COOL — C++ Object-Oriented Library

Mary Fontana
LaMott Oren

Texas Instruments Incorporated
Computer Science Center
Dallas, TX

Martin Neath

Texas Instruments Incorporated
Information Technology Group
Austin, TX

1. Introduction

The C++ Object-Oriented Library (COOL) is a collection of classes, templates, and macros for use by C++ programmers writing complex applications. It raises the level of abstraction for the programmer to concentrate on the problem domain, not on implementing base data structures, macros, and classes. In addition, COOL also provides a system independent software platform on top of which applications are built, since COOL encapsulates such system specific functionality as date/time and exception handling. This paper discusses the following COOL features:

- Preprocessor and macros
- Parameterized templates
- Symbols and Packages
- Polymorphic management
- Exception handling
- Coding style and conventions
- Class hierarchy and overview

COOL is the first piece of what is expected to be an ever changing and growing C++ class library. As such, some constraints will be necessary in order to achieve compatible and seamless integration of new or modified features. This paper outlines the major technologies and conventions that can be used and followed to allow this to happen. This paper should be used as an aid in understanding the COOL library, its organization, structure, and layout. It assumes the reader to have a working knowledge of C++[1]. For more detailed information and examples on each topic, the reader is referred to the appropriate section(s) of the COOL User's manual[2].

2. Preprocessor and Macros

The COOL macro facility is an extension to the standard ANSI C macro preprocessing functions available with the #define statement. The COOL preprocessor is a modified tokenizing ANSI C preprocessor that allows a programmer to define powerful extensions to the C++ language in an unobtrusive manner. This enhanced preprocessor is portable and compiler independent and can execute arbitrary filter programs or macro expanders on C++ code fragments. Macros such as those that support parameterized templates are implementations of theoretical design papers published by Bjarne Stroustrup[3]. Others provide significant language features and enhanced power for the programmer heretofore unavailable with conventional C++ implementations. It is important to note, however, that once a macro is expanded, the resulting code is conventional C++ 2.0 syntax acceptable to any conforming C++ translator or compiler[4].

The COOL preprocessor is supplied as part of the library and is the point at which all language and computing enhancements available in COOL are implemented. The proposed draft ANSI C standard indicates that extensions and changes to the language and/or features implemented in a preprocessor/compiler should be made by using the #pragma statement. The COOL preprocessor follows this recommendation and uses this as the means by which all macro extensions are made. The #pragma defmacro statement is the single hook through which features such as the class macro, parameterized templates, and polymorphic enhancements have been implemented.

The COOL preprocessor is derived from and based upon the DECUS ANSI C preprocessor made available by the DEC User's group in the public domain and supplied on the X11R3 source tape from MIT. It complies with the draft ANSI C specification with the exception that trigraph sequences are not implemented. In addition to support for COOL macro processing discussed above, the preprocessor has several new command line options to support C++ comments and includes file debugging aids.

The #pragma defmacro statement is implemented in the COOL C/C++ preprocessor and is the single hook through which features such as the class macro, parameterized templates, and polymorphic enhancements have been implemented. The defmacro facility provides a way to execute arbitrary filter programs on C++ code fragments passing through the preprocessor. When a defmacro style macro name is found, the name and everything until the delimiter (including all matching {} [] () <> #" '' and comments found along the way) is piped onto the standard input stream of the indicated program or filter procedure. The procedure's standard output is scanned by the preprocessor for further processing. The expansion replaces the macro call and is passed onto the compiler for parsing.

The implementation of a defmacro can be either external to the preprocessor (as in the case of files and programs) or internal to the preprocessor. For example, the template, declare and implement macros that implement parameterized types is internal to the preprocessor to provide a more efficient implementation. The defmacro facility first searches for a file or program in the same search path as that used for include files. If a match is not found, an internal preprocessor table is searched. If a match is still not found, the error message, "Error: Cannot open macro file [xxx]", where xxx is the name as it appears in the source code, is sent to the standard error stream. The fundamental COOL macros are defined with defmacro in the header file <COOL/misc.h>, which must be included in any COOL C++ source file.

Porting COOL to a new platform or operating system starts with the preprocessor. The preprocessor contains support for the defmacro statement and also implements several important macros internally for efficiency and performance considerations. In addition, a powerful macro language, simplifying many library functions is available via the **MACRO** keyword. **MACRO** implements an enhanced #define syntax that supports multiple line, arbitrary length, nested macros and preprocessor directives with positional, optional, optional keyword, required keyword, rest, and body arguments. Many of the COOL features would be very difficult, if not impossible, to implement without this enhanced macro language.

3. Parameterized Templates

The development and successful deployment of application libraries such as COOL is made easier and more useful by a language feature called parameterization. Parameterized templates allow a programmer to design and implement a class template without specifying the data type. The user customizes the template to produce a specific class by indicating the type in a program. Several versions of the same parameterized template (each with a different type) can exist in a single application. Parameterized templates can be thought of as meta-classes in that only one source base needs to be maintained to support numerous variations of a type of class.

Regardless of the type of object a parameterized class is to manipulate, the structure and organization of the class and the implementation of the member functions are the same for every version of the class. For example, a programmer providing a vector class knows that there will be several member functions such as insert, remove, print, sort, and so on that apply to every version of the class. By parameterizing the arguments and return values from the various member functions, the programmer provides only one implementation of the vector class. The user of the class then specifies the type of vector at compile-time.

An important and useful type of parameterized template is known as a container class. A container class is a special kind of parameterized class where you put objects of a particular type. For example, the **Vector**<*Type*>, **List**<*Type*>, and **Hash_Table**<*KType,Vtype*> classes are container classes because they contain a set of programmer-defined data types. Since container classes are so commonplace in many applications and programs, parameterized container classes provide a mechanism to maintain one source base for several versions of very useful data structures. COOL supplies several common container class data structures that can be used in many typical application scenarios.

Each of the COOL parameterized container classes support the notion of a built-in iterator that maintains a current position in the container and is updated by various member functions. These member functions allow you to move through the collection of objects in some order and manipulate the element value at that position. This might be used, for example, in a function that takes a pointer to a generic object that is a type of container object. The function can iterate through the elements in the container by using the current position member functions without needing to know whether the object is a vector, a list, or a queue.

In addition to this built-in current position mechanism, COOL provides support for multiple iterators over the same class by using the **Iterator** <*Type*> class. For example, a programmer may need to write a function that moves through the elements of a container class and, at some point, needs to save the current position and begin processing elements at another location. After a period of time, the secondary processing terminates, at which point flow of control returns to the previous stopping point where the current position is restored from the iterator object and processing continues.

A programmer 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, parameterized types, and the COOL macro language. CCC controls and invokes the various components of the compilation process. In particular, however, it looks for command line arguments specific to the parameterized template process and processes them accordingly as suggested by Stroustrup in his design paper[2]. Other options and arguments are passed on to the system C++ compiler control program.

4. Symbols and Packages

A package provides a relatively isolated namespace for various COOL components called symbols. A symbol that is owned by a particular package is said to be interned in that package. In general, the term interned means that a particular object is uniquely identifiable in some context. When a symbol is interned, it becomes uniquely identifiable by the symbol name within a namespace context. The package system provides logical groupings of symbols supporting relationships established between named objects and the values they contain. Although the notion of symbols being grouped into packages is fairly straightforward, the nature of the relationships that can exist between packages and the way in which they establish a namespace can be quite complex. COOL provides several kinds of macros to simplify the usage and manipulation of symbols and packages.

A symbol is a data object that defines a relationship between a name, a package, a value, and a property list. The name is a character string used to identify the symbol. Once a name is established for a symbol, it may not be changed. The value field is used to refer to some C++ object. Property lists are lists of alternating names and values. The property list allows the programmer to associate supplemental attributes with a symbol. Initially, the property list for a symbol is empty.

The Symbol and Package classes implement the fundamental COOL symbolic computing support as standard C++ classes. The Symbol class implements the notion of a symbol that has a name with an optional value and property list. Symbols are interned into a package, which is merely a mechanism for establishing separate name spaces. The Package class implements a package as a hash table of symbols and includes public member functions for adding, retrieving, updating, and removing symbols.

COOL supports efficient and flexible symbolic computing by providing symbolic constants and run-time symbol objects[5]. You can create symbolic constants at compile-time and dynamically create and manipulate symbol

objects in a package at run-time by using any of several simple macros or by directly manipulating the objects. Symbols and packages in COOL manage error message textual descriptions with translations, provide polymorphic extensions to C++ for object type and contents queries, and support sophisticated symbolic computing not normally available in conventional languages.

5. Polymorphic Management

C++ version 2.0 as specified in the AT&T language reference manual[6] implements virtual member functions that delay the binding of an object to a specific function implementation until runtime. This delayed (or dynamic) binding is useful where the type of object might be one of several kinds, all derived from some common base class but requiring a specialized implementation of a function. The classic example is that of a graphics editor where, given a base class graphic_object from which square, circle, and triangle are derived, specialized virtual member functions to calculate the area are provided. In such a system, a programmer can write a function that takes a graphic_object argument and determine its area without knowing which of all the possible kinds of graphical objects the argument really is.

This dynamic binding capability of C++, while powerful and providing greater flexibility than most other conventional programming languages, is still not enough for some types of problems. Highly dynamic languages such as Lisp allow the programmer to delay almost all decisions until runtime[7]. In addition, facilities for querying an object at runtime to determine its type or request a list of all possible member functions available are often present. These kinds of features are commonly used in many symbolic computing and complex knowledge-intensive operations management problems tackled today.

COOL supports enhanced polymorphic management capabilities with a programmer-selectable collection of macros, classes, symbolic constants, run-time symbolic objects, and dynamic packages[1]. This is facilitated by the Generic class that, combined with macros, symbols, and packages, provides efficient run-time object type checking, object query, and enhanced polymorphic functionality unavailable in the C++ language.

The Generic class is inherited by most other COOL classes and manipulates lists of symbols to manage type information. Generic adds run-time type checking and object queries, formatted print capabilities, and a describe mechanism to any derived class. The COOL class macro (discussed below) automatically generates the necessary implementation code for these member functions in the derived classes. A significant benefit of this common base class is the ability to declare heterogenous container classes parameterized over the Generic* type. These classes, combined with the current position and parameterized iterator class, lets the programmer manipulate collections of objects of different types in a simple, efficient manner.

One of the simplest and most useful features facilitated by Generic is the runtime type checking capability. The type_of() and is_type_of() virtual member functions accomplish this kind of run-time type query for an object that is derived at some point from the COOL Generic class. Type determination and function dispatch can become quite tedious, however, if there are many types of objects. Ideally, each would be derived from a common base and include support for a virtual member function for each important operation that might be required. However, it is sometimes not feasible to have such a situation, especially with a high number of objects obtained from several sources. An alternate scheme similar to the one mentioned above is the type_case macro, analogous to the C++ switch statement. It gathers all possible type cases and allows the user to symbolically dispatch on the type of object represented by the case statements. This automates some of the symbol collection and manipulation required with the earlier mechanism.

The class keyword is implemented as a COOL macro to add symbolic computing abilities to class definitions. It takes a standard C++ class definition and, if the class contains Generic somewhere in its inheritance hierarchy, generates member functions for support of run-time type checking and query. In addition, a symbol for the derived Generic class type is added to the COOL global symbol package SYM. The actual code which is expanded in a class definition and after a class definition is controlled by the classmac macro discussed below.

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 macros that can be changed

or customized by the application programmer. The COOL Generic class uses the data member hook to implement the map_over_slots() member function. 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. 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 a classmac macro. For example, a general purpose mechanism that automatically creates accessor member functions to get and set each data member can be created by defining a classmac macro that is attached to the data member hook of the class macro . No changes to the COOL preprocessor are required.

The member functions added by Generic and the class macro to derived COOL classes manipulate symbols stored in the global SYM package. These symbols reflect the inheritance tree for a specific class. They may have optional property lists containing information associating supported member functions and their respective argument lists. User-defined classes derived from Generic are also automatically supported in an identical fashion, resulting in addition symbols in the global symbol package. As discussed earlier, these symbols must have storage allocated for them and code to initialize the package at program startup time. This is managed by the COOL file symbols.C that should be compiled and linked with every application that uses COOL. An automated method for insuring correct package setup and symbol initialization is accomplished by establishing the correct dependency in an application make file.

6. Exceptions

In COOL, program anomalies are known as exceptions. An exception can be an error, but it can also be a problem such as impossible division or information overflow. Exceptions can impede the development of object-oriented libraries. Exception handling offers a solution by providing a mechanism to manage such anomalies and simplify program code. The COOL exception handling scheme is a raise, handle, and proceed mechanism similar to the Common Lisp Condition System[8]. When a program encounters an anomaly that is often (but not necessarily) an error, it can:

- Represent the anomaly in an object called an exception
- Announce the anomaly by raising the exception
- Provide solutions to the anomaly by defining and establishing handlers
- Proceed from the anomaly by invoking a handler function

The COOL exception handling facility[9] provides an exception class (Exception), an exception handler class (Excp_Handler), a set of predefined exception subclasses (Warning,, Fatal, System_Error, System_Signal, and Error), and a set of predefined exception handler functions. In addition, the macros EXCEPTION, RAISE, STOP, and VERIFY allow the programmer to easily create and raise an exception at any point in a program.

When an exception is raised (through macros RAISE or STOP, for example), a search begins for an exception handler that handles this type of exception. An exception handler, if found, deals with the exception by calling its exception handler function. The exception handler function can correct the exception and continue execution, ignore the exception and resume execution, or end the program. In COOL, an exception handler for each of the predefined exception types exists on the global exception handler stack.

An exception handler invokes a specific exception handler function for a specific type of exception. Handling an exception means proceeding from the exception. An exception handler function could report the exception to standard error and end the program, or drop a core image for use by the programmer with a debugger. Another way of proceeding is to query the user for a fix, store the fix in the exception object, and return to where the exception was raised. When an exception handler object is declared, is is placed on the top of a global exception handler stack. When an exception is raised, a call searches for a handler. The handler search starts at the top of the exception handler stack.

There are six predefined exception type classes provided as part of COOL. The exception class is the base class from which specialized exception subclasses are derived. Derived from Exception are Warning, System_Signal, Fatal and Error. From the Error class, the System_Error and Verify_Error classes are derived. The default

exception handlers are called only if no other exception handler is established and available when an exception is raised. COOL offers users the option of defining their own exception types. Such types can be derived from the Exception class of one of the derived exception types. All user-defined exception classes should have public data slots.

The COOL exception handling facility provides several macros which simplify the process of creating, raising, and manipulating exceptions. These macros are implemented with the COOL macro facility. The EXCEPTION macro simplifies the process of creating an instance of a particular type of exception object. The RAISE macro allows the programmer to easily raise an exception and search for an exception handler. The STOP macro is similar to the the RAISE macro, except that it guarantees to end the program if the exception is not handled. The VERIFY macro raises an exception if an assertion for some particular expression evaluates to FALSE. Finally, the IGNORE_ERRORS macro provides a mechanism to ignore an exception raised while executing a body of statements.

The COOL exception handling mechanism is similar in several ways to that proposed by Koenig and Stroustrup[10]. Their try statement and catch clauses can be implemented by using a COOL exception handler with the system functions **setjmp** and **longjmp**. In addition, both designs treat exceptions as objects that are instantiated and passed as arguments to handler functions. The AT&T proposal proposes a grouping of exception objects as a mechanism for organizing exceptions into groups similar to the compile-time mechanism for organizing classes into hierarchies. COOL, on the other hand, maintains an exception derivation hierarchy and uses some of the COOL symbolic computing facilities for run-time type checking and query. A grouping of exception names similar to the AT&T proposal is currently being implemented where alias name(s) may be defined for an exception object. For example, rather than defining a new exception class derived from Error, an Error exception can be raised with the specified alias/group name(s) passed as an argument. This is expected to reduce the need for many different types of exception classes whose only difference is the type name.

7. Coding Style and Conventions

A standard source code style allows several programmers to easily maintain and understand each other's code because additional semantic information can be inferred from a section's format and style. In addition, a single style presents a more coherent, professional software package for potential source code users. This is particularly important for COOL, since parameterized templates require complete access to all source code. Finally, one of the foundations of object-oriented programming is code reuse. This is much easier if a programmer is able to browse through source code and understand its organization and layout. The COOL source code addresses the following C++ coding style conventions:

- Variable and class naming conventions
- Organization and contents of class header files
- Private/Protected/Public data members
- Source code documentation
- Source code indentation and layout
- Error message text resource package
- Regression test suite
- Source code system independence
- Build procedure

7.1. Naming Conventions

A prime objective for a naming convention is allowing programmers to recognize what sort of component a name refers to. Another goal is using meaningful names, which has not typically been done in C applications. The following naming conventions are used throughout the COOL source code. The reader is strongly encouraged to follow the same guidelines:

- Directory, .C, and .h file names should be the same or close to the class being defined and the declaration and implement files should be in a single directory. For example, the String class is defined and

implemented in the files String.h and String.C and contained in the ~COOL/String subdirectory.

- Class, struct, and typedef names should be capitalized with the words separated by underscores:

```
class Generic_Window { ... };
struct String_Layout { ... };
typedef int Boolean;
```

- All function names should be lowercase with each word separated by an underscore character:

```
void my_fun (int foo);
char* get_name (ostream&);
```

- Predicate functions should begin with is_:

```
Boolean is_type_of (int);
```

- Variable and data member names should be lowercase with words separated by underscores:

```
int ref_count;
char* name;
```

- Global and static variables should be appended with _g or _s, respectively:

```
int node_count_g;
static char* version_s;
```

- Preprocessor statements and MACRO names should be uppercase:

```
#define ABS ((x < 0) ? (-x) : x)
```

- Constants (const) declarations should be uppercase:

```
const int FALSE=0;
const int TRUE=!FALSE;
```

7.2. Class Header File Organization

All header files defining the structure of a class or parameterized template should be organized into sections in the following order:

- Included files and typedefs necessary for the class.
- Definition of private data members.
- Declaration of private member functions and friends.
- Definition of protected data members.
- Declaration of protected member functions and friends.
- Declaration of public member functions and friends.
- Inline member functions of the class follow the class definition.
- Other member/friend function definitions are in a separate file.

In general, only the data member definitions and function prototypes of the member functions and friend functions should appear in the class construct. This separates the implementation from the specification and reduces clutter. Define inline functions after the class {...}; statements. In addition, the keyword inline should appear in both the class definition and in the actual implementation as a documentation aid. The optional private keyword usage is explicitly stated. Finally, avoid multiple instances of scoped sections: There should be no more than one each of the private, protected, and public labels.

7.3. Private, Protected, and Public Data

In general, class data should be encapsulated in either the private or protected sections. Data specific to a particular class with no use for possible derived classes should be located in the private section. Data located in the protected section might include such things as configuration or adjustment data members that a derived class might want to monitor or change. No COOL classes contain public data, and the user should not declare such data. Aside from being bad object-oriented programming style, classes with public data cannot be made persistent and stored in the OODB. The one exception to this standard are the derived exception classes which may require public data members in order to allow query and/or update of alternate values.

7.4. Documentation

Documentation of all files is very important. Terseness should be the general rule for all header files and completeness the rule for all code files. Parameterized templates have a single header/source file and all documentation should be located there. If in doubt, more documentation is better than less documentation. A high-level abstract at the top of each file should provide a description of the file's functionality. Class header files should also contain a brief description of the public interface.

Each function in a source code file should have a preceding block comment specifying the input and output parameters as well as giving a brief synopsis of the functionality. For complex inline definitions in header files, a block comment of this type should only be used when the purpose is not obvious because these comments do not appear in the code file. Since most inline functions contain trivial code (usually providing an accessor to some private data member), comment requirements for inline function can be relaxed.

All source code should be commented every few source lines. Specifically, large block comments every 100 lines is unacceptable. No comment should contain operating system specific names or terms unless that section of code is truly specific. When this is necessary, the code should be surrounded by conditional compilation constructs. These are handled by the preprocessor relative to that specific operating system.

Finally, documentation in the form of a man page should be written for every class. Layout and organization will be as that with the -man macro package available for nroff(1)/troff(1). Section names and requirements for a class man page include Name, Synopsis, Base Class, Friend Classes, Description, Constructors (public or protected as necessary), Protected Member Functions (when appropriate), Public Member Functions, Files, See Also, and Bugs (when necessary). Introductory and high-level material can and should also be documented.

7.5. Source Code Indentation

Indentation and source code structure is relaxed, but it is suggested that the programmer use the C++ mode available for GNU Emacs and supplied with COOL. In general, statements should be restricted to one line with indentation reflecting block and scoping visibility. Location of such items as braces, spacing around parentheses, and so on is left up to the programmer. If the C++ mode is used, whole regions can be marked and indented appropriately, providing a simple means by which all source code can be brought into the same format.

7.6. Error Message Resource Package

All error message text strings in an application should use the ERR_MSG package available in COOL. The COOL exception handling scheme automatically uses this package insuring that all text strings associated with error messages are stored as the value of a symbol. All error message symbols are automatically processed and located in one file, thus facilitating easy update or configuration. In particular, a language translation can be added to the property list of each symbol entry, providing an efficient and convenient means for internationalizing the text messages in an application.

7.7. Regression Test Suite

Each new or modified class contained in or added to COOL must also include a standalone test program. This should fully exercise all features and functions and report success or failure via the test macros contained in the

~COOL/include/test.h header file. This test program is used in regression tests for new releases and ports to other software platforms to insure a complete and working implementation.

7.8. Source Code System Independence

COOL places great importance upon system independent code and features. As such, system-specific functions should be surrounded with #if preprocessor directives where appropriate. In general small performance sacrifices in implementation are preferred if system independence and portability is improved.

7.9. Build Procedure

COOL contains a modified Imake utility from the MIT X11R3 source tape that implements a system-independent build procedure. This should be used for all new classes and source code. Imake provides configuration and rules files for localization and/or customization of system build utilities and commands to aid in porting activities to other operating systems and hardware platforms.

8. Class Hierarchy

The C++ Object-Oriented Library (COOL) class hierarchy implements a rather flat inheritance tree, as opposed to the deeply nested SmallTalk model. All complex classes are derived from the Generic class, to facilitate run-time type checking and object query Simple classes are not derived from Generic due to space efficiency concerns. The parameterized container classes all inherit from a base class that results in shared type-independent code. This reduces code replication when a particular type of container is parameterized several times for different objects in a single application. The COOL class hierarchy is as follows:

Pair<*T1*,*T2*>
Range
    Range<*Type*>
Rational
Complex
Generic
    String
    Gen_String
    Regexp
    Vector
        Vector<*Type*>
            Association<*Pair<T1,T2>*>
    List_Node
        List_Node<*Type*>
    List
        List<*Type*>
    Date_Time
    Timer
    Bit_Set
    Exception
        Warning
        Error
            System_Error
        Fatal
        System_Signal
        Verify_Error
    Excp_Handler
        Jump_Handler
    Hash_Table
        Set
        Hash_Table<*Key*,*Value*>

Package
Matrix
Matrix<*Type*>
Queue
Queue<*Type*>
Random
Stack
Stack<*Type*>
Symbol
Binary_Node
Binary_Node<*Type*>
Binary_Tree
Binary_Tree<*Type*>
AVL_Tree<*Type*>
N_Node<*Type*>
D_Node<*Type*>
N_Tree<*Type,Node,nchild*>

8.1. String Classes

The String class provides dynamic, efficient strings for a C++ application. The intent is to provide efficient char*-like functionality that frees the programmer from worrying about memory allocation and deallocation problems, yet retains the speed and compactness of a standard char* implementation. All typical string operations including concatenation, case-sensitive and case-insensitive lexical comparison, string search, yank, delete, and replacement are provided.

The Regexp class provides a convenient mechanism to present regular expressions for complex pattern matching and replacement and utilizes the built-in char* data type. The Gen_String class provides general-purpose, dynamic strings for a C++ application with support for reference counting, delayed copy, and regular expression pattern matching. The intent is to provide a sophisticated character string function for the application programmer. The Gen_String class combines the functions of the String and Regexp classes, along with reference counting and self-garbage collection, to provide advanced character string manipulation.

8.2. Number Classes

The COOL number classes are a collection of numerically-oriented classes that augment the built-in numerical data types to provide such features as extended precision, range-checked types, and complex numbers. The Random class implements five variations of random number generators. The Complex class implements the complex number type for C++ and provides basic arithmetic and trigonometric functions, conversion to and from built-in types, and simple arithmetic exception handling. The Rational class implements an extended precision rational data type for inadequate round-off and/or truncation results from the built-in numerical data types. Finally, the parameterized **Range** <*Type*> class enables arbitrary user-defined ranges to be implemented in C++ classes. Typically, this is used with other number classes to select a range of valid values for a particular numerical type.

8.3. System Interface Classes

System Interface classes include classes for calculating the date and time in different timezones and countries and measuring the time duration between two points in some application program. The Date_Time class executes time zone-independent date and time functions. This class also supports all time zones in the world, along with several special cases requiring alternate handling based upon political or daylight saving time differences. The Timer class is publicly derived from Generic and provides an interface to system timing. It allows a C++ program to record the time between a reference point (mark) and now.

8.4. Ordered Sequence Classes

The ordered sequence classes are a collection of basic data structures that implement sequential access data structures as parameterized classes, thus allowing the user to customize a generic template to create a user-defined class. The **Vector**<*Type*> class implements single dimension vectors of a user-specified type. The **Stack**<*Type*> class implements a conventional first-in, last-out data structure, similar to the **Queue**<*Type*> class but it implements a conventional first-in, first-out data structure. These two classes each hold a user-specified data type. The **Matrix**<*Type*> class implements two-dimensional arithmetic matrices for a user-specified numeric data type. The vector, stack, and queue classes can be dynamic in size.

8.5. Unordered Sequence Classes

The unordered sequence classes are a collection of basic data structures that implement random access data structures as parameterized classes, thus allowing the user to customize a generic template to create a specific user-defined class. The **List**<*Type*> class implements Common Lisp style lists providing a collection of member functions for list manipulation and management. A list consists of a collection of nodes, each of which contains a reference count, a pointer to the next node in the list, and a data element of a user-specified type. The **Pair**<*T1,T2*> class implements an association between one object and another. The objects may be of different types, with the first representing the key of the pair and the second representing the value of the pair. The **Association**<*Ktype,Vtype*> class is privately derived from the **Vector**<*Type*> class and implements a collection of pairs. The first of the pair is called the key and the second of the pair is called the value. The **Hash_Table**<*Ktype,VType*> class implements hash tables of user-specified types for the key and the value.

8.6. Set Classes

The set classes implement two basic data structures for random access set operations as parameterized classes, thus allowing the user to customize a generic template to create a specific user-defined class. The **Set**<*Type*> class implements random access sets of objects of a user-specified type using the parameterized type capability of C++. Classical set operations such as union, intersection, and difference are available. The **Set**<*Type*> class is publicly derived from the **Hash_Table**<*KType,VType*> class and is dynamic in nature. The Bit_Set class implements efficient bit sets. These bits are stored in an arbitrary length vector of bytes (unsigned char) large enough to represent the specified number of elements. Elements can be integers, enumerated values, constant symbols from the enumeration package, or any other type of object or expression that results in an integral value.

8.7. Node and Tree Classes

The node and tree classes are a collection of basic data structures that implement several standard tree data structures as parameterized classes, thus allowing the user to customize a generic template to create a specific user-defined class. The **Binary_Node**<*Type*> class implements parameterized nodes for binary trees. The **Binary_Tree**<*Type*> class implements simple, dynamic, sorted sequences in a tree where each node has two subtree pointers. The **AVL_Tree**<*Type*> class implements height-balanced, dynamic, binary trees. The **AVL_Tree**<*Type*> class is publicly derived from the **Binary_Tree**<*Type*> class.

The **N_Node**<*Type,nchild*> class implements parameterized nodes of a static size for n-ary trees. This node class is parameterized for both the type and some initial number of subtrees that each node may have. The constructors for the **N_Node**<*Type,nchild*> class are declared in the public section to allow the user to create nodes and control the building and structure of an n-ary tree where the ordering can have a specific meaning, as with an expression tree. The **D_Node**<*Type,nchild*> class implements parameterized nodes of a dynamic size for n-ary trees. This node class is parameterized for the type and some initial number of subtrees that each node may have. The **D_Node**<*Type,nchild*> class is dynamic in the sense that the number of subtrees allowed for each node is not fixed. **D_Node**<*Type,nchild*> uses the **Vector**<*Type*> class, supporting run-time growth characteristics.

The **N_Tree**<*Node,Type,nchild*> class implements n-ary trees, providing the organizational structure for a tree (collection) of nodes, but knowing nothing about the specific type of node used. **N_Tree**<*Node,Type,nchild*> is parameterized over a node type, a data type, and subtree count, where the node specified must have a data

member of the same Type as the tree class. The subtree count indicates the number of possible subtree pointers (children) from any given node. Two node classes are provided, but others can also be written.

9. Status of COOL

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, 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] Stanley Lippman, *C++ Primer*, Addison-Wesley, Reading, MA, 1989.

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

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

[4] Mary Fontana, Martin Neath and Lamott Oren, *A Sophisticated C++ Macro Facility*, Information Technology Group, Austin, TX. Internal Original Issue January 1990.

[5] Mary Fontana, Martin Neath and Lamott Oren, *Symbolic Computing for C++*, Information Technology Group, Austin, TX. Internal Original Issue January 1990.

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

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

[8] Andy Daniels and Kent Pitman, *Common Lisp Condition System Revision #18*, ANSI X3J13 subcommittee on Error Handling, March 1988.

[9] Mary Fontana, Martin Neath and Lamott Oren, *Exception Handling in COOL*, Information Technology Group, Austin, TX. Internal Original Issue January 1990.

[10] Andrew Koenig and Bjarne Stroustrup, *Exception Handling for C++*, C++ At Work Conference, Boston, MA, November 6-8, 1989.
