A Portable Implementation of Parameterized Templates Using A Sophisticated C++ Macro Facility

Mary Fontana
LaMott Oren

Texas Instruments Incorporated
Computer Science Center
Dallas, Texas, 75265

Martin Neath

Texas Instruments Incorporated
Information Technology Group
Austin, Texas, 78759

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. Parameterized types, symbolic computing and exception handling are significant features of COOL which improve the development capabilities available to the programmer. These features are implemented in COOL with a sophisticated C++ macro facility. This paper describes the COOL macro facility, discusses how support for parameterized templates is built upon it, and provides details of two programmer interfaces (both implemented) for easy use of parameterized templates in application programs.

1. Introduction

The Texas Instruments C++ Object-Oriented Library (COOL) is a system-independent software platform consisting of classes, templates and macros for use by C++ programmers writing complex applications. It is designed to raise the level of abstraction for the programmer in order to facilitate concentration on the problem domain, not on implementing basic data structures, macros, and classes. Parameterized templates, symbolic computing, and exception handling are significant features of COOL that substantially improve the development capabilities available. We wished to provide these facilities in a compiler- and machine-independent manner across several hardware platforms. We examined the macro language found in standard C-preprocessors and determined that it was insufficient for implementing these features. As a result, we developed the COOL macro facility to allow the programmer to define powerful extensions to the C++ language in a seamless and unobtrusive manner. This macro facility is implemented as an extension to a C preprocessor [1]. The modifications made to the preprocessor are both portable and compiler independent. This paper describes this enhanced macro facility, discusses how parameterized templates is built upon it, and provides details of two programmer interfaces (both implemented) for easy use of parameterized templates. For an overview of COOL see the paper, *COOL - A C++ Object-Oriented Library* [2]. For complete details, see the reference document, *COOL User's Manual* [3].

2. The COOL Preprocessor and the defmacro Keyword

Many C++ language implementations separate the preprocessor and compiler functions into two separate programs. Others combine the preprocessor and compiler into one step. Since we needed a portable utility to

The authors may be reached via electronic mail at fontana@csc.ti.com, oren@csc.ti.com, and neath@itg.ti.com.

massage C++ source code that works under both scenarios and executes after include files and standard preprocessor directives have been expanded, but before the C++ compiler itself begins parsing, we decided to modify a C-preprocessor to support the COOL C++ language extensions. Thus, the COOL preprocessor is derived from and based upon a public domain C-preprocessor (the DECUS C preprocessor) made available by the DEC User's group and supplied on the X11R3 source tape from MIT. It has been modified to comply with the draft ANSI C specification with the exception that trigraph sequences are not supported.

The draft-proposed ANSI C standard indicates that extensions and changes to the language or features implemented in a preprocessor or compiler should be made by using the #**pragma** keyword. The COOL preprocessor recognizes a #**pragma defmacro** declaration and is the single hook through which features such as the class macro, parameterized templates, and polymorphic enhancements have been implemented. The **defmacro** keyword provides a way to define and execute arbitrary filter programs or macro expanders on C++ code fragments. The syntax of the **defmacro** declaration is:

#**pragma defmacro** *name* "*program*" *options*
#**pragma defmacro** *name* <*program*> *options*

where *name* is the name of the macro, *program* is either the name of an executable file or the name of an internal preprocessor function which implements the macro expansion, and *options* are any combination of the following optional parameters:

recursive	- the macro may be recursively expanded.
expanding	- the input to the macro is expanded.
delimiter= *x*	- the default delimiter (semi-colon) is replaced with *x*.

Unknown *options* are passed as arguments to the macro expander named *program*. This provides the necessary handle through which COOL functions and language extensions can be identified. For example, the **MACRO** and **template** keywords are defined in a top-level header file with:

```
#pragma defmacro MACRO "macro" delimiter=} recursive
#pragma defmacro template "template" delimiter=}
```

The implementation of the macro expander program may be either external or internal to the preprocessor. Fundamental COOL macro-expanders are implemented internal to the COOL preprocessor for the sole reason of providing a more efficient execution profile to reduce compile time for the application programmer. When the preprocessor encounters a **defmacro** declaration in the source code, it searches first for an executable file named *program* on the same search path as that used for include files. If a match is not found, it then searches for *program* in an internal preprocessor function table. If a match is still not found, an error is reported indicating that the macro expander could not be found. This search order allows an internal preprocessor definition to be overridden by an external one.

When a **defmacro** name is successfully recognized, the name and all following characters upto and including the delimiter character (including all matching and nested levels of {} **[] ()** <> "" **'' ''** and comments found along the way) are piped into the standard input of the macro expander program. The expander program performs whatever function(s) is appropriate and the resulting massaged character stream is piped back onto the standard output of the macro expander. This output stream is scanned as new input by the preprocessor for any further processing that might be necessary. The original text in the source file is replaced with the preprocessor output before being sent onto the C++ compiler. The expansion replacing a **defmacro** name in the source code is C++ 2.0 syntax acceptable to any conforming C++ translator or compiler [4].

3. The MACRO Keyword

The COOL **MACRO** keyword provides a powerful and flexible macro capability used to implement and simplify many of the features and functions contained in the library. A **defmacro** named **MACRO** (all uppercase) provides an enhanced #**define** macro that supports multi-line, arbitrary length, nested macros and cpp-directives with positional, optional, optional keyword, required keyword, rest, and body arguments. **MACRO** has the following syntax:

MACRO *name* **(** *parmlist* **)** { *body* }
parmlist := [**KEY:** | **REST:** | **BODY:**] *identifier* [**=** *identifier*] [**,** *parmlist*]

where *name* is the name of the macro, *parmlist* is a list of parameters separated by commas, and *body* is the code which replaces the **MACRO** reference. The *parmlist* specification allows positional, keyword, rest, and body parameters to be identified by the programmer. The positional and keyword parameters may be required or optional. Optional parameters are supported by use of an equal sign followed by an *identifier* that specifies the default value. All the optional positional parameters must follow all of the required positional ones.

When **KEY:** is specified in the *parmlist*, all parameters which follow are keyword parameters. Keyword parameters are position-independent parameters. A keyword parameter is provided a value in an argument list by supplying the keyword name followed by an equal sign and the argument value. **REST:** in the *parmlist* indicates that the remaining parameters are referenced by one named *identifier*. An optional equal sign followed by an *identifier* sets the *identifier* after the equal sign to the number of arguments remaining. Finally, **BODY:** in the *parmlist* indicates that the parameter which follows is expanded to include all the source code within the braces after the **MACRO** call. This is useful for passing a source code fragment onto other nested **MACRO**s. Examples of these three types of arguments are given below.

3.1. MACRO Examples

The following examples show some of the power and flexibility of **MACRO**. This first example uses both positional parameters and keyword parameters.

```
MACRO set_val (size, value=NULL, KEY: low=0, high) {
    init (size, value, low, (high-low))
}
```

set_val has three parameters: *size* is a required positional parameter, *value* is an optional positional parameter that if not specified in a particular call has a default value of **NULL**, *low* is an optional keyword parameter with a default value of **0**, and *high* is a required keyword parameter. In this example, the expansion calls the function **init** with four arguments. The following shows several expansions of **set_val**.

```
set_val (0, high=20)          ---->    init (0, NULL, 0, (20-0))
set_val (0, low=5, high=15)   ---->    init (0, NULL, 5, (15-5))
set_val (1, 2, high=25)       ---->    init (1, 2, 0, (25-0))
```

The next example uses the **REST:** parameter. Note that there are two **MACRO**s defined: **build_table** calls **build_table_internal** to do most of the work.

```
MACRO build_table (name, REST: rest) {
    char* name[] = { build_table_internal(rest) NULL}
}

MACRO build_table_internal (first, REST: rest=count) {
    #first,
    #if count
    build_table_internal (rest)
    #endif
}
```

build_table has two parameters: *name* is the name of the table of **char***'s and *rest* refers to all the remaining arguments. **build_table** calls **build_table_internal** passing its *rest* argument. Note that this call is embedded within the initialization braces of the table and is followed by a **NULL**. In **build_table_internal**, *first* is set to the first argument of the *rest* argument list in the invoking macro call, and the remaining *count* arguments are left in *rest*. **build_table_internal** uses the ANSI # character on *first* to double quote the value. A conditional clause tests *count* to see if there are remaining arguments. If *count* is non-zero, the macro is called recursively with the remaining arguments. When there are no more arguments, **build_table** regains control and appends the **NULL** character and closing brace to the result of **build_table_internal**.

A sample use of **build_table** is shown below to illustrate the construction of a **NULL**-terminated table containing character strings. The first line shows the macro call and the second shows the resulting expansion.

```
build_table (table, 1,2,3,4,5,6,7);
```

expands to:

```
char* table[] = {"1", "2", "3", "4", "5", "6", "7", NULL};
```

This last example uses the **BODY:** parameter and also takes advantage of the current position feature found in the COOL container classes [2]. This is used to implement a general purpose loop macro similar to that found in Common LISP [5].

```
MACRO LOOP (type, identifier, object, BODY: body) {
    { type identifier;
      for ( object.reset(); object.next(); ) {
        identifier = object.value();
        body
      }
    }
}
```

LOOP has four parameters: *type* is the type of each element in a container class (such as, **int**), *identifier* is the name of a variable to be declared of the given type, *object* is the name of a container class instance, and *body* is the body of code to apply on each element in the container object. A specific example for the parameterized **List<int>** class is shown below.

```
extern List<int> list1;
LOOP (int, var1, list1) { cout << var1; }
```

expands to:

```
extern List<int> list1;
{ int var1;
  for ( list1.reset(); list1.next(); )  {
    var1 = list1.value();
    cout << var1;
  }
}
```

In this example, **list1** is an instance of **List<int>** which is a container class representing a list of integers. **LOOP** takes this list object and iterates through the elements, assigning each to a temporary integer variable **var1** and printing its value. The net result will print all elements in the list.

4. COOL Parameterized Templates

One of the main uses of the COOL macro facility is the implementation of **template**, **DECLARE** and **IMPLEMENT** for supporting parameterized templates. The syntax of the **template** grammar is that as specified by Stroustrup in his paper, *Parameterized Types for C++* [6]. COOL fully implements this functionality such that there will be minimal source code conversion necessary when this feature is finally implemented in the C++ language. COOL provides templates for a number of parameterized classes (such as, **Range** and **Iterator**) and container classes (such as, **Vector**, **List**, **Binary_Tree** and **Hash_Table**) which are described in *COOL User's Guide* [3].

4.1. The template Keyword

The **template** keyword provides a mechanism for defining parameterized classes. A parameterized class is a type-independent class. A typical use is a container class where the type of the contained object is specified at compile-time. For example, vectors can be declared to hold a specific type of element, such as, a vector of integers or a vector of doubles, from a single parameterized class, **Vector**<*type*>.

A **template** is divided into the declarative part and the implementation part of a class. The declarative part may occur many times in an application and is analogous to including a header file for a class which contains the class definition and its inline member functions. The implementation part is analogous to the file that contains the source code implementing the member and friend functions of the class. COOL provides four variations of **template** for these two parts:

template <**class** *type* [, *parms*]> **class** *name*<*type*> { *class_description* };
Defines the class template for the declarative part of the *name* class.

template <**class** *type* [, *parms*]> **inline** *result name*<*type*>::*function* { ... };
Defines an inline member function for the declarative part of the *name* class.

template <**class** *type* [, *parms*]> *result name*<*type*>::*function* { ... };
Defines a member function for the implementation part of the *name* class.

template <**class** *type* [, *parms*]> *name* { *anything* };
Defines anything else associated with a template for the *name* class.

This last form is used to define such things as **typedef**s or friend functions of a parameterized class. When this form is found before the class template, the contents are expanded before the class definition. When this form is found after the class template, the contents are expanded as part of the class implementation. Note that this form is not part of the parameterized type syntax described by Stroustrup [6]. Rather, it is something we found lacking in the original proposal and found very useful in several COOL container classes for defining predicate types for the class under C++ 2.0. Another use of this form is to provide automatic declarations of nested parameterized classes, that is, to declare a parameterized class for a class template which is itself derived from another parameterized class template.

Each variation of **template** allows additional optional parameters with the following syntax:

parms ::= *type name* [**=** *value*] [**,** *parms*]

where *type* is the type of the parameter, (such as, **class** or **int**); *name* is the name of the parameter that is substituted when **template** is expanded; and *value* is the default value of parameter *name*.

The following is an example of **template** for the class, **Vector**<*type*>**.**

```
template <class Type> Vector {                  // predicate functions
   typedef int (*Vector_## Type##_ Predicate) (const Type&, const Type&);
   typedef Boolean (*Vector_##Type##_Compare) (const Type&, const Type&);
};

template <class Type> class Vector<Type> {      // Parameterized Vector class
  private:
    Type* v;                                    // Vector of pointer to Type
    int num_elements;                           // Element count
    int size;                                   // Size of vector object
  public:
    Vector<Type> ();                            // Empty constructor
    Vector<Type> (int);                         // Constructor with size
    Vector<Type> (const Vector<Type>&);         // Constructor with reference
    ~Vector<Type> ();                           // Destructor
    inline Type& operator[](int n);             // Operator[] overload for Type

    ...                                         // ... other member functions ...
};
```

```
template <class Type>                          // Overload operator []
inline Type& Vector<Type>::operator[] (int n) {
    return this->v[n];
}

template <class Type>                          // Constructor with size
Vector<Type>::Vector<Type> (int n) {
    this->v = new Type[n];
    this->size = n;
    this->num_elements = 0;
}

 ...                                           // ... other member functions ...
```

4.2. An Initial Programmer Interface: DECLARE and IMPLEMENT

As stated earlier, a **template** for a parameterized class is divided into a declarative part and an implementation part. In our first attempt at implementing parameterized template support, the programmer creates instances of a parameterized class using **DECLARE** to expand the declarative part and **IMPLEMENT** to expand the implementation part. **DECLARE** defines the parameterized class for a specific type and **IMPLEMENT** generates the member functions supporting this type-specific class. **DECLARE** must be used in every file that includes or makes use of the parameterized class. **IMPLEMENT** must be used only once in the application for each type over which the class is parameterized; otherwise the linker will generate errors about multiple versions of the same member functions. For example, to create a vector of doubles, the following would be used:

```
#include <Vector.h>
DECLARE Vector<double>;
IMPLEMENT Vector<double>;
Vector<double> vs(30);
```

DECLARE expands to code which defines a vector class of doubles and its associated inline member functions. **IMPLEMENT** causes a class definition with its associated member functions to be generated and expanded in the file. When compiled, this causes the class **Vector_double** to be declared and defined. One drawback of the use of **IMPLEMENT**, however, is the fact that the *entire* class with all its member functions is generated and linked into the program image, even if the programmer only requires the use of two or three member functions. This problem can be avoided by the use of the COOL C++ Control Program (**CCC**) discussed below. Continuing with the example above, the **template** for the **Vector**<*type*> class for doubles would expand to the following code:

```
                                               // predicate functions
typedef int (*Vector_double_Predicate) (const double&, const double&);
typedef Boolean (*Vector_double_Compare) (const double&, const double&);

class Vector_double {                          // Parameterized Vector class
private:
    double* v;                                 // Vector of pointer to double
    int num_elements;                          // Element count
    int size;                                  // Size of vector object
public:
    Vector_double ();                          // Empty constructor
    Vector_double (int);                       // Constructor with size
    Vector_double (const Vector_double&);      // Constructor with reference
    ~Vector_double ();                         // Destructor
    inline double& operator[](int n);          // Operator[] overload for double
```

```
    ...                                          // ... other member functions ...
};

                                                 // Overload operator []
inline double& Vector_double::operator[] (int n) {
    return this->v[n];
}

                                                 // Constructor with size
Vector_double::Vector_double (int n) {
    this->v = new double[n];
    this->size = n;
    this->num_elements = 0;
}

 ...                                             // ... other member functions ...

Vector_double vs(30);
```

Declarations of nested parameterized types and the use of non-type arguments in a template definition are also supported. For example, it is possible to declare a vector of vectors of ints with **Vector**<**Vector**<**int**>>. In addition, a class template derived from another class template is supported, that is, a type parameter in one template class can be used to declare another class template of that type. For example, the COOL **Association**<*T1*,*T2*> class is a parameterized container class that takes two type arguments, *T1* and *T2*. The header file for this class has the following templates.

```
template <class T1, T2> Association {
    DECLARE Pair<T1, T2>;                        // Declare Pair object type
    DECLARE Vector<Pair<T1, T2>>;                // Declare Vector of Pairs
}

 ...                                             // ...
// Association<T1,T2> class definition here


template <class T1, T2> Association {
    IMPLEMENT Pair<T1, T2>;
    IMPLEMENT Vector<Pair<T1, T2>>;
}
```

By using **template** in this manner, **DECLARE** for the **Association**<*T1*,*T2*> class invokes **DECLARE** on the correct types for the **Pair**<*T1*,*T2*> and **Vector**<*Type*> classes. Likewise, **IMPLEMENT** for the **Association**<*T1*,*T2*> class invokes **IMPLEMENT** for the **Pair**<*T1*,*T2*> and **Vector**<*Type*> classes.

Non-type arguments as template parameters are used to provide guidelines to be used when a template is expanded. For example, the **N_Tree**<*Node*,*Type*,*nchild*> class in COOL takes as arguments a node type (either static or dynamic), a type specifying the value-type each node will hold, and an argument that specifies the number of initial subtrees (or children) each node is to have. The node argument is itself the name of a parameterized class and a nested parameterized template definition is automatically generated based upon the supplied type and number arguments. As such, a single template can be used to generate several different classes with different behaviors and features.

4.3. A Revised Programmer Interface: COOL C++ Control Program (CCC)

The **DECLARE** and **IMPLEMENT** macros discussed above were the first programmer interface implemented for parameterized template support. We soon discovered, however, that this macro expansion mechanism had two serious problems. First, the type over which a class was parameterized would have to support all operators used in the template, even if not applicable or needed. For example, the COOL **List**<*Type*> class has several member functions that use **operator**<. However, if what the programmer needs is a list of window objects and does not ever use **List**<*Type*> member functions that require **operator**<, compile-time errors from the offending functions that got macro-expanded are nevertheless generated. Second, with the simplistic linkers available on many operating systems today, an application gets all of these member functions linked into the executable image. Typically, an application uses only a small percentage of the member functions of a parameterized class. The remaining unused member functions are useless overhead, increasing program size and memory requirements.

A revised programmer interface for parameterized templates was implemented to resolve these problems and centers around a new program to be used as the main interface between the user and the preprocessor/compiler in a make file. This program, the COOL C++ Control program (**CCC**), augments the standard **CC** script. For most operations, user options and command line arguments are passed straight through to the underlying CC program. However, when the **-X** option is specified, the **CCC** program goes to work in the following manner. As Stroustrup[6] suggested, **-X**"**Foo**<**Bar**>" is used on the command line to indicate that the programmer wants to parameterize class **Foo**<*Type*> over some type **Bar**. Additional options for include file search path and a user-defined library archive are required as described below. **CCC** finds the header file(s) implementing class Foo and type Bar, then proceeds to define that type for the compiler. It then *fractures* the implementation of this new type along template boundaries, placing each non-inline member function in a separate source file, compiling it, and putting the resulting object file in a user-specified library archive. If a particular operator is not defined for the type over which the class is parameterized (as with the example of **operator**< above), a compile time error for that one file is generated. However, the remaining member functions, one in each fractured template, are still compiled and added to the user library.

For each parameterized class in an application, **CCC** fractures the parameterized class definition along template boundaries, causing each **template** specifying a member function of the parameterized class to be compiled into a separate object file. These separate object files are then added to an application-specific object library. Since each member function is in its own object module in the library, only those member functions actually used in the application are linked into the final executable image. To use **CCC**, the programmer specifies a library name, one or more header files containing templates, and specific parameterized classes as command line arguments to **CCC**. Other arguments are passed on unchanged to the C++ compiler and system linker. A single invocation of **CCC** can either process a parameterized class type or compile a C++ source file, but not both. For example,

```
CCC -lapp -c List.h String.h -X "List<String>"
```

expands the template for a list of strings. The resulting object files from the fractured parameterized **List**<*Type*> class are stored in the library, **libapp.a**. The **-c** option is passed to the compiler to indicate that it should not continue with the link phase. The library archive **libapp.a** is added to the list of libraries specified in the make file to be searched during the link step.

The net result is a library archive containing object files, each implementing one member function for the parameterized class and type. This process solves the two problems identified above with the use of the macros **DECLARE** and **IMPLEMENT**. First, operators not defined for a type cause compile-time errors on that one file. Once a parameterized class has been implemented and provided in a library, compile errors will only occur when a type is selected that does not have all operators implemented. The user of the class will see these, and if the member function in question is required, s/he can add that necessary operator to the type class. Second, only member functions actually used in an application are linked into the final executable image.

4.4. Future Improvements to CCC

CCC essentially provides a more sophisticated version of the **IMPLEMENT** macro discussed above. However, the programmer is still required to place the **DECLARE** macro in the appropriate files. One option under consideration to resolve this problem is the use of a command line switch similar to the +e0/+e1 switchs on the AT&T cfront translator. Under this scenario, the equivalent of the +e0 option would be used to declare the type for a parameterized class and generate the inline member functions (as the **DECLARE** macro does) but not to generate the remaining member functions. The programmer would use the equivalent of the +e1 option on one source file to cause the remaining non-inline member functions to be generated and placed in a library archive.

A second problem with **CCC** concerns the specification of nested parameterized classes. A programmer should be able to use **-X"Vector<List<int>>"** on the command line to specify creation of nested parameterized classes. Currently, **CCC** does not handle this case appropriately. A more sophisticated command line parser should be able to recognize and implement nested types before trying to expand the outer most parameterized class.

5. Conclusion

The COOL macro facility provides a mechanism to implement significant language features and extensions for C++ that are unavailable with current language implementations. The macro facility is implemented in an enhanced preprocessor that is both efficient and portable, thus allowing for delivery of enhanced language features on many platforms. This macro extension is at the heart of the parameterized templates functionality. **CCC** is used in place of the normal procedure for controlling the compilation process. It provides all of the functionality of the original **CC** program with additional support for the COOL preprocessor and parameterized type expansion. Finally, the preprocessor provides an ideal mechanism for quickly prototyping and testing additional language functions and syntax without requiring access to or modification of a compiler.

6. Status of COOL

Texas Instruments has been using the enhanced macro facility and the implementation of parameterized templates internally on several projects for the last year. Many classes and programs have been successfully designed and implemented, taking full advantage of the power of parameterized templates and the enhanced macro facility. In particular, we have found that the use of a class library supplying many basic parameterized container classes significantly increases the productivity of the programmer, enabling applications to be prototyped in a shorter time period than might otherwise be possible. COOL is currently up and running on a Sun SPARCstation 1 (TM) running SunOS (TM) 4.x, a PS/2 (TM) 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 2.1 and the XENIX and OS/2 ports utilize the Glockenspiel translator 2.0a with the Microsoft C 6.0 compiler.

The COOL preprocessor source code is available in compressed tar(1) format in the file /pub/cpp.tar.Z via anonymous FTP from CSC.TI.COM (128.247.159.141). Permission is granted to any individual or institution to use, copy, modify, and distribute this software, provided that all copyright statements and permission notices are maintained, intact, in all copies and supporting documentation. Texas Instruments Incorporated provides this software "as is" without express or implied warranty.

7. References

[1] Brian Kernighan and Dennis Richie, *The C Programming Language*, Second Edition, Printice-Hill, Englewood Cliffs, NJ, 1988.

[2] Mary Fontana, Martin Neath and Lamott Oren, *COOL - A C++ Object-Oriented Library*, Information Technology Group, Austin, TX, TI Internal Document, 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.

[3] Texas Instruments Incorporated, *C++ Object-Oriented Library User's Manual*, Information Technology Group, Austin, TX, TI Internal Document, Original Issue January 1990.

[4] AT&T Incorporated, *C++ Language System Release 2.0,* AT&T Product Reference Manual Select Code 307-146, 1989.

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

[6] Bjarne Stroustrup, *Parameterized Types for C++*, Proceedings of the USENIX C++ Conference, Denver, CO, October 17-21, 1988, pp. 1-18.