The Features of the Object-oriented Abstract Type Hierarchy (OATH)

Brian M. Kennedy
bmk@csc.ti.com

Computer Systems Laboratory
Computer Science Center
Texas Instruments

ABSTRACT

The Object-oriented Abstract Type Hierarchy (OATH) instantiates an approach to C++ class hierarchy design that exploits subtyping polymorphism, provides greater implementation independence, and supports implicit memory management of its objects. The primary design goal of OATH was to provide an abstract type hierarchy that is consistent with the concepts being modelled by utilizing a strict subtyping approach to hierarchy design. This approach increases the polymorphism and implementation independence of code that uses OATH. The second major design goal was to provide robust garbage collection of OATH objects, fully implemented within a portable C++ class library. OATH is implemented via parallel hierarchies of internal types and "accessors". Although similar to the infamous "smart pointers", OATH accessors do not suffer many of the same problems. In particular, OATH accessors never release "dumb pointers" to the environment and fully support hierarchical argument matching. OATH accessors also offer the opportunity to "leaf" implementation classes to bypass the virtual mechanism for efficiency, when generality is not needed.

The Object-oriented Abstract Type Hierarchy (OATH) instantiates an approach to C++ class hierarchy design that exploits subtyping polymorphism, provides greater implementation independence, and supports implicit memory management of its objects. Although the core OATH library provides numerous basic types and data structures, it is the features of the hierarchy design that are most valuable to the user and developer of application-specific OATH classes. These features and their implementation are the focus of this paper.

Throughout this paper, numeric types and basic container types are used as examples. These types have the advantage that they are simple and well-understood, so no prior explanation must be offered. However, the benefits provided by OATH, and by object-oriented programming in general, are only fully realized when applied to more complex (usually application-specific) problems.

1 THE FEATURES OF OATH

The primary design goal of OATH was to provide an abstract type hierarchy that is consistent with the concepts being modelled by utilizing a strict subtyping approach to hierarchy design. This approach increases the polymorphism and implementation independence of code that uses OATH.

The second major design goal was to provide robust garbage collection of OATH objects, fully implemented within a portable C++ class library. Explicit management of dynamic memory in C++ is a burdensome and error-prone task. C++, unlike CLOS and Smalltalk, does not force the overhead of

garbage collection on all programs (many of which do not need it); however, C++ does provide sufficient functionality that garbage collection can be provided in a library, so that applications that need it can have it.

OATH also provides heterogeneous container classes. Heterogeneous containers are more general and flexible than homogeneous containers, and often more natural to use. Such generality, however, requires the ability to determine the type of an object after it is removed from a container.

In C++, heterogeneity is often obtained via void*, which can point to any type and can later be cast to the appropriate type. However, such casts essentially abort the C++ type system and are thus very error-prone. The programmer must enforce some policy to ensure that objects are cast to an appropriate type.

OATH provides dynamic type determination in the form of "safe casts". A "safe cast" from an OATH type to a more derived type returns the object if it is truly of the derived type, or Nil if it is not. Nil is also a useful feature on its own. Nil can be assigned to an accessor of any type, but is itself a "non-object". Nil is similar in concept to the null pointer in C++.

2 SUBTYPING

The OATH hierarchy was designed to reflect the subtyping relationships between the types that it represents. The use of C++ inheritance for subtyping was strictly separated from implementation and code reuse. This approach to hierarchy design provides greater implementation independence, for both code inside the library and code that uses the library. The hierarchy also allows greater exploitation of subtyping polymorphism.

2.1 Subtyping v. Code Reuse

C++ classes are used to define both an abstract type (the functionality of an object) and an object implementation (the internal structure of an object). Similarly, inheritance in C++ is used for both subtyping (inheriting functionality) and code reuse (inheriting implementation). Although these two features are provided in C++ with the same mechanism, they are distinctly different concepts [Amer90].

Code reuse is a powerful feature of C++; however, it is a poor basis for object-oriented design. A type hierarchy should be designed to reflect the behavior of the objects being modelled. It should not be designed to reflect the most convenient computer representation of the objects.

For example, consider `rational` and `integer`, common multi-precision numeric types. Many class libraries have been proposed and/or implemented (e.g. Smalltalk [Gold83], NIH OOPS [Gorl87], libg++ [Lea88]) that define `rational` as a sibling class of `integer` implemented as a pair of `integers`. Such a definition is simple, exploiting well the power of code reuse via composition. Unfortunately, this definition does not correspond to the mathematical concepts being modelled. Mathematically, all integers are rational numbers -- integer is a subtype of rational. Similarly, all rational numbers are real numbers, and all real numbers are complex -- rational should be a subtype of real which should be a subtype of complex.

The primary goal of OATH is to provide a meaningful abstract type hierarchy: a hierarchy of behavioral specifications that correspond to the concepts being modelled (see Figure 1). Given a consistent abstract type hierarchy, implementation classes (in italics in Figure 1) can be added at the leaves of the hierarchy to implement the behavior of the abstract types (in bold). Code reuse can be exploited at this phase, but should not enter into the design of the abstract type hierarchy.

For instance, in OATH `integer` is derived from `rational`, which is derived from `real`, which is derived from `complex`. These are all abstract types. The type `integer` can be implemented in more than one way. The implementation type `bigInteger` is multiple-precision, whereas `longInteger` is implemented as a `long`. Similarly, `rational` can have several implementations: `bigRational` is a pair of `bigIntegers` and `longRational` is a pair of `longs`. Note that code reuse via composition was exploited without affecting the abstract type hierarchy.

In addition to the conceptual consistency of the hierarchy, this subtyping approach to hierarchy design clearly separates the implementations from the behavioral specifications. This simplifies code maintenance, allows alternate implementations to be added later, and provides a level of implementation independence in code that uses the library. For example, a user of `integer` need not know the implementation. It may be implemented as a `long`, as two `longs`, as an array of `longs`, or as a linked-list of `longs`. Furthermore, another implementation of integer may be added later without rewriting the hierarchy or the code that uses the hierarchy.

2.2 Subtyping Polymorphism

This hierarchy design also provides a great deal of subtyping polymorphism. For example, any code

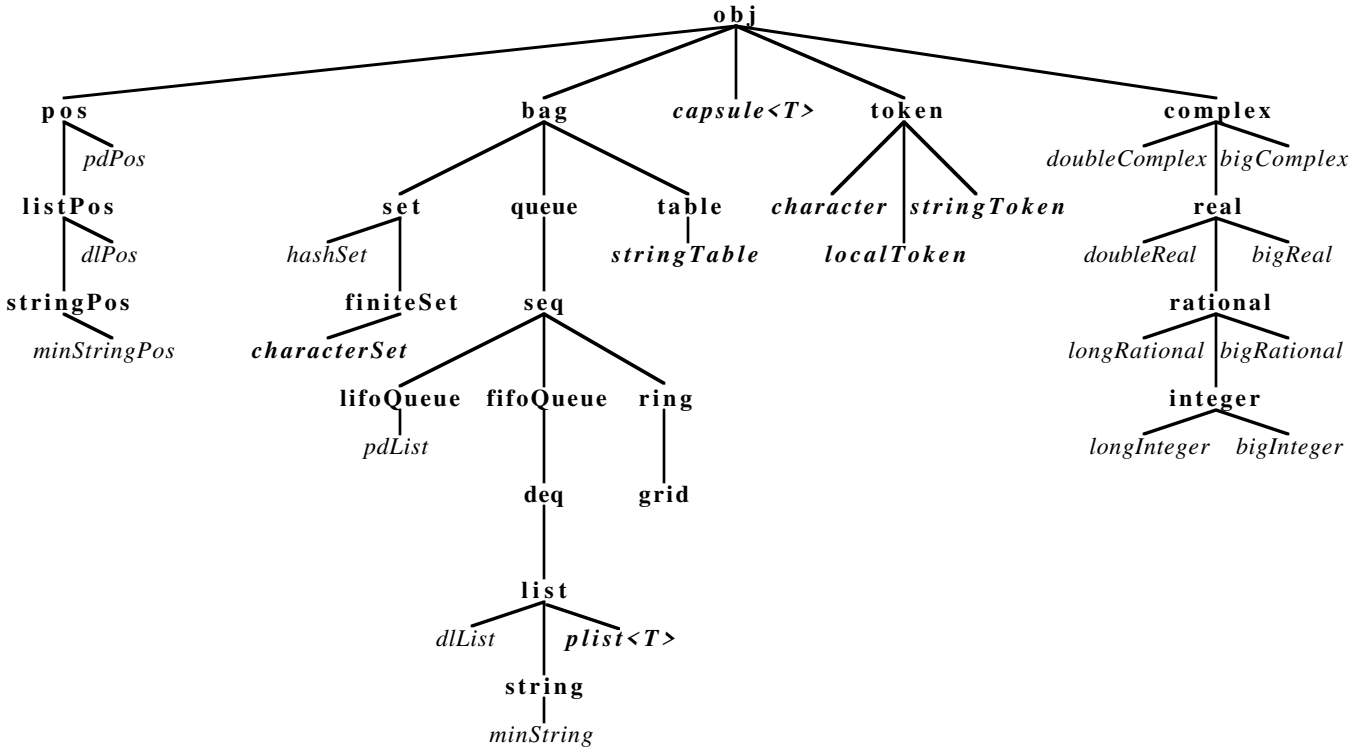

Figure 1. Part of the Object-oriented Abstract Type Hierarchy

written using `complex` numbers will work with `reals`, `rationals`, and `integers`. Similarly, any code written using `bags` will also work with `queues`, `lists`, and `strings`.

For example, consider the common tree search algorithm which records in a structure all possible paths (decisions), and then chooses one. Later it may return to the structure in order to try an alternate path. Separate algorithms could be coded for breadth-first, depth-first, and prioritized searches.

In contrast, a single search algorithm could be coded which is polymorphic on the alternate path structure. A `queue` has sufficient functionality (insertion and extraction) to implement the search algorithm, without knowing exactly what queueing paradigm is used. If this algorithm is passed a `lifoQueue` (last-in first-out queue, or stack), then it will perform a depth-first search. If it is passed a `fifoQueue`, (first-in first-out queue), then it will perform a breadth-first search. If it is passed a prioritized queue, then it will perform a prioritized search.

2.3 Subtyping Domain

For `bags` (OATH container classes), there are two forms of subtyping: increasing functionality and restricting domain. For instance, the OATH `string` is a subtype of `list` that can contain only `characters`. It also has additional functionality, such as <, <=, >, >=, uppercase, lowercase, and hash, which make sense because of `string`'s restricted domain.

`Bag` and its descendants were designed to allow this restriction of domain, or filtering. Any object may be inserted into a `bag`; however, depending upon the subtype of that `bag`, the object may fall through (conceptually as if the bag were a sieve or filter).

This ability to subtype domain can be used to increase polymorphism. For instance, returning to the search algorithm above, a filtering lifoQueue could be passed to the algorithm in order to completely ignore paths that were undesirable. In this way, a completely different search can be made with the same search algorithm.

Although OATH `bags` were designed to support applications that need heterogeneous containers, there are many applications that do not need heterogeneity. Homogeneous containers provide simplicity of expression, increased type safety, and execution efficiency. Homogeneous class implementations can be conveniently provided as subtypes of their heterogeneous counterparts via `parameterized types`. For instance, `plist<complex>` may be an implementation of `list` that holds only `complex` numbers. (Note: this is conceptual -- parameterized types have not yet been used with OATH).

2.4 Subtyping and Execution

This subtyping approach to hierarchy design can effect execution efficiency. Since most functionality is defined as virtual functions, separate from the implementations, out-of-line virtual calls are common when utilizing the generality of the abstract classes. However, the design of the OATH accessors (described below) allows the definition of "leaf" implementations that allow more efficient execution by bypassing the virtual mechanism.

3 OATH ACCESSORS

Users of OATH do not access the objects directly; OATH objects can only be accessed through OATH "accessors". For each OATH type there is a corresponding accessor class. The accessors can be initialized and assigned OATH objects to access (analogous to C++ pointers). However, any other operation on an accessor is applied directly to the abstract object that it accesses (analogous to C++ references). Thus, accessors can be used as if they were the objects themselves, but assigned and passed as function parameters as if they were pointers.

Since the OATH accessor lies between pointers and references, a new but similar syntax would be

nice. For instance, "@" could be used instead of "*" or "&":

```
list@ L;
```

However, OATH is intended to be a C++ library, not a new language. So, the OATH accessor types are suffixed with a capital letter "A":

```
listA L;
```

Analogous to pointers, constructing a `listA` constructs only an accessor, not the list itself. In the declaration above, `L` is initialized to Nil. Given an existing `listA K`, then `L` could be initialized to access the same list object that `K` accesses:

```
listA L = K;
```

Note that both `K` and `L` access the same list. To make a new OATH object, a "make" function must be called. For instance, to initialize `L` to access a copy of the list accessed by `K`, the function `makeCopy` can be used:

```
listA L = K.makeCopy();
```

To make an object from scratch, an implementation type must be chosen and its "make" function, a static member of the accessor class, must be called. For instance,

```
listA L = dlListA::make();
```

makes an empty dlList (doubly-linked list) and assigns it to the list accessor `L`.

In addition to static "make" functions, the accessor types also have a static member function `isa`, which is the OATH "safe cast". For instance, given a `bagA B`,

```
listA L = listA::isa(B);
```

attempts to "safe cast" `B` to a list and assign it to `L`. If `B` is not really a list, then `isa(B)` will return Nil.

3.1 Parallel Hierarchies

OATH is implemented as two parallel hierarchies: the accessor type hierarchy and the internal type hierarchy. The internal types contain the object representation (the data members) and the virtual functions. The accessor types contain all of the externally accessible functions of the abstract types. These functions often do little more than call the appropriate virtual function(s) in the internals hierarchy. The accessors have a single data member, a pointer to an object in the internals hierarchy. Thus, accessors are one-word structures which can be held and passed in registers, and otherwise optimized like the built-in pointer types.

Efficient use of the accessors is natural. For instance, there is no significant cost in passing or returning accessors by value. Further, construction and destruction is equivalent in cost to re-assignment, so placing an accessor in a loop does not have hidden overhead.

```
stringPosA P = S.makePos();
for(; P(); ++P)
   {characterA C = *P;
    // do something with C
   }
```

Unlike many multi-word C++ classes, there is no extra cost in placing the `characterA` declaration within the for-loop (where it belongs). Thus, code can be written fairly naturally with accessors without incurring unforeseen inefficiencies.

3.2 Accessors v. Smart Pointers

OATH accessors are similar in concept to smart pointers, which have been proposed [Stro87] and implemented [Wang89][Edel90] many times before. However, OATH accessors offer some significant advantages over smart pointers.

First and foremost, OATH accessors never release "dumb" pointers outside of their member functions. This will have important consequences on garbage collection when compiled with C++ compilers that destruct temporary objects as soon as possible (as permitted by the current draft standard and [Elli90]).

Smart pointers overload operator -> to return a dumb pointer to its "internal" object. Such a definition is convenient, since it makes all members of the internal object immediately available through the smart pointer. However, this definition is problematic. Consider,

```
O2 = O1->makeCopy()->transform();
```

The desire is to set `O2` to a transformed copy of `O1`. `O1->` returns a dumb pointer which is used as `this` for the member function `makeCopy()`. The member function `makeCopy()` returns a smart pointer to a new object that is a copy of `O1`. The compiler will create a temporary object to hold that smart pointer. The operator `->` applied to the temporary yields a dumb pointer to the new copy. At this point, prior to the call to the call of `transform()`, the compiler can destruct the temporary that holds the smart pointer. It can do this because it no longer needs the smart pointer once it has obtained the dumb pointer returned from operator `->`. Since the temporary was the only smart pointer referencing the new copy, that copy may be collected (destruction of the smart pointer may cause its immediate destruction, or the invocation of `transform()` could cause a garbage collection). (Note that many current C++ compilers keep temporary objects alive past the end of the expression, so the above has not been a problem. Future compiler implementations will probably not.)

In contrast, OATH accessors define the type interface. So, a member function invocation on a temporary OATH accessor invokes a member function of that object, thereby guaranteeing that the temporary will exist until the end of the function. All uses of "dumb" pointers to internal OATH objects are dynamically within a call to an OATH accessor member function.

OATH accessors also offer the advantage of reference semantics, which makes their use much more natural in the presence of overloaded operators. Finally, since OATH accessors are defined in a parallel hierarchy, they can be assigned and passed as arguments to overloaded functions naturally, obeying the implicit conversion preferences of the inheritance hierarchy. Definition of user-defined conversion operators are often necessary with smart pointers: preference based upon depth in the hierarchy is not considered when user-defined conversions are invoked.

3.3 Accessors and Execution

The parallel hierarchy structure of OATH allows an implementation type to be defined as a "leaf", such that when it is used directly it bypasses the virtual mechanism. Whether or not to define an implementation class as a leaf is a direct trade-off between execution efficiency and reusability via derivation.

For instance, it may be desirable to have an efficient multiple-precision integer class, bigInteger. The following expression with general OATH integers,

```
I1 += I2 * I3;
```

would require two virtual function invocations. To prevent this for the leaf class bigInteger, the accessor class `bigIntegerA` redefines operators `+=` and `*` to call the internal function directly, bypassing the virtual mechanism, by using a scoped function call. If both the internal functions and the accessor functions are inline, then the whole expression can be coded inline when `bigIntegerA`s are used:

```
bigIntegerA BI1, BI2, BI3;
// code here
BI1 += BI2 * BI3; //no virtual
```

Thus, bigInteger can be used as an implementation of integer, rational, real, and complex via the general virtual mechanism in code that needs generality. Alternately, for code that needs efficiency, bigInteger can be used specifically and the virtual functions can be bypassed. The disadvantage of defining an implementation class as a leaf is that it cannot be easily reused via derivation.

4 LIBRARY-BASED GC

Explicit management of objects allocated in freestore is notoriously error-prone and generally completely disjoint from the algorithm that is being coded. One of the major features of OATH is a library-based garbage collection mechanism for OATH objects. This mechanism is a hybrid reference counting and marking algorithm capable of collecting all garbage (including circular references).

4.1 Reference Counting

The internal representations of OATH objects are always allocated in freestore and always accessed via OATH accessors. Thus, it is simple to maintain accurate reference counts [Knut73] on OATH objects. When an accessor is assigned an OATH object, it increments the reference count of the object being assigned, and it decrements the reference count of the object that it previously accessed. Construction and destruction increment and decrement the reference count, respectively.

4.2 Modes

The programmer can select one of four garbage collection modes at compile-time: no GC, incremental GC, stop-and-collect, or combined. No GC mode eliminates the overhead associated with garbage collection. This mode is suitable for short-lived programs and programs that make few objects during execution.

Incremental GC mode maintains reference counts on the objects. If a reference count is zero after being decremented, then the object is deleted. This mode is more convenient than stop-and-collect, since the programmer need not decide when to invoke garbage collection. However, circularly-referenced garbage will not be collected. This mode is probably best suited for programs that do not produce circular references or do not live long enough that the lost storage will matter.

Stop-and-collect mode will maintain reference counts, but will only collect when the programmer calls the function:

```
objA::collectGarbage(int)
```

The int parameter specifies "quick" collection or "full" collection. Full collection will collect circularly-referenced garbage, but can be significantly more time consuming. An extra word of storage per object is used in stop-and-collect mode. This mode is typically preferred for programs that produce circular references, but do not overflow from physical memory into virtual memory.

The combined GC mode collects incrementally and `collectGarbage()` can be invoked to collect circularly-referenced garbage. However, this mode requires two extra words of storage per object. This is the best mode for programs that are long-lived or utilize virtual memory.

In stop-and-collect and combined modes, all OATH objects are linked together so that they can be traversed by `collectGarbage()`. The extra storage required by these two modes is due to the links. In stop-and-collect mode, the objects are singly-linked (hence, one extra word per object). To facilitate incremental collection in combined mode, the objects are doubly-linked (hence, two extra words per object).

4.3 Collecting Circular References

Reference counting is an efficient and robust way to implement library-based garbage collection. However, circularly-referenced garbage cannot be collected from reference counting alone.

Traditional two-pass marking garbage collectors maintain a set of root pointers. Any object that is unreachable from the set of root pointers is garbage. To identify all garbage, the first pass starts from each root pointer and marks all objects that can be reached. The second pass over the objects simply collects all unmarked objects.

OATH accessors can be split into two groups, "internal accessors" and "root accessors". Internal accessors are accessors that are held by OATH objects -- these are the accessors that cause circular references. Root accessors are accessor objects held by the application, on the stack, in static storage, or as members of non-OATH objects. For the two-pass marking algorithm above, the set of root pointers would be the root accessors. Maintaining a record of the root accessors, excluding internal accessors, would be difficult, making use of accessors very expensive. In contrast, OATH uses a three-pass

algorithm which requires only the reference counts ([Chri84] proposes a similar five-step algorithm).

At the beginning of the first pass, the reference counts include references due to both root and internal accessors. During the first pass, the virtual function `clearReferences()` is called on each object. This function clears the mark (a one-bit flag in each object) and then calls `deref()`, which decrements the reference count, on each object that it references. At the end of this first pass, all reference counts due to internal accessors have been removed. The reference counts that remain are due to root accessors.

The second pass calls the virtual function `setReferences()` on each object that has a non-zero reference count (is referenced by a root accessor) and is not marked (has already been visited). The function sets the mark flag and then calls `ref()`, which increments the reference count, and, if unmarked, `setReferences()` on each object that it references. At the end of the second pass, all reference counts due to internal accessors that are reachable from root accessors have been restored. The reference counts of circularly-referenced garbage will remain zero. A final pass is then made to delete each object with reference count equal to zero.

This three-pass marking algorithm is clearly more expensive than the traditional two-pass algorithm; however, it is the same order of complexity. Furthermore, this cost is only incurred by programs that need to collect circularly-referenced garbage. Incremental collection, which is quite efficient, will be sufficient for most programs.

AVAILABILITY

The current release of the experimental OATH library is available via anonymous ftp from site csc.ti.com (192.94.94.1) in the file pub/oath.tar.Z. Comments and further research on this library is encouraged. Please send questions and comments to oath@csc.ti.com.

ACKNOWLEDGEMENT

OATH is a result of many long, often heated, discussions with Larry Spry, whose insights and experience proved quite valuable.

REFERENCES

[Amer90] P. America and F. van der Linden, A parallel object-oriented language with inheritance and subtyping, *ECOOP/OOPSLA '90 Proceedings*, 21-25 Oct 1990, p. 161-168.

[Chri84] T. W. Christopher, Reference count garbage collection, *Software -- Practice and Experience*, 14(6), p. 503-507, June 1984.

[Edel90] D. R. Edelson, *Dynamic Storage Reclamation in C++*, Master's Thesis, University of California at Santa Cruz, UCSC-CRL-90-19, June 1990.

[Elli90] M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[Gold83] A. Goldberg and D. Robson, *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, 1983.

[Gorl87] K. E. Gorlen, An object-oriented class library for C++ programs, *Software -- Practice and Experience*, v17(12), Dec 1987, p. 899-922.

[Kenn91] B. M. Kennedy, The features of the Object-oriented Abstract Type Hierarchy (OATH), 1991 USENIX C++ Conference, p. 41-50.

[Knut73] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, 1973.

[Lea88] D. Lea, libg++, the GNU C++ library, USENIX C++ Conference, 1988, p. 243-256.

[Stro87] B. Stroustrup, Possible directions for C++, USENIX C++ Workshop, 1987, p. 399-416.

[Wang89] T. Wang, *The "MM" Garbage Collector for C++*, Master's Thesis, California Polytechnic State University, 1989.