Original Issue January 1990.

SunOS and SPARCstation 1 are trademarks of Sun Microsystems, Inc.
PS/2 is a trademark of International Business Machines Corporation.
XENIX is a registered trademark of Microsoft Corporation.
OS/2 is a trademark of International Business Machines Corporation.

[6] John Interrante and Mark Linton, *Runtime Access to Type Information in C++*, Proceedings of the USENIX C++ Conference, San Francisco, CA, April 9-11, 1990.

[7] Adele Goldberg and David Robson, *SmallTalk-80: The Language and its Implementation*, 1983.

[8] Guy L. Steele Jr, *Common LISP: The Language*, Second Edition, 1990.

Figure 1: The Symbol class

```
class Symbol : public Generic {                   // Define the Symbol class
  friend class Package;                      // Package class needs access

protected:
  const char* pname;                         // Symbol name
  Generic* val;                              // Symbol value
  Association<Symbol*, Generic*>* proplist;     // Property list
  inline Symbol (const char* name);          // Use Package::intern()

public:
  Symbol ();                                 // Used for constant symbol[]
  ~Symbol();                                 // Destructor
  Boolean get (const Symbol*, Symbol_GenericP&); // Lookup value
  void put (const Symbol*, Generic*);               // set plist value
  Boolean remove (const Symbol*);            // Remove value from plist
  inline const char* name () CONST;                 // Accessor for pname
  inline Generic* value ();                  // Accessor for value
  inline Generic* set (Generic*);            // Set new value
  inline Association<Symbol*, Generic*>* plist(); // Accessor for properties
  friend ostream& operator<< (ostream&, const Symbol*);// Print symbol
  friend ostream& operator<< (ostream&, const Symbol&);// Print symbol
};
```

Figure 2: CCC Foo.C and Bar.C

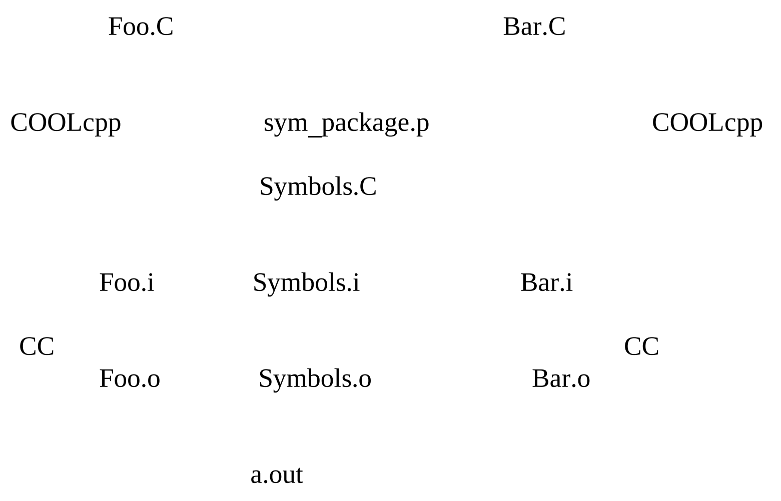

Figure 3: The Generic class

```
class Generic {
protected:
  Generic(){};                    // Abstract class's have Protected constructors
  virtual Symbol** type_list();
public:
  virtual ~Generic();                       // Virtual destructor
  inline Symbol* type_of();          // Return the type symbol
  Boolean is_type_of(Symbol* type);       // Type checking predicate
  int select_type_of(const Symbol** type_vector);
  virtual Boolean map_over_slots(Slot_Mapper procedure, void* rock = NULL);
  virtual void describe(ostream&);   // Display all slots in some "raw" format
  friend ostream& operator<< (ostream&, const Generic&); // Overload output operator
  friend ostream& operator<< (ostream&, const Generic*); // Overload output operator
  void print(ostream&);                     // terse print
};

Symbol* Generic_types[] = {SYM (Generic), NULL};

Symbol** Generic::type_list() CONST {
  return Generic_types;
}

inline Symbol* Generic::type_of() {
  return *(this->type_list());
}

extern Boolean compare_types( Symbol** typelist, Symbol* type);
inline Boolean Generic::is_type_of(Symbol* type) CONST {
  return compare_types(this->type_list(), type);
}

extern int compare_multiple_types(Symbol** type_list, const Symbol** test_types);
inline int Generic::select_type_of(const Symbol** test_types) CONST {
  return compare_multiple_types(this->type_list(), test_types);
}
```
