The Design of the C++ Booch Components

Grady Booch
Wizard Software
2171 South Parfet Court Lakewood, CO 80227
gbooch@ajpo.sei.cmu.edu
(303)986-2405

Michael Vilot
ObjectWare, Inc.
16 Warton Road Nashua, NH 03062
mjv@objects.mv.com
(603)888-4729

ABSTRACT

This paper describes design issues encountered developing a reusable component library. The design applied encapsulation, inheritance, composition, and type parameterization. The implementation uses various C++ mechanisms, including: virtual and static member functions, templates, and exceptions.

The resulting library contains about 500 components (mostly template classes and functions) and an optional utility for instantiating templates. The components provide variations of basic collection/container abstractions with various time and space complexities.

A key insight gained from this project: the design process centered on developing a "template for the templates" — designing a component framework and orderly process for generating the template classes.

1 Introduction

The purpose of this project was to translate "The Ada Booch Components"® to C++ (going from an object-based to an object-oriented implementation). It is worth noting that this was not a research effort. The resulting library is a software product intended for use in commercial settings.

This project provided an opportunity to investigate both the OOD method [Booch90] and the C++ language [ATT89] in the context of designing reusable component libraries. We learned an important lesson about designing with multiple class lattices: the tradeoffs of various inheritance structures turned out to be a fundamental design issue.

The library focuses on providing a number of core data structures. A key feature of the library is that each data structure is implemented several ways (each exhibiting different, but well-defined time/space complexity). This provides the library user with the flexibility to select the most appropriate implementation. Another distinctive feature is support for concurrent access to the library.

As part of the implementation in C++, we provide mechanisms to support the key abstractions (including utilities for an early implementation of the template facility and scaffolding for the exception handling proposed for the language).

1.1 Goals of the Project

The main goal of the library is to provide a carefully designed collection of useful data structures. The implementation in C++ included four additional goals:

Efficiency: Our goal was to provide easily assembled components (efficient in compilation resources), which impose minimal runtime and memory overhead (efficient in execution

resources), and are more reliable than hand-built mechanisms (efficient in developer resources).

Ease of Use: A clear and consistent organization should make it easy to identify and select appropriate forms of data structures. Providing nearly independent parts simplifies combining components to create instantiations.

Extensibility: It should be possible to add new data structures and new storage mechanisms. Using inheritance, these changes should be expressible as incremental extensions.

Adaptability: The library's environment-specific aspects should be clearly identified, and packaged in a way that local substitutions for these aspects are possible.

1.2 Scope of the Project

Our primary concentration was on data structures similar to the "Collections" of the NIH class library [Gorlen87] or the "Containers" of `libg++` [Lea88].

The library also has some notable differences: Unlike NIH, we did not wish to emulate the Smalltalk environment. Therefore, we do not explicitly address nor model "metaclasses." Unlike `libg++`, we did not wish to provide duplicate, alternative versions of "standard" C++ libraries (for example, the `streams` classes). Unlike InterViews [Linton87], or Iris [Gansner88], we did not provide classes for windows-based libraries. We consider this an advantage — clients can easily combine the two kinds of libraries within the same application.

A notable feature of this library is its support for concurrency. We provide implementations of data structures that perform correctly in the presence of both multi-threaded and multi-process use. This reflects our experience on larger projects, where multiple processes are routinely applied to meet performance and/or distribution requirements. It also reflects the growing trend toward multiprocessing architectures, operating systems, and applications.

1.3 Overview of the Paper

The rest of this paper discusses the important design issues we encountered. Section 2 introduces the Booch components library. Section 3 discusses the key abstractions we found important in shaping the design. Section 4 discusses the mechanisms used in the implementation of those abstractions — it highlights the C++ language features which proved effective supporting the mechanisms.

2 Background — The Booch Components

[Booch87] describes the original Ada components, which were implemented as 501 packages and subprograms. Figure 1 provides a top-level class diagram,[1] and shows that the C++ library contains five class categories:

- Core Data Structures
  members include `List`, `Set`, `Bag`, etc.
- Storage Managers
  members include Bounded, Unbounded, Managed, Controlled
- Concurrency Managers
  members include `Semaphore`, `Monitor`
- Exceptions
  members include `Overflow`, `Underflow`, `Null`, `Position_Error`, `Unbound`, etc.
- Tools
  members include pipes, filters, sorts, searches, etc. and class utilities

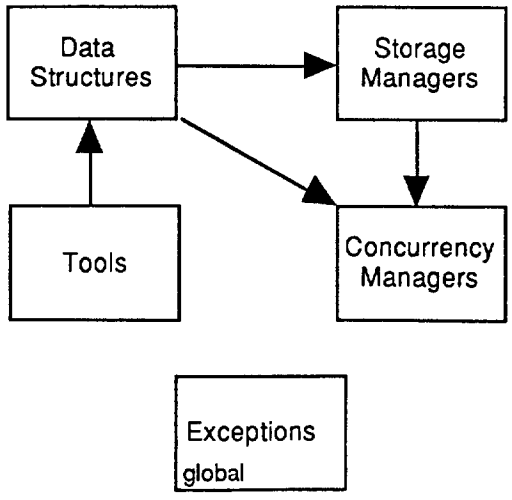

Figure 1
Booch Components Class Categories

[1] The boxes represent groups of related classes, while the arrows represent uses relations.

Data Structures	Storage Managers			
	Bounded	Unmanaged	Managed	Controlled
Stack	S,G,C,M	S,G,C,M	S,G,C,M	S
List	S	S	S	S
Double List	S	S	S	S
String	S,G,C,M	S,G,C,M	S,G,C,M	S
Queue	S,G,C,M	S,G,C,M	S,G,C,M	S
Priority Queue	S,G,C,M	S,G,C,M	S,G,C,M	S
Balking Queue	S,G,C,M	S,G,C,M	S,G,C,M	S
Balking Priority Queue	S,G,C,M	S,G,C,M	S,G,C,M	S
Deque	S,G,C,M	S,G,C,M	S,G,C,M	S
Priority Deque	S,G,C,M	S,G,C,M	S,G,C,M	S
Balking Deque	S,G,C,M	S,G,C,M	S,G,C,M	S
Balking Priority Deque	S,G,C,M	S,G,C,M	S,G,C,M	S
Ring	S,G,C,M	S,G,C,M	S,G,C,M	S
Map	S,G,C,M	S,G,C,M	S,G,C,M	S
Cached Map	S,G,C,M	S,G,C,M	S,G,C,M	S
Discrete Map	S,G,C,M	S,G,C,M	S,G,C,M	S
Set	S,G,C,M	S,G,C,M	S,G,C,M	S
Discrete Set	S,G,C,M	S,G,C,M	S,G,C,M	S
Bag	S,G,C,M	S,G,C,M	S,G,C,M	S
Discrete Bag	S,G,C,M	S,G,C,M	S,G,C,M	S
Binary Tree (single)	S	S	S	S
Arbitrary Tree (single)	S	S	S	S
Binary Tree (double)	S	S	S	S
Arbitrary Tree (double)	S	S	S	S
Graph	S	S	S	S
Undirected Graph	S	S	S	S

Table 1
Core Data Structures Class Category

The Exceptions class category is global to all others, since they are the library's single error reporting mechanism.

The Data Structures are the core of the library. Note that each data structure is *not* a class, but rather a *template* for a class (parameterized by type). The data structures combine with various storage and concurrency management ***forms*** into components.

Table 1 summarizes these components: the columns indicate the various combinations of data structures with storage managers. Each entry has letter(s) denoting the concurrency manager(s). For example, the Controlled forms only require Sequential implementations — the others have more entries.

Tools are clients of the components defined in the library. Many of these are parameterized functions. They are layered upon the core data structures to provide useful services to library users. Table 2 lists the tools provided. Note that

Utilities	Filters & Pipes	Sorting	Searching	Pattern Matching
character	input	straight insertion	sequential	simple
string	output	binary insertion	ordered sequential	Knuth Morris Pratt
integer	translate	shell	binary	Boyer Moore
floating point	expand	bubble	list	regular expression
calendar	compress	shaker	binary tree	
		quick	arbitrary tree	
list (single)	bounded pipe	radix	graph (bounded)	
list (double)	unmanaged pipe	straight selection	graph (unmanaged)	
binary tree (single)	managed pipe	heap	graph (managed)	
binary tree (double)		natural merge	graph (controlled)	
arbitrary tree (single)		polyphase		
arbitrary tree (double)		topological (bounded)		
graph		topological (unmanaged)		
undirected graph		topological (managed)		
		topological (controlled)		

Table 2
Tools Class Category

several of the Utilities classes directly support classes in the Data Structures class category.

Each core data structure primarily describes behavior. To provide the variety of time/space performance, each data structure uses different representations. The choice of representation implies different Storage Management implementations.

The Storage Manager classes describe strategies for managing storage for a given choice of representation:

- Bounded storage management preallocates a fixed amount of storage (trading space flexibility for time efficiency). The other storage managers all deal with unbounded storage.
- Unmanaged forms disregard the details of storage management.
- Managed forms are careful to allocate and deallocate each `Node` from a free list.
- Controlled forms support concurrent storage management.

Managed and Controlled forms use the Storage Managers. The key to these forms is active management of free lists of `Nodes`. The Bounded and Unmanaged forms do not use storage managers. Bounded forms do not create garbage, and using an Unmanaged form relies on the primitive `new` and `delete` operations.

The Guarded, Concurrent, and Multiple forms of the components provide varying degrees of 'manual' and 'automatic' concurrency control (Multiple forms provide for multiple readers and single writers).

The key thing to notice about the organization of the core data structures is the regular structure. Each entry in Table 1 is an instance of a tuple of the form:

<Data Structure, Storage Mgr, Concurrency Mgr>

These define the ***key abstractions*** of the library. The main design activity was devising appropriate ***mechanisms*** to reflect clearly this organization, and to provide convenient and effective ways for clients to combine and use selected components.

3 Design — Key Abstractions

In the design of the library's key abstractions, we addressed four fundamental issues:

- How to use inheritance
- Whether and how to use parameterized types
- How to provide a uniform and flexible error reporting
- How to provide concurrent forms

3.1 Using Inheritance

A drawback to the Ada design was redundancy: significant portions of the library had similar, but slightly different implementations. This made it an obvious candidate for implementation in an object-oriented programming language.

The use of inheritance was the most significant change in going from an object-based design to an object-oriented one. The main issue was whether or not to be "completely object-oriented" or just apply it where it proved to be useful and effective. We chose a "forest" approach over a "tree" approach for three reasons:

- it accurately reflected the regular structure of the component forms
- it involved less complexity and overhead when selecting one
- it avoided the endless ontological debates engendered by a "completely object-oriented" approach.[2]

We found the following uses for inheritance:

- layering abstractions and code reuse
- polymorphism
- design structure
- implementing (some) composition relationships

Layering abstractions: [Booch87] (Section 11.3) discusses the implementation of layers of abstractions, building on the most primitive classes.

---

[2]For example, is a `Binary_Tree` a special case of an arbitrary `Tree`, or is an arbitrary `Tree` a kind of tree with more than the default number of `Nodes`? There are advantages to structuring the hierarchy either way.

For example, a `Priority_Queue` or a `Balking_Queue` can be built from a base `Queue` class. Figure 2 is an OOD Class diagram which illustrates how we were able to use inheritance to express this notion.[3]

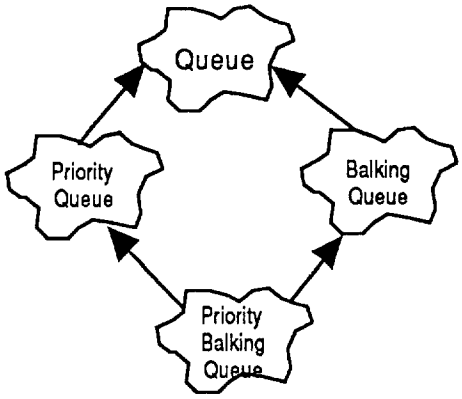

Figure 2
Inheritance and Layered Abstractions

Polymorphism: (the "usual" use of inheritance) is more useful in *contents*, especially in passive iterators. We actually faced a dilemma, trying to decide whether the data structures should store items by reference or by value.

In the end, we decided to leave the decision in the hands of the library's clients. The templates appear to pass items by value. However, simply instantiating them with reference or pointer arguments allows clients to use the complementary scheme.

Capturing design intent: We found "abstract classes" useful. Clients cannot create instances of abstract classes: their purpose is to express the common aspects of derived classes — providing a way to explicitly record design structuring decisions.

As an example, the three classes `List`, `Tree`, and `Graph` can be abstract bases for the classes `Single_List`, `Double_List`, `Binary_Tree`, `Arbitrary_Tree`, `Undirected_Graph`, and `Directed_Graph` (respectively), since the latter differ mainly in their choice of representation. Figure 3 illustrates the case for the `List` classes.

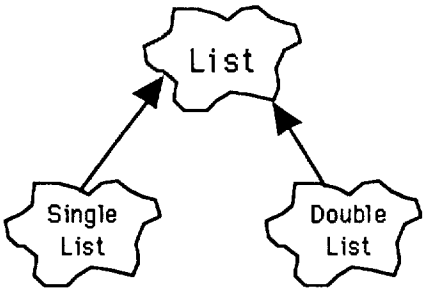

Figure 3
Inheritance and Representation

Composition: We found composition to be as useful as inheritance. For example, a Guarded form *uses* a Semaphore, but is not a *kind of* Semaphore. As another example, a Managed form *uses* a Storage Manager, but is not a *kind of* Storage Manager. Figure 4 illustrates how we structured the combination of core data structures and concurrency classes.[4]

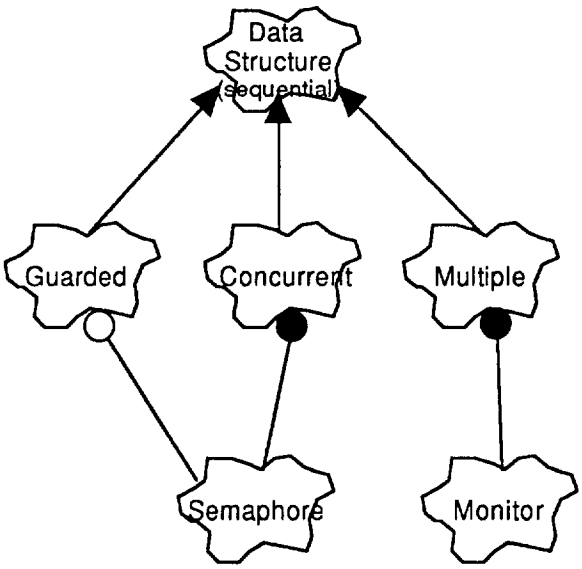

Figure 4
Inheritance and Composition

The concurrent forms are derived from the sequential form, which reflects the design intent (for example, a `Guarded_Queue` is a kind of

[3]The arrows represent the inherits relationship, pointing from the derived to the base classes involved

[4]The lines with circles represent the uses relationship (with the circle a the client's end). An unfilled circle indicates the relationship is visible in the client's declaration, a filled circle indicates it is private to the client's definition.

Queue that supports concurrent access). In this particular case, inheritance also achieves code sharing, because the concurrent forms are incremental extensions of the sequential forms.[5]

3.2 Using Type Parameterization

Inheritance is certainly the most visible (and popular) aspect of object-oriented design. But it is not the only structuring principle. Type parameterization is central to the design of a simple and effective scheme for combining data structures with various storage managers.

Depending on a mechanism supporting this abstraction was essentially a question of timing (of C++ language release). In the end, we decided to provide an early implementation of the "template" mechanism [Stroustrup88] as an interim measure.

Figure 4 shows the recursive application of the instantiates relationship. These are simply different binding times: classes are instantiated at compile time, objects are created at run time.

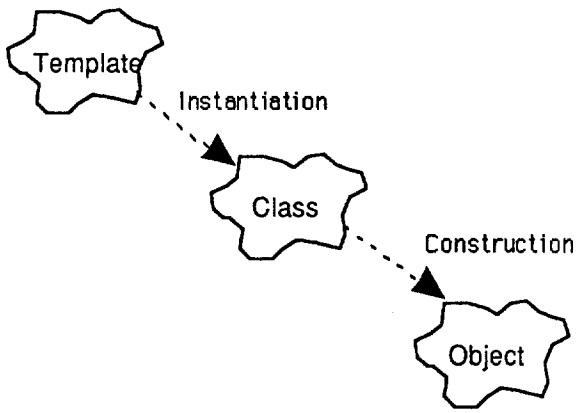

Figure 4
Templates → Classes → Objects

This approach eliminates a level of indirection by providing the opportunity to bind certain decisions at compile time rather than at run time. Template arguments are per-class variables (such as the choice of Storage Manager), while constructor arguments are per-object variables. For example, library clients can choose to create a second kind of Bounded storage management which is stack-based. [Stroustrup86] (Section 5.5.8) discusses the strategy we used in the library's default heap-based Bounded form: objects have part of their state allocated in the heap (the maximum size is an argument to the constructor).

Using template parameters, clients could extend the library to provide completely stack-based objects which do not use free store management. The cost is that the size is fixed for all objects of the given class (due to the earlier binding time). However, since clients can easily make different instantiations with different sizes, it is not any harder to produce a variety of forms.

Clients can also avoid the overhead of indirect function calls in cases where such performance issues are important. This exploits the difference between in-line expansion during template compilation, and passing function pointers in constructors during object creation.

3.3 Error Reporting

When using library functions, there are basically two categories of errors:

- non-fatal usage errors
- fatal resource and/or corruption errors

For non-fatal errors, each data structure provides ***selector*** member functions which report the state of a class instance. Clients can call these functions to detect invalid conditions before invoking the ***modifier*** function which would report an error.

However, this is not sufficient to handle other kinds of errors. We considered the following issues when designing the error abstractions:

- the type and amount of information required to describe an error
- client-provided definitions of errors and what to do about them
- communication between the library and its environment

[5] Also, note that abstractions use storage managers, and that storage managers use abstractions (especially Lists). See [Booch87], p. 427, for a comment on this circularity. At the base of this recursion is an abstraction which uses the built-in storage management provided by the language and its runtime environment.

Exceptions are an effective mechanism for communicating problems from the library to the environment. We adopted a variation of the exception reporting mechanism of [Stroustrup90] as the library's error reporting mechanism.

One aspect of exceptions in C++ not present in Ada is the opportunity to combine them with inheritance. This provides a natural classification mechanism (and the corresponding extensibility and flexibility to clients).

We generalized the notion of exceptions, and separated them from the mechanisms involved in reporting them. In this library, we provide classes describing exceptions, and a simple mechanism for creating instances and reporting them to library users. This approach is simple, extensible, and adaptable to the proposed C++ language mechanism.[6]

3.4 Concurrent Forms

A purely sequential version of the library could be made to work in a concurrent environment (for example, client applications which manually guard every call to any library operation). The Guarded forms support this approach.

However, besides being tedious and error-prone, such an approach can be inefficient. Certain operations (especially in complex components) are more effective when implemented at the finer granularity available to the library's implementor. The Concurrent, Multiple, and Controlled forms provide the required flexibility.

An important part of the original library is support for concurrent uses of the data structures. In Ada, it is possible to implement these forms in a portable manner: concurrency is part of the language. When designing the C++ version, we made the following assumption: those environments that care will have ported or implemented at least a `Semaphore` class. Others will not care and will appreciate not having to pay any overhead.

[6] A key advantage of the proposed addition of exceptions to C++ is their integration into the type system. We see this as a significant advantage over Ada's exceptions.

4 Implementation — Mechanisms in C++

C++ provides many features that are helpful to those implementing software intended for reuse. Indeed, much recent activity in evolving the language has been specifically geared towards supporting library development. Our design relied on several important abstraction concepts: encapsulation, inheritance, composition, type parameterization, exceptions, and concurrency.

We found the following mechanisms to be most significant from a design perspective:

- classes (especially inheritance)
- templates (type parameterization)
- exceptions (as a uniform error reporting mechanism)
- the task library (as a concurrency control mechanism)

The existing (2.1) version of C++ provides mechanisms supporting the first concept, future versions will support the next two, and an existing library supports the last.

4.1 Classes

The library's design relies heavily on four aspects of C++'s class mechanism:

- Access specifications (including friends)
- Constructors and destructors
- Class-specific operators `new` and `delete`
- Inheritance (including private derivation and multiple inheritance)

Access and friends: The C++ `friend` mechanism is useful as a way to record design decisions regarding visibility and access control. The library provides both "active" and "passive" iterators (using the terms of [Booch87]).[7] The iterators are `friend` classes, as in Sections 6.8 and 7.3 of [Stroustrup86]. Passive iterators take the form of ***applicators***, after the fashion of Section 4.5 of [Dewhurst89].

[7] It is too early to tell which form clients will prefer — the passive form adheres to the original Ada design, while the active form is popular in the C++ community.

Constructors and destructors provide the obvious hooks for the Managed forms to do their work. The various approaches to storage management discussed in [Stroustrup86] are all applicable. An important feature added in 2.0 C++ is the ability to redefine the `new` and `delete` operators for certain classes

New and Delete: The C++ semantics of class-specific operators `new` and `delete` are as static member functions. There is one storage manager per class — providing one free list per class (instantiation) rather than one per container object.[8] This exactly matched our needs for the Managed and Controlled forms.

The library provides various default kinds of `Nodes` . For Managed forms, the default `Node` classes use Sequential storage management. For Controlled forms, the default `Node` classes use a Concurrent storage manager.

The data structure classes simply invoke `new` and `delete` operations as required. This turned out to be far simpler than the original Ada design, which required a completely different syntax when using the library's Storage Manager instead of the predefined free store operators.

Inheritance: C++ provides a unique feature for turning services (the uses relationship) into sub-objects ("contains," the part of relationship): it is possible to use private derivation (a kind of inheritance) as a way to implement a uses relationship. Private derivation ensures that the implicit conversion rules (that is, using a derived object as a base object) do not apply, but make the private base object a part of the object.

We used public derivation to express the relationships between the data structures. We also used private derivation to implement using relationships between the data structures and the storage manages. Thus, we found multiple inheritance to be a straightforward and effective way to realize the design.

4.2 Templates

Because type parameterization is central to the design of the library, Ada's **generic** facility was widely used in the original version. We considered using the macro pre-processing approach to implementing parameterized types. The apparent advantage of using them (portability) is deceptive — there are significant variations between "standard" preprocessors. While the various forms of "name pasting" and "stringitization" can be accommodated, the restrictions on macro size and (generated) line length proved to be too troublesome.[9]

We decided, instead, to built a filter which accepted the proposed C++ template mechanism. There are two parts to the mechanism: instantiating a new class declaration, which happens upon encountering a declarative use; and instantiating the class definition, which is done by hand. Figure 11 illustrates how the template utility fits into the compilation environment.

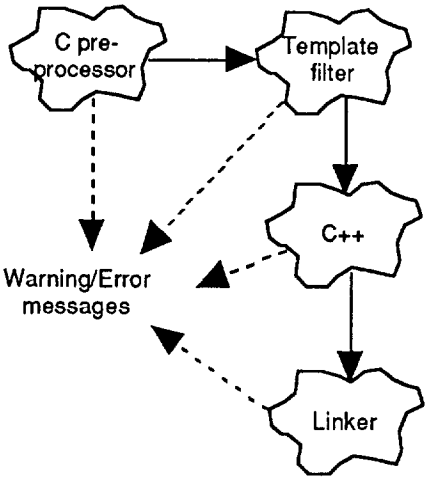

Figure 11
Template Filter and Environment

Using such a filter allows the resulting output to match the multiple-line format of the input. It is also possible to provide scoping of template argument names. This significantly improves error message clarity and source-level debugging.

[8] Also possible: one pool of `Nodes` shared by several classes. Clients simply instantiate them with the same `Node` class which has definitions of `new` and `delete` providing common free list management.

[9] Using preprocessor macros, the resulting output must fit on one line — large macros generate long lines. These long lines can exceed the capacity of underlying tools (most notably, the file system).

4.3 Exceptions

It turns out to be considerably easier to *use* the exception mechanism proposed for C++ than to actually *implement* one outside the language (see, for example, [Miller88]), This is because error reporting (in the library) is easier than error handling (in the client application). Therefore, we do not implement an exception handling mechanism in the library. Instead, we simulate a *throw-expression* with calls to a `_catch()` function.[10]

`Exception` objects record the operation that caused them and the condition violated. The default `_catch()` action is to report the `Exception`'s parameters and then to invoke the library function `terminate()` (whose default action is to exit the process). Clients can provide an alternative action by invoking the `set_terminate()` library function.

This approach achieves the goal of error notification, but does not pretend to provide robust error handling (that is, it does not unwind the stack and invoke the "appropriate" destructors automatically). Our design is compatible with a language implementation: using the same (or similar) syntax, it preserves evolution to future language implementations.

4.4 Tasking

Although concurrency is not part of the language, it is available in library form [Stroustrup87] [Beck90]. Its use is optional, built as a straightforward extension to the sequential forms of the components.

We assume clients using this library in C++ environments supporting concurrency will follow an approach similar to the AT&T task library. Our design assumes an implementation of a `Semaphore` class as described in [Shopiro87]. Any similar class with `wait()` and `signal()` members can be substituted.

Note that we do not implement tasking as part of this library (the default `Semaphore` class is simply a place holder for the linker). We decided to structure the concurrent forms as an optional, layered part of library, and to rely on local implementations of concurrency mechanisms. The classes `Semaphore` and `Monitor` provide the connection to the local mechanism.

Our design provides a simple way to incorporate the components into an environment which provides concurrency. If the environment supplies an implementation that supports concurrency, then the library can take full advantage of it.

While reviewing the exception handling proposal in [Stroustrup89], we noticed the 'resource acquisition is initialization' model is an ideal match to the needs of the library's Concurrent and Multiple forms. Building on the `Semaphore` and `Monitor` classes, we added the classes `Lock`, `Read_Lock`; and `Write_Lock`. Their constructors and destructors provide a simple and reliable mechanism for correctly implementing concurrency semantics — even in the presence of exceptions.

5 Summary

The design of this library focused on:

- external interface (behavior) of the key abstractions
- ways to structure and combine forms into templates
- effective implementation (C++ language features) of the mechanisms (made easier with the addition of specific support)

The central design activity was careful consideration of the basic "building block" components (the core data structures, the storage managers). The key insight was designing a simple way of composing them into the various templates in the library.

---

[10] `cfront` reserves the `catch` keyword with a "Sorry, not implemented" message, denying us the most obvious candidate function name. Note that this is simply a modified form of the approach used in Section 7.3.4 of [Stroustrup86]. However, instead of using arbitrary names for the functions, we use the name of the mechanism that will eventually appear in the language.

5.1 Results

We feel this project met all of its goals:

Efficiency: Using a component from this library incurs little overhead beyond "hand coded" versions — increasing the likelihood of reuse. Purely sequential environments and applications pay no overhead (time or space) for not using the concurrent forms — they are strictly optional.

Ease of Use: The time/space variety of the key data structures provides a clear organization for the library. The template mechanism provides a convenient way to combine forms.

Extensibility: The library is extensible through the application of inheritance. The design for this library achieves an appealing orthogonality — it is possible to extend the library with new data structures, new storage managers, or both.

Adaptability: The library is adaptable through explicit connections to the (user-controlled) environment:

- underlying (global) storage management
- compilation environment (template expansion)
- default implementation for exceptions and concurrency

The storage management approach is extensible with new kinds of `Nodes`, independent of extensions to data structure classes. For example, some clients may wish to extend this library to accommodate persistent storage in databases.

The library's template utility program can become a routine part of C++ compilation, run by hand as a separate program, or avoided entirely (in those environments that already support templates).[11]

Library clients can control the basic error handling mechanism: the library provides default versions of the `_catch()` function (taking a reference to an `Exception` object). It is possible to replace this with functions to call, for example, `longjmp()`.[12] C++ environments directly supporting *throw-clauses* can omit the `_catch()` functions entirely.

Allowing local tailoring also meets the special needs of specific environments that have already invested in mechanisms similar to the ones we used (for example, concurrent runtime libraries on multiprocessor platforms).

5.2 Conclusions

We found that the concepts of inheritance, composition, and type parameterization were equally valuable aids in structuring the design. From a design perspective, inheritance is as useful for expressing existing commonality as for achieving code reuse. It is also important as a way to "design in" extensibility.

It is nearly impossible to design a useful library of collection/container data structures without using type parameterization. Implementing the template scheme proposed for C++ seems to have been a reasonable investment for the resulting leverage gained for this library.

C++ proved to be an effective language for implementing an object-oriented design. The existing and planned features of C++ (especially templates and exceptions) are indeed useful for library design. By conforming to the existing (and proposed) "standard C++" techniques, and adding a little support for features that are not yet generally available, we were able to implement our design in C++.

The resulting library components provide a variety of important data structures which users can apply to many different uses. The variety and extensibility of the storage management, concurrency control, and error reporting mechanisms provide a great deal of flexibility. We expect these components to be as applicable to small, embedded applications as they are to large, distributed, multi-person projects.

---

[11] Another advantage to discarding the template utility is that, with proper template expansion built into the compiler, the programmer need never manipulate files containing the generated source text.

[12] Assuming that such clients also have solutions to the stack unwinding/destructor invocation issues involved.

REFERENCES

[ATT89] C++ Language System Release 2.0 Product Reference Manual, AT&T Select Code 307-146, June 1989.

[Beck90] Beck, B., "Shared-Memory Parallel Programming in C++," *IEEE Software*, 7(4), July 1990.

[Boehm88] Boehm, H.-J., and Weiser, M. "Garbage Collection in an Uncooperative Environment," *Software — Practice and Experience*, 18(9): pp. 807-820, September 1988.

[Booch87] Booch, G., *Software Components with Ada*, Benjamin/Cummings, Reading MA, 1987.

[Booch90] Booch, G., *Object Oriented Design with Applications*, Benjamin/Cummings, Reading MA, 1990 .

[Gansner88] Gansner, E.R., "Iris: A Class-Based Window Library," *C++ Conference*, USENIX Association, Denver CO, October 1988.

[Gorlen87] Gorlen, K., "An Object-Oriented Class Library for C++," *C++ Workshop*, USENIX Association, Santa Fe NM, November 1987.

[Lea88] Lea, D., "libg++, the GNU C++ Library," *C++ Conference*, USENIX Association, Denver CO, October 1988.

[Miller88] Miller, W.M., "Exception Handling Without Language Extensions," *C++ Conference*, USENIX Association, Denver CO, October 1988.

[Shopiro87] Shopiro, J., "Extending the C++ Task System for Real-Time Control," *C++ Workshop*, USENIX Association, Santa Fe NM, November 1987.

[Stroustrup86] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading MA, 1986.

[Stroustrup87] Stroustrup, B., and Shopiro, J., "A Set of C++ Classes for Co-routine Style Programming," *C++ Workshop*, USENIX Association, Santa Fe NM, November 1987.

[Stroustrup88] Stroustrup, B., "Parameterized Types for C++," *C++ Conference*, USENIX Association, Denver CO, October 1988.

[Stroustrup89] Stroustrup, B., "Exception Handling for C++," *C++ at Work*, Tyngsboro MA, November 1989.

[Stroustrup90] Stroustrup, B., "Exception Handling for C++ (revised)," *C++ Conference*, USENIX Association, San Fransico CA, April 1990.