Checked Out And Long Overdue: Experiences in the Design of a C++ Class Library

Mary Fontana

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 is a portable collection of classes, templates and macros for use by C++ programmers writing complex applications. Developed over a two year period, it has been used on several internal projects and undergone significant design changes and improvements. In this paper, we discuss the initial goals of the project, the design and implementation approaches considered, and some of the reasons behind our decisions. Finally, we analyze what was learned in building this library, examine the overall issue of code reuse through C++ class libraries, and suggest some guidelines that can lead to wider acceptance and use of future class libraries.

1. Introduction

The Texas Instruments (TI) C++ Object-Oriented Library (COOL) is a portable collection of classes, templates, and macros for use by C++ programmers writing complex applications. It raises the level of abstraction to allow the programmer to concentrate on the problem domain, not on implementing basic data structures, macros, and classes. In addition, COOL provides a system independent software platform to ease the porting of applications which are built on top of it. In this paper we discuss the rationale behind some of the important aspects of COOL, such as its use of polymorphism, parameterized templates, and a resumptive exception handling mechanism. We also share what we learned in designing and implementing COOL and the feedback obtained from application programmers who have used it.

Our motivation for use of C++ and development of a rich class library has its roots in our extensive experience with Lisp Machine environments. TI has had considerable success using Lisp to design and implement complex, symbolic applications, such as diagnostic expert systems and production scheduling advisors. While most customers were willing to see Lisp used for prototyping, many showed considerable resistance to Lisp Machines as delivery vehicles.

The authors may be reached via electronic mail at fontana@csc.ti.com and martin@tivoli.com.
Martin Neath now works for TIVOLI Systems, Inc., Austin, TX.

As a result, we began investigating other, more mainstream languages that can provide some of the expressiveness of Lisp and which are well supported on a variety of conventional platforms. After evaluating several languages, we decided (for mainly non-technical reasons) on C++ and began the design of a comprehensive class library.

Over the course of about one year we designed and implemented many generalized classes. We began with the basics (such as, **String**, **Date_Time**, and **Complex**) to gain experience with the language, examine possible design approaches, and understand portability and efficiency issues. We next added an implementation of Stroustrup's templates [12] (such that there would be minimal source code conversion necessary when this feature is finally implemented in the C++ language) and proceeded to design and implement a variety of parameterized, polymorphic container classes (such as, **Vector**<*Type*> and **N_Tree**<*Node,Type,nchild*>). As the project proceeded, we realized the need for an object-oriented exception handling mechanism. Since no such facility had yet been proposed, we designed and implemented a resumptive capability for raising and handling exceptions similar in spirit to the Common Lisp Condition System [2]. Finally, a comprehensive, automatic runtime type query system was completed to round out the symbolic capabilities of the library.

COOL has been an ever-changing and growing C++ class library, with considerable effort spent reimplementing internal details, adding new features, extracting common functionality into base classes, etc. As such, some constraints were necessary in order to achieve compatible and seamless integration of new or modified features. Overall, the design and development of a C++ class library has been a very valuable experience, as much for the things we learned *not* to do, as well as for the positive feedback we received for the things we did correctly. This class library is currently in use on several internal projects and is largely in a maintenance-only mode of development. We expect to make the necessary changes to support the standard parameterized template and exception handling mechanisms when those features become available in commercial compilers. For more detailed information and examples of the COOL classes, the reader is referred to the appropriate sections of the reference document, *The COOL User's Guide* [13].

2. Core Technology Components

The fundamental cornerstones of COOL are an implementation of parameterized templates, a resumptive exception handling mechanism, an automated runtime type checking facility, and consistent polymorphic operations. This functionality is implemented through an enhanced preprocessor with a sophisticated macro facility [4] which generates conventional C++ source code acceptable to any conforming C++ translator or compiler [3]. The use of this compiler independent front-end allowed us to define powerful and portable extensions to the C++ language in an unobtrusive manner. This enabled us to experiment and gain experience with a variety of proposed language extensions long before they were available in a commercial product.

2.1. Parameterized Templates

We quickly found that the development and successful deployment of application libraries such as COOL required the planned (but not yet available) C++ language feature called type parameterization. This allows a class to be defined without specifying the specific data types needed. The application programmer using the class specifies the data types for each unique use of it in the application code.

An important and useful variety of parameterized template is known as a container class. This

is a special kind of parameterized class where objects of some type are structured and stored together. COOL supplies the many common containers needed by typical complex applications. These have turned out to be the most important and most used classes in the COOL library. Indeed, the container classes come closest of all the classes in COOL to fulfilling the promise of true code reuse.

Each COOL container class supports the notion of a built-in iterator that maintains a current position within the collection of objects. A set of consistently named member functions allows a program to move through the collection of objects in a sequential order and manipulate the element at the current position. This might be used, for example, in a function that takes a pointer to a generic 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. The capability to easily replace one type of container with another is critical since complex applications often must change their data storage mechanisms to meet requirements that evolve over time.

Several interesting issues arose for both the COOL and application programmers in designing and using parameterized classes. First, should a template make assumptions about or enforce a specific type modifier over which the class is to be parameterized or should that be part of the usage specification? Second, how much code for a template class can be moved into a base class to reduce code replication? Third, can the source code be effectively packaged to provide a rich set of member functions without burdening applications that do not use them all? Finally, what is the most effective mechanism for introducing a new parameterized type to the compiler and arranging for the inclusion of the code for that type exactly once across file boundaries?

The answers to some of these questions seemed obvious, while others required several attempts before a comfortable and correct direction was selected. When we began considering the *Type* parameter to a template, it seemed appropriate to allow the user to control the type modifier specification. This would allow one user to use a **Vector**<*Type*> template to contain "integers", while another might select "pointers to integers". Although a single template class can satisfy both uses, some slight performance degradation and loss of efficiency may result when copying and accessing a contained object. For example, the **operator[]** may return a reference that, if it is a pointer, results in an extra pointer dereference. The design decision to not enforce a particular type modifier resulted in the copy semantics of the contained object being left to the user and the template member functions using the object's **operator=** to copy and move objects.

One early decision was to design each parameterized class to inherit from an appropriate 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 base classes typically included class-specific data members, member functions which manipulated the current position in container classes, and member functions which raised exceptions. There would have been more code in the base classes if we had decided differently on the previous issue and restricted the type parameter to data type pointers only.

The third issue (which would not even be an issue if more sophisticated linkers were available on standard operating systems) concerns the "full-featured" versus "lean-and-mean" philosophies. After considerable analysis and experimentation, we decided to embrace both philosophies by providing rich functionality with a template fracturing capability. This mechanism

splits the source file on template boundaries so that each member function is copied into and compiled from its own file. The resulting object files, one for each function, are then placed in an application archive library for use at link time. This provides for only those member functions that are actually used in an application to be pulled into the executable image.

To control the introduction of a new parameterized type to the compiler and to automate the generation of the member functions, our first attempt used **DECLARE** and **IMPLEMENT** macros that were carefully located in the application source code. This was later changed to allow a command line interface through a compilation control program, where the user specifies the parameterized type on the command line as suggested by Stroustrup with the -X compiler option [12]. This mechanism is quite usable in traditional separate compilation systems, but more elegant solutions (which might have additional benefits) are possible for use in emerging integrated C++ programming environments.

We have found through our experience with COOL that parameterized container classes are the most important part of a general C++ class library. In addition, the basic design approach taken for container classes and the way in which the open issues with parameterization are solved determine the ultimate acceptance and use of the class library by application programmers.

2.2. Exception Handling

In COOL, program anomalies are known as exceptions. An exception can be a program error such as an argument out of range, or an encapsulation of a more fundamental problem such as arithmetic overflow. We believe that the current lack of an exception mechanism in the language seriously impedes the development of flexible and portable object-oriented libraries. An exception handling system offers a solution by providing a mechanism to manage such anomalies, simplify program code, and ease portability of an application. As an interim measure, we developed the COOL exception handling scheme, which is a raise, handle, and proceed mechanism similar to the Common Lisp Condition System [2].

The COOL exception handling facility [5] provides an exception class (**Exception**), an exception handler class (**Excp_Handler**), a set of predefined exception subclasses (**Warning**, **Error**, **Fatal**, **System_Error**, and **System_Signal**), and a set of predefined exception handler functions. In addition, an easy-to-use macro interface (**EXCEPTION**, **RAISE**, **STOP**, **VERIFY**, **DO_WITH_HANDLER**, and **HANDLER_CASE**) allows a programmer to create and raise an exception, and establish exception handlers at any point in a program. When a COOL class encounters an anomaly that is often (but not necessarily) an error, it represents the anomaly in an object called an exception and then announces the anomaly by raising the exception. The application program using COOL classes has the option of providing solutions to the anomaly by defining exception handler functions and establishing exception handler objects.

When an exception handler object is created, it is placed at the top of a global exception handler stack. This stack is maintained in a similar way to that described by Miller [11]. When an exception is raised, a search for an appropriate handler starts at the top of the exception handler stack. When a match against an exception type is found, the exception handler object invokes its handler function. COOL provides default exception handlers for the predefined exception types, such as reporting a description of the exception to the standard error stream and exiting the program or dumping a core image. A default handler is only invoked if no handler for the raised exception is found on the global exception handler stack.

The COOL exception handling macros, **RAISE** and **HANDLER_CASE**, provide the same type of functionality as the **throw** and **try/catch** statements proposed by Koenig and Stroustrup [9]. Both **throw** and **RAISE** transfer control to the most recently established handler for a particular type of exception. However, any object may be used as an argument in a **throw** expression, whereas **RAISE** only allows exception objects. In a similar manner, the **try/catch** block and the **HANDLER_CASE** macro establish handlers while executing a body of statements. The difference here is that the **catch** expression in a **try** block is like a function definition and any data type can be specified in the declaration. The case statements in the **HANDLER_CASE** macro, on the other hand, accept only COOL symbols which identify an exception type.

The differences mentioned above are minor, however, when compared to the philosophical models each system follows: termination versus resumption. In the one, the **throw** unwinds the stack before the call of the exception handler in the **try/catch**, thus supporting a termination model for exception handling, while in the second, **RAISE** expands into a function call which searches for an exception handler to invoke, thus supporting the resumptive model of exception handling.

It is interesting to note that although COOL allows both termination and resumptive models for handling exceptions, only default handlers and termination (or more appropriately, retry) handlers were used for exceptions raised in the COOL classes. Support for a resumptive model did not require much additional implementation work, but we discovered that the termination/retry model is the most appropriate for a generalized class library. A tight binding (or contract) between the class member function invoking an exception and the application function in which the exception might be resumed is absolutely necessary to ensure that *all* semantic and state information is transmitted and understood effectively by a handler. It is unlikely that this scenario is true in anything other than tightly coupled modules of a single application, which makes the usefulness of supporting a resumptive system questionable.

2.3. Symbolic Computing

COOL supports efficient and flexible symbolic computing by providing symbolic constants and runtime symbol objects [7]. You can create symbolic constants at compile-time and dynamically create and modify symbol objects at runtime by using a simple macro interface or by directly manipulating the objects. Symbols and packages are used within COOL to manage error message text for translation, to provide polymorphic extensions for object type and contents queries, and to support sophisticated symbolic operations not normally available in conventional compiled languages.

The fundamental COOL symbolic computing capability is supported through the **Symbol** and **Package** classes. The **Symbol** class implements the notion of a symbol that has a name with an optional value and property list. The name is a character string used to identify the symbol. The value field refers to some C++ object. Property lists are lists of alternating names and values which allow the programmer to associate supplemental attributes with a symbol. This property list feature has been used, for example, to easily add an international representation for all message strings to an application by representing the messages as symbol objects with the translations for different languages stored on the property list.

Symbols are interned into a package, which is merely a mechanism for establishing isolated namespaces. The **Package** class implements a package as a hash table of symbols and includes member functions for adding, retrieving, updating, and removing symbols. This

package information is maintained across file module boundaries in an application-specific file, providing a crude application database for the registration of shared information. This file is used in COOL to store such things as class hierarchy information, class names, and the location of where a parameterized template class is generated. This last item is necessary in order to automate the expansion of a template exactly once within a single application. Clearly, such implementation techniques are an indication that C++ is stretching the limits of the separate compilation model of software development traditional in the UNIX® environment. We have found this type of inter-file support, from both the language and the supporting programming tools, to be absolutely necessary for the productive development of complex C++ applications using libraries of reusable components.

2.4. Runtime Type Support

COOL supports an efficient runtime type checking and query capability, and a describe mechanism for classes which derive from the **Generic** class [6]. The COOL preprocessor automatically generates for each **Generic**-derived class a list of symbols which provides the class type and class inheritance information and which is used by the **type_of()** and **is_type_of()** member functions of the **Generic** class.

The **Generic** class is inherited by most of the COOL classes. A significant benefit of this common base class is the ability to declare heterogeneous container classes parameterized over the **Generic*** type. These classes, combined with the current position and parameterized **Iterator** class, allow the programmer to manipulate collections of objects of different types in a simple, efficient, and extensible manner.

The symbols generated by the COOL preprocessor are added to a single file that functions as the application symbol repository. This file is compiled and linked with the application to allocate storage, and to initialize the symbols and the global symbol package at program startup time. An automated method for insuring correct package setup and symbol initialization is accomplished by establishing the correct dependency in an application makefile, and through global static object initialization supported by the C++ language.

The power of the symbolic computing features available in COOL significantly enhances the ability of the application programmer to solve problems in a variety of domains. Unfortunately, the complexity of the symbol system and the necessity for an application-specific database to support it has severely limited its use. We believe that many of the questions and difficulties reported to us are directly or indirectly related to this feature. The basic problem is the lack of a containing environment with knowledge about the whole application structure. This problem is reflected in the template expansion process, the runtime type system, and the symbol/package mechanism. In addition, the difficulty C++ compilers are having with enforcement of the "one-definition" rule and 100% type-safe linkage can also be directly traced to this problem. We do not believe that file-based storage repositories are the answer, no matter how much automation and "magic" is used. Fundamentally, these types of issues require a supportive environment for a robust and elegant solution, much as is found in other languages such as Lisp and Smalltalk.

---

3. Class Hierarchy Overview

The COOL class hierarchy is a rather flat inheritance tree, as opposed to the deeply nested Smalltalk model. All complex classes are derived from the **Generic** class to facilitate runtime type checking and object query. Simple classes are not derived from **Generic** due to space and efficiency concerns. Each parameterized container class inherits from a base class which includes all type-independent code. The COOL class hierarchy is as follows:

Bignum
Complex
Pair<*Type1,Type2*>
Range
  Range<*Type*>
Rational

Generic
  Binary_Node
    Binary_Node<*Type*>
  Binary_Tree
    Binary_Tree<*Type*>
      AVL_Tree<*Type*>
  Bit_Set
  Date_Time
  Exception
    Error
      System_Error
      Verify_Error
    Fatal
    System_Signal
    Warning
  Excp_Handler
    Jump_Handler
  Generic<Type>
  Gen_String
  Vector
    Vector<*Type*>
      Association<*Type1,Type2*>
  List_Node
    List_Node<*Type*>
  List
    List<*Type*>
  Hash_Table
    Hash_Table<*Type1,Type2*>
      Package
    Set<*Type*>
  Iterator<*Type*>
  Matrix
    Matrix<*Type*>
  D_Node<*Type,nchild*>
  N_Node<*Type,nchild*>
  N_Tree<*Node,Type,nchild*>

Generic (cont'd)
- Queue
  - Queue<*Type*>
- Stack
  - Stack<*Type*>
- Symbol
- String
- Timer
- Random
- Regexp

4. Who Tried To Use COOL And Were They Successful?

A C++ class library should be targeted for a particular audience and domain in order for some measure of success to be easily determined. Initially, the COOL project had no specific customer in mind. Our project goals were to gain experience with the C++ language and determine if a rich collection of classes could be designed and used by a variety of application programmers in a practical and worthwhile manner. At the time, there were several groups in different parts of the corporation that had expressed some interest in the language. Many of these users were former Lisp programmers. Others were considering using ADA, while still others were C programmers on UNIX and PC platforms. To collect and disseminate information to this diverse user-group, we established an e-mail forum for discussion of ideas, issues, design reviews, and so forth. This provided valuable information and insight into the needs of a variety of customers. It also, however, resulted in many conflicting requests that required space/time tradeoffs in the class designs and implementations.

The potential users turned out to be programmers who were likely to use C++ and our class library in the short term and programmers who had no immediate need or opportunity for use, but were interested for possible future projects, intellectual, or personal reasons. Several projects had just begun their design and prototyping phases when they evaluated COOL. One project which was building a VHDL simulator, decided not to use COOL and developed their own C++ classes for performance and efficiency reasons. A second project which was working on an embeddable forward-chaining rules system, used our class library and the extended features such as parameterized types and the symbolic computing capability. Another project used some components of our class library, yet also designed their own versions of some classes too. In each case, there was a desire and need for many of the basic data structures found in COOL. The answer to the question "Were they successful?" is "partially". The primary reason for our lack of success was that programmer expectations, design, requirements, and the C++ language itself were not always in line with each other. The following sections contain details about which COOL components were used and how they were or were not appropriate for the work at hand.

4.1. What Did Our Users Like?

To date, the two most favorably received aspects of COOL are the implementation of parameterized templates and the portable nature of the library. The rules compiler project mentioned above extensively used not only our collection of parameterized container classes, but also wrote several application-specific template classes using the same mechanism. In general, we have received favorable response from this project concerning the syntax and expressive power of the template mechanism. Some project leaders expressed considerable willingness to give up a small percentage of performance and/or efficiency if that resulted in a highly portable

software platform upon which they could design and build their applications. This position, however, was not universally shared with the engineers responsible for implementing the application.

Another strongly echoed statement is that the distribution of the class library in source format significantly enhanced the understanding of the C++ language, and the use of the classes, polymorphism, and class derivation within an application framework. Many users were in the process of learning C++ and found the ability to examine working source code for various class library components a great aid. We also found that all possible uses of a given class could not always be anticipated in our iterative design process, so that decisions such as the **private**/**protected** interface were often incorrectly specified. In addition, when an application uses multiple inheritance with library classes, it sometimes becomes necessary to change the inheritance specification for one or more shared base classes to **virtual**. Finally, 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 kind of class. This requires distribution of template source code unless each vendor adopts its own encrypted or partially compiled format. This, however, seems too restrictive and not desirable from the user's point of view.

We often found that users who needed a particular type of class would examine the COOL design and implementation for a similar class, then proceed to copy the source code and significantly alter and modify the interface, resulting in a class very different to that initially supplied. The reason most often sighted for this course of action was not due to functionality deficiencies or difficulty with the **private**/**protected** interface, but rather a perception that it must be inefficient because it was not hand-crafted by the individual. This reaction is at the heart of the problem of code reuse and whether or not programmers will accept such a course of action. It seems to depend partially upon the individual's "pain-threshold" for modifying and/or creating a new class verses the perceived value of the library class. For small-value classes such as string, the answers seems to be variable, but for larger value classes such as regular expression and text buffer, the decision is much more likely to favor using the library class [1].

We feel that for the most part, COOL provides very efficient implementations of a variety of data structures. However, the full-featured nature of the classes may be inappropriate for all users. We originally implemented one heavy-weight **String** class, for example, that contained pattern matching capabilities and implemented reference counting and other memory management techniques. We later added a streamlined version of this class that had a subset of the member functions and provided only the most basic string operations. In many situations, this was the more popular class of the two. In those cases where a more full-featured class was needed, the ability to upgrade and have a compatible interface was appreciated. A similar request was made for the **List**<*Type*> class. We therefore feel that one possible design choice is to provide two libraries (which, if we were in advertising, would be promoted as COOL and COOL-Lite). This approach seems appropriate for general purpose class libraries and could be applied to other more specific categories. For example, a database library might have light-weight classes that provide basic storage and retrieval facilities perhaps built on a flat file ISAM for speed and portability, but also offered a richer and more powerful class with concurrency control, nested transaction support, logging and recovery.

4.2. What Did Our Users Dislike?

The most common problem voiced from users concerned the requirement that any of their applications that used a COOL class derived from **Generic** required that the entire symbol and package mechanism also be linked into their application, even if they did not use the symbolic computing facilities. This is a problem inherent with class hierarchies and libraries with complex or intertwined dependencies. For example, a class derived from **Generic** will result in implementations of the templates for **Hash_Table** and **Package** also being linked into the executable image. An additional concern already mentioned is the complexity of the symbol setup and the necessity for an auxiliary database file to be compiled and linked with each application. This is partially due to our implementation and the necessity for portability, but also because of the state of current linker technology on many platforms. Many commercial vendors implementing environments should not have this problem.

Another significant difficulty was the communication between the library developers and the library users on the intended use of and interface for the classes. A C++ class browser utility would greatly simplify the problems of educating a programmer about the available classes and their functionality. As class libraries grow and the relationships between objects become more complex, the usefulness and applicability of traditional tools such as **grep**(1) and **more**(1) begin to break down. More modern tools designed for this problem such as the graphical class browser in the Saber C++™ development environment will substantially ease the learning curve and information explosion. Using such tools, a programmer will be able to more easily identify inheritance problems, locate state and member function definitions, and assimilate a more complete mental model of the library. This will be particularly true if in fact the promise of integrating several class libraries for different components within a single application is to be realized.

5. What Did We Learn and What Would We Do Differently?

With what we now know, we would make several different design decisions, the first and foremost of which would be to simplify the interdependencies between the classes. This would be accomplished by essentially flipping the class hierarchy. Instead of having a base class **Generic** that provides the run time typing capability and from which most other classes are derived, we would provide this class as a standalone class. Classes such as **String** and **Vector**<*Type*> would not be derived from **Generic**. In this manner, users who wanted the functionality of one or more objects found in COOL would not get runtime type capability linked into their application. Those users who needed the symbolic computing facilities could use multiple inheritance to derive the appropriate class. For example, a **String** class with type-query support could be multiply derived from **Generic** and **String** to produce the desired result.

The **Exception** and **Excp_Handler** classes are the only classes in COOL which require the runtime type checking capability and symbol and package mechanism. This results from our current implementation of handling exceptions. We would change this implementation and use simple character strings instead of COOL symbol objects to represent each exception class name. Most of the COOL classes raise exceptions and we would still want to eliminate the necessity of including the symbol and package mechanism when using any of these classes.

In addition to removing the **Generic** dependency in all classes, we would also provide a

---

Saber C++ is a trademark of Saber Software, Inc.

common base class for the container classes that support the notion of an iterator object to allow for the commonality of these class objects. This base class would probably contain only pure virtual member function specifications to enforce a particular interface in the derived classes.

Finally, we would also provide both a simple version and full-featured version of many of the classes. For example, we would implement a version of the parameterized classes which restricts the type parameter to data type pointers and removes the use of references. We believe that providing the simple version of classes would satisfy many users who would otherwise alter a COOL class implementation to reduce its complexity for space/time considerations. In addition, it would simplify the design for the COOL classes making them easier to use, and allowing for more code reuse.

We like the "forest" hierarchy structure and believe that it is more appropriate for applications written in C++ than the Smalltalk deeply nested structure found in other class libraries, such as Gorlen's NIH class library [8]. In an effort to reduce even further the complexity and size of an application that uses the class library, we would also opt for placing every non-inline member function in its own source file and the resulting object file in the archive. This would force linkers to link only those member functions actually utilized in an application into the executable image. Finally, we are concerned about the single namespace and the inevitable name clashes that result when two independently developed class libraries are combined in a single application. In one case we are familiar with, a user trying to use COOL with the Stanford InterViews class library [10] had to make several changes to class names and functions that were common to both libraries. It appears that with the current language definition, the only reasonable solution is to require prefixing all global names with a two or three letter prefix in order to reduce the chance of a clash. This, however, is clearly not acceptable and we believe a language extension to isolate namespaces is the only viable long-term solution.

6. Conclusion

The COOL project has been an exciting and rewarding experience, serving not only to fulfill our initial intent of providing valuable experience with C++, but also as a focal point for a larger discussion within the company regarding software productivity and code reuse. Our diverse user-group has provided valuable information concerning applicability and potential for utilization of C++ class libraries in a variety of projects and platforms. We conclude that a class library consisting of many basic data structures and templates:

- can significantly aid the portability of an application
- will be viewed with considerable skepticism by many C programmers
- must be supplied in source code format
- will often be modified/decomposed to suit the purpose
- provides a medium for the dissemination and spread of ideas and techniques

While most applications require many of the basic data structures found in COOL, there are always other necessary components. It is our conjecture (based on a limited application test set), that many projects will actually require four types of class components:

- basic data structures -- strings, templates, containers, etc.
- user-interface widgets -- menus, buttons, dialog boxes, etc.
- network/communication -- file transfer, TCP/IP, remote access, etc.
- application-specific -- domain-specific "high value" objects

If there is one significant lesson we have learned from COOL, it is that W.C. Fields was right

in saying: "You can't satisfy all the people all the time!" On the other hand, a C++ data structure class library organized in a "forest" hierarchy has components that can, in combination with other libraries, satisfy most of the people most of the time. We think this is about the best that is possible.

7. Status

COOL is currently running on a Sun SPARCstation™ 1 running SunOS™ 4.x, and a PS/2™ model 70 running OS/2™ 1.2. The SPARC port utilizes the AT&T C++ translator (cfront) and the OS/2 port utilizes the Glockenspiel C++ translator (which is a port of the AT&T translator) with the Microsoft C compiler.

8. Acknowledgements

Many people contributed ideas, suggestions, and criticisms that have helped shape the class library evolution and development. Amongst these are Fred Burke, Terry Caudill, Merrill Cornish, Carey Jung, Asif Malik, Dane Meyer, LaMott Oren, Jeri Steele, Dan Stenger, and Brian Victor.

9. References

[1] James Coggins, *Design Criteria for C++ Libraries*, USENIX C++ Conference, San Francisco, CA, April 1990.

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

[3] Margaret Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual*, Addison Wesley, 1990.

[4] 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 April 1990.

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

[6] Mary Fontana, Martin Neath and Lamott Oren, *A Runtime Type Checking and Query Mechanism for C++*, Information Technology Group, Austin, TX, Internal Original Issue November 1990.

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

[8] Keith Gorlen, *An Object-Oriented Class Library for C++*, USENIX C++ Workshop, Santa Fe, NM, November 1987.

[9] Andrew Koenig and Bjarne Stroustrup, *Exception Handling for C++*, Submitted as document X3J16/90-042 to the ANSI C++ committee, July, 1990.

[10] Mark A. Linton, Paul R. Calder, and John M. Vlissides, *InterViews: A C++ Graphical Interface Toolkit*, Technical Report CSL-TR-88-358, Stanford University, July 1988.

---

[11] Mike Miller, *Exception Handling Without Language Extensions,* Proceedings of the USENIX C++ Conference, Denver, CO, October 17-21, 1988, pp. 327-341.

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

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