** DRAFT **
Symbols and Packages in C++
** DRAFT **

Mary Fontana
LaMott Oren
Dane Meyer

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 and sample usage of symbols and packages.

** Note that currently this paper describes the implementation of symbols and packages in a very technical reference format. We are in the process of modifying it to place more emphasis on why a programmer would want to use this capability in object oriented C++ programs. This will include example situations, usage, and comparison with other alternatives, in addition to the current reference material. **

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 symbol and package mechanism. Symbols and packages make extensive use of the COOL macro facility. This macro facility, implemented as an extension to the ANSI C preprocessor, provides the ability to define powerful extensions to the C++ language in an unobtrusive way. For a full discussion of this macro facility, see the paper *A Portable Implementation of Parameterized Templates Using A Sophisticated C++ Macro Facility*, [5]. For details on the runtime type checking capability in COOL, see the paper, *A Runtime Type Checking and Query Mechanism for 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* [6].

The ability to manipulate symbols and packages is is found in Common LISP [7] but is unavailable in the C++ language. COOL supports efficient symbolic computing by providing symbolic constants and run-time symbol objects. Dynamic user-defined packages support symbol storage, query and modification at run-time and are

available via an easy-to-use programming interface.

2. Symbol and Package

The **Symbol** class implements the notion of a symbol object that has a name, value and property list. Symbols are interned into a package, which is merely a mechanism for establishing separate name spaces for groups of symbols. The **Package** class is implemented as a hash table of symbol objects and includes public member functions for adding, retrieving, updating and removing symbols. It also provides "Emacs" style completion, "InterLisp" style spelling correction, and "apropos" (which finds a symbol whose name contains a specified substring) on the symbol names. Because symbols are unique within their own package, they can be used as dynamic enumeration types as well as run-time variables defined at run-time.

In addition to dynamic, run-time symbols, the COOL **DEFPACKAGE** mechanism makes it possible to define constant pointers to symbols at compile-time. These compile-time symbols are interned in their package at application startup time. For example, the **SYM** package and **ERR_MSG** package in COOL are initialized in this manner. This mechanism allows for highly efficient symbol and package manipulation and is used extensively by COOL to implement run-time type checking and query capabilities.

3. DEFPACKAGE

DEFPACKAGE allows declaration of a package which contains only constant symbols with associated default values and properties. This is useful when the application needs to set up a table of symbols where all information is known at compile time. This saves the run-time overhead associated with the **Package** class. An application declares a new type of constant package with the following statement:

DEFPACKAGE *name <file> options*

where *name* is the name of the package to create, *file* is the name of the file where symbol definitions in the package are kept, and *options* are optional control parameters separated by commas. The list of options available is shown below.

count = *identifier*
: Defines *identifier* as a symbolic name constant. If specified, the package file will include "#**define** *identifier num*", where *num* is the total number of symbols defined in the package at compile-time. The default is to not define this constant.

use_first = *num*
: When *num* is non-zero, the first value defined for a symbol is the one used. Redefinition attempts are ignored. This option is used by **ONCE_ONLY**. The default is 0.

start = *num*
: Use *num* as the starting (first) index of symbols in the package. The default is 0.

increment = *num*
: Increment each symbol index in package by *num*. The default is 1.

template = *num*
: Inclusive-or each symbol index in package with *num*. The default is 0.

max = *num*
: When *num* is non-zero, generate an error when the number of constant symbols in the package exceeds *num*. The default is 0.

nospace = *num*
: When *num* is non-zero, remove all whitespace from symbol names. The default is 0.

case = upper | **lower** | **cap** | **sensitive**
: **upper** and **lower** converts all symbol name alphabetical characters to uppercase and lowercase, respectively. **cap** capitalizes the first letter of each symbol name, and convert remaining letters to lowercase. **sensitive** preserves the case of the symbol name and is the default.

DEFPACKAGE builds an internal package table entry based on its parameters. The package entry gets further

updated with symbols specified throughout the program with **DEFPACKAGE_SYMBOL**.

4. DEFPACKAGE_SYMBOL

DEFPACKAGE_SYMBOL adds, updates and retrieves the values and properties of constant symbols. It is implemented as a COOL **defmacro** [2] and has the following syntax:

DEFPACKAGE_SYMBOL (*package***,** *symbol***,** [*type*]**,** [*value*]**,** [*property*]**,** [*expander*]**)**

where *package* is the name of the package that the symbol belongs to, *symbol* is the name of the symbol being added or accessed, *type* is the optional data type of the value, *value* is the optional value of the symbol or property, *property* is the optional name of the property, and *expander* is the name of an optional user-defined macro which is to be invoked in place of the default expansion of **DEFPACKAGE_SYMBOL**. The *expander* will be called by **DEFPACKAGE_SYMBOL** with the following arguments:

*expander (index***,** *symbol***,** *type***,** *value***)**

where *index* is the index number of symbol in the package, and *expander*, *symbol*, *type* and *value* are the arguments supplied to **DEFPACKAGE_SYMBOL**.

DEFPACKAGE_SYMBOL causes a macro definition unique to the specified package to be written to the package definition file. The <package>_DEFINITIONS macro expands to macro calls which define the symbols, set the symbol values, and set the symbol properties. Each **DEFPACKAGE_SYMBOL** would generate a define_macro, value_macro or property_macro in the package definition file.

```
MACRO package##_DEFINITIONS (define_macro value_macro property_macro) {
    define_macro (index, name)
    value_macro (index, type, value)
    property_macro (index, property, type, value)
    ...
}
```

For example, the **SYM** package is defined with **DEFPACKAGE**. It's symbols will be stored in the file, **sym_package.p**. The named constant, **SYM_count**, will contain the total number of symbols in the **SYM** package. The case of each symbol is preserved. A symbol's value may be updated. Zero is the starting index for the package. No increment or inclusive-or is done on the symbol's index. Finally, there is no limit on the maximum number of symbols allowed in this package.

```
DEFPACKAGE SYM <sym_package.p> count=SYM_count, case=sensitive, use_first=0,
                    start=0, increment=1, template=0, max=0
```

Shown below, **SYM** is a macro defined to add a symbol name to the **SYM** package. **expand_sym** is the expander macro for SYM which replaces occurrences of **SYM** with the symbol table entry of the symbol. **SYM_symbols** is an array of pointers to Symbol objects and is initialized with the symbol names found in the file **sym_package.p**.

```
Symbol* SYM_symbols [SYM_count];

#define expand_sym (index, symbol, type, value)        (SYM_symbols[index])

MACRO SYM (symbol) {
  DEFPACKAGE_SYMBOL(SYM, #symbol,,,, expand_sym)
}
```

COOL supports run-time type query by using the **SYM** package to store all class name which inherit from the **Generic** class. The **class defmacro** in COOL uses the **SYM** macro to add a derived Generic class to the **SYM** package. An application uses **SYM** to access a symbol in the SYM package.

```
Generic* obj;
obj->is_type_of (SYM(Error));
```

expands to:

```
Generic* obj;
obj->is_type_of (SYM_symbols[1]);
```

Each **DEFPACKAGE_SYMBOL** adds or updates a symbol table entry and optionally sets either the symbol's value or property. After the symbol table is built, the package is written into the package definition file specified in **DEFPACKAGE**. How this package file of symbols is loaded into the application depends on the type of package being created.

Several macros in COOL simplify the process of creating, accessing and initializing a package at run-time. **ENUMERATION_PACKAGE**, **SYMBOL_PACKAGE**, **TEXT_PACKAGE** and **ONCE_ONLY** create a specific type of package, define macros which are used to add symbols to the package and update their values and properties, and define the run-time initialization of the package.

Examples

(1) Assuming the SYM package was created with:

```
DEFPACKAGE SYM <SYM> name=sym_package.p, count=SYM_count, case=sensitive,
                                start=0, increment=1, template=0, max=0
```

and the macro SYM is used to add a symbol to the SYM package:

```
#define expand_SYM (index, symbol, type, value)       (SYM_symbols[index])

MACRO SYM (symbol) {
      DEFPACKAGE_SYMBOL(name, #symbol,,,, expand_SYM)
}
```

In an application program:

```
SYM(Exception);  SYM(Warning);  SYM(Error);  ...  SYM(Random);
```

would expand to:

```
SYM_symbols[0];  SYM_symbols[1];  SYM_symbols[2]; ...  SYM_symbols[35];
```

and the following would be included (by the COOL preprocessor) in the sym_package.p file for the SYM package:

```
#define SYM_count 36

MACRO SYM_DEFINITIONS (define_macro, value_macro, property_macro) {
      define_macro  (0, "Exception")
      define_macro  (1, "Warning")
      define_macro  (2, "Error")
      ...
      define_macro  (35, "Random")
}
```

In COOL, the SYM package would be initialized with the symbols found in the sym_package.p file by using the implement_symbol_package macro in the application program. implement_symbol_package uses SYM_DEFINITIONS to create an instance of Package for SYM.

```
implement_symbol_package(SYM, "sym_package.p")
```

(2) Assuming the Once_Only package was created with:

```
DEFPACKAGE Once_Only <Once_Only> name=once_only.p, count=, case=sensitive,
                                 start=0, increment=1, template=0, max=0
```

the following shows what would be included in the once_only.p file for the Once_Only package:

```
MACRO Once_Only_DEFINITIONS(define_macro, value_macro, property_macro) {
    define_macro  (0, String_Support)
    value_macro   (0, char*, "vtbl.C")
    ...
    define_macro  (22, List_Support)
    value_macro   (22, char*, "Base_List.C")
    ...
}
```

(3) The following shows how the ENUMERATION_PACKAGE macro is implemented using DEFPACKAGE and DEFPACKAGE_SYMBOL:

```
MACRO enumeration_package (name, file, REST: options) {
    DEFPACKAGE name file options
    #define expand_##name (index, symbol, type, value)   index
    #define name (symbol)   DEFPACKAGE_SYMBOL (name, symbol,,,, expand_##name)
}
```

5. ENUMERATION_PACKAGE

ENUMERATION_PACKAGE is a simple interface to the **DEFPACKAGE** facility for creating and accessing a package of constant, compile-time symbols. Enumeration symbols have no value. When a symbol is referenced, its index is returned. Symbols in an enumeration package can be used anywhere a standard **enum** can be used. Enumeration symbols are easier to add then **enums** since they are collected throughout the application source base and maintained in a separate file.

ENUMERATION_PACKAGE is a COOL **MACRO** [2] and has the following syntax:

ENUMERATION_PACKAGE (*name***,** *file***, REST:** *options***)**

where *name*, *file* and *options* are parameters that are passed on to **DEFPACKAGE** and **DEFPACKAGE_SYMBOL**. The macro *name*(*symbol*) is automatically defined to add *symbol* to the *name* package if it does not exist and return the index of *symbol* in *name* package.

Example

In the example below, the **FONT** package is created with **ENUMERATION_PACKAGE**. **fg_13**, **fg_25** and **vg_13** are symbols in this package. **FONT(fg_13)**, **FONT(fg_25)**, and **FONT(vg_13)** expand to 1, 15, 22, respectively and are the 1st, 15th and 22nd element in the **FONT** package, respectively.

```
ENUMERATION_PACKAGE (FONT, "font_package.h");

if (f == FONT(fg_13) || f == FONT(fr_25)) do_fixed_width_font();
else if (f == FONT(vg_13)) do_variable_width_font();
```

expands to:

```
if (f == 1 || f == 15) do_fixed_width_font();
else if (f == 22) do_variable_width_font();
```

6. TEXT_PACKAGE

TEXT_PACKAGE is a simple interface to the **DEFPACKAGE** facility for creating and accessing a package containing symbols whose default value is the same as its symbol name. For example, all error messages in an application could be implemented as text symbols and the symbol definition file for this text package would then contain a summary of all the messages. The message text can be substituted with another version (perhaps in another language) at run-time. The COOL error message facility [3] is implemented in this manner. **TEXT_PACKAGE** is implemented with **MACRO** and has the following syntax:

TEXT_PACKAGE (*name***,** *file***, REST:** *options***)**

where *name*, *file* and *options* are parameters that are passed on to **DEFPACKAGE** and **DEFPACKAGE_SYMBOL**. The macro *name*(*symbol*) is defined to add and retrieve *symbol* from the *name* package. If *symbol* has not already been added to the package, it is added and the **char*** value is returned. If *symbol* is already present in the package, the existing **char*** value is returned.

Example

In the following example, after creating the **ERR_MSG** package with **TEXT_PACKAGE**, the ERR_MSG macro is used to add and retrieve error message symbols. ERR_MSG expands to the current translation of the error message in the **ERR_MSG_entries** table. This table is initialized at run-time with the compile-time error message strings found in the **err_package.p** file.

```
TEXT_PACKAGE (ERR_MSG, "err_package.p");
ERR_MSG("String::resize(): Invalid size %d");
```

expands to:

```
ERR_MSG_entries[37].value;
```

and is the **37**th symbol and contains the default value of "**String::resize(): Invalid size %d**".

7. SYMBOL_PACKAGE

SYMBOL_PACKAGE is a simple interface to the **DEFPACKAGE** facility for creating and accessing a package which contains symbols whose values can be assigned at run-time. Entries in a symbol package are pointers to **Symbol** objects. Symbols known at compile time are interned in their package at application startup time and additional symbols are added at run-time. Symbols may have a value and a set of properties. If not specified, the value and properties are non-existent, that is, no space other than storage for a **NULL** pointer is allocated for them in a table. **SYMBOL_PACKAGE** is a COOL **MACRO** [2] and has the following syntax:

SYMBOL_PACKAGE (*name***,** *file***, REST:** *options***)**

where *name*, *file* and *options* are parameters that are passed on to **DEFPACKAGE** and **DEFPACKAGE_SYMBOL**. **SYMBOL_PACKAGE** creates a package and defines three macros for use in adding, updating, and retrieving symbols in this package. The macro *name(symbol)* adds or retrieves *symbol* from the *name* package. The macro **DEF**_*name* adds or updates the value of a symbol in the *name* package. The macro **DEF**_*name*_**PROPERTY** adds or updates a property of a symbol in the *name* package.

Example

The **SYM** package could be created using **SYMBOL_PACKAGE**:

```
SYMBOL_PACKAGE (SYM, "sym_package.p")
```

and the following macros for accessing the **SYM** package would be defined:

SYM (*name***)**
: Adds the specified symbol *name* to the **SYM** package, if not already defined. Returns a pointer to a **Symbol** object. For example, **SYM(foo)** adds the symbol name, **foo** to the **SYM** package and

expands to **&SYM_symbols[37]**.

DEF_SYM (*name***,** *type***,** *value***)**
Sets the specified symbol *name* in the **SYM** package with the specified *value* and *type*. For example, **DEF_SYM (foo, String, String(**"**Greetings!**"**));** sets the value of symbol **foo** to the string "**Greetings!**".

DEF_SYM_PROPERTY (*name***,** *property***,** *type***,** *value***)**
Sets the named *property* of the symbol *name* in the **SYM** package with the specified *type* and *value*. For example, **DEF_SYM_PROPERTY (foo, value-type, Symbol, SYM (String));** sets the **value-type** property of symbol **foo** to the symbol **String.**

8. ONCE_ONLY

ONCE_ONLY provides a simple interface to the **DEFPACKAGE** facility to allow an application to control the processing of a section of code. This is useful in an application where a function or table is referenced in several source files and needs to be defined the first time it is referenced. **ONCE_ONLY** is a COOL **MACRO** [2] and has the following syntax:

ONCE_ONLY (*symbol***)** { *body* }

where *symbol* is the name of a symbol and *body* is the body of code to insert in the source only once. *symbol* is used to ensure that *body* is compiled only once. *symbol* is added to the **ONCE_ONLY** package and its initial value is the file name where *symbol* was first defined. When *symbol* is encountered and its current value is the same as the current file, then the body of code is compiled. If *symbol* is found in a different file, then the code is not inserted into the source stream.

Example

The **AUTO_DECLARE** macro defined below insures that **IMPLEMENT** is called only once in an application using a parameterized type. It would be used instead of **DECLARE** and **IMPLEMENT** and would be included in every source file that references the parameterized type.

```
MACRO AUTO_DECLARE(class, type) {
   DECLARE class<type>;
   ONCE_ONLY (Implement_##class_##type) {
      IMPLEMENT class<type>;
   }
}

AUTO_DECLARE (List, int);
List<int> list1;
```

9. Status of COOL

Texas Instruments has been using the symbolic computing capability in COOL for the last year. Applications have utilized COOL symbols, packages, and run-time type checking and type query of **Generic**-derived class objects. In addition, the COOL exception handling facility [3] takes advantage of the run-time type checking of exception objects.

COOL is currently up and running on a Sun SPARCstation 1 (TM) running SunOS (TM) 4.x, a TI System 1500 running TI System V, a PS/2 model 70 running SCO XENIX® 2.3, a PS/2 (TM) model 70 running OS/2 1.1,

---

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

and a MIPS running RISC/os 4.0. The SPARC and MIPS ports utilize the AT&T C++ translator (cfront) version 2.0 and the XENIX and OS/2 ports utilize the Glockenspiel translator with the Microsoft C compiler.

10. 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, *A Runtime Type Checking and Query Mechanism for C++,* Information Technology Group, Austin, TX, Internal Original Issue January 1990.

[5] 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.

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

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