The GNU C++ Library

Doug Lea
SUNY Oswego & NY CASE Center

DRAFT

The GNU C++ library (`libg++`) was among the first widely available general-purpose C++ class libraries. Some classes were designed and implemented as early as 1985 (originally in support of other efforts). The library was made available in 1987. I was the primary original developer. Several others have contributed ideas and code. Contributors include Dirk Grunwald, Doug Schmidt, Kurt Baudendistel, Marc Shapiro, Eric Newton, Michael Tiemann, Richard Stallman, and Per Bothner. Cygnus Support currently maintains and distributes the library on behalf of the Free Software Foundation (FSF). Also, hundreds of users have contributed improvements, fixes, suggestions, clarifications, and bug reports. While it has been ported to other platforms, `libg++` is normally used on Unix systems in conjunction with the GNU C++ (`g++`) compiler. It is available via anonymous ftp from `prep.ai.mit.edu`, among other sources.

The basic structure of `libg++` remains almost unchanged from that described in a 1988 Usenix C++ conference paper. It contains:

1. Classes representing strings, numbers, and other black box values, along with similar *Abstract Data Type* (ADT) classes representing sets, sequences, maps, etc.
2. IO Streams and related support provided by any minimal C++ library.
3. Storage allocation classes and utilities.
4. "Lightweight" veneers organizing functionality commonly supported in C libraries.
5. A few other uncategorizable classes and sample applications.

`Libg++` is mainly an "abstract data structure library". Most `libg++` classes are somewhat different in design philosophy, design, and implementation than the classes that you or I ordinarily construct for specific applications. The remainder of this article focuses mainly on these differences without otherwise going into much detail about particular components.

Abstract Data Types and Values

While both may be described as C++ classes, there is a big difference between, say, a `Complex` number and, say, a `BankAccount`. For example, there is a large, well-established mathematical theory of complex numbers, but essentially none for bank accounts. One consequence is that it is simply much easier to develop a `Complex` class containing features that one may be reasonably certain will makes sense across a wide range of applications. This is much less true of any `BankAccount` class one could construct.

A more important distinction underlies the resulting design differences. The "theory" of complex numbers revolves around the properties of complex *values*, not *objects*. Mathematical approaches typically abstract over the actual identities of objects possessing *(Re, Im)* attributes, and just deal with the values themselves – the complex quantity *(2.4, 17.17)* remains the same regardless of which or how many objects report this quantity as `real()`, and `imag()` attribute functions. By design, many operations don't care about the objects, and just deal with the quantities. However, this would be a losing attitude for a class like `BankAccount`. For example, even when we happen to both have the same bank balance, the fact that a particular identifiable `BankAccount` instance belongs to *you* and not *me* is an obvious but critical design issue. These differences result in different styles, approaches, and plans of attack for designing the associated classes and utilities. For example, while it is perfectly sensible to write a "constructive" function that accepts two complex numbers and returns a third representing their sum, there is hardly ever a reason to create a function that accepts two bank accounts and returns a third representing (among other things) the sum of their balances. Instead, the `BankAccount` class contains methods such as `withdraw`, `transfer`, and so on that mutate the states of particular objects.

`Libg++` contains substantially more components like `Complex` than those like `BankAccount`. Many classes maintain "value semantics", in a manner more similar to classic ADT approaches than to classic OO approaches.

There are some distinct advantages to ADT approaches in those cases where they are appropriate:

- Value properties are generally better behaved than object properties. Well-understood algebraic properties may be relied upon in the specification, design, testing, and use of associated classes.
- When users care only about values, not about object structure, it is often easier to hide clever representations and algorithms behind the scenes, and to develop interoperable versions of the same general functionality.
- The resulting semantics are familiar to most programmers. For example, a value-based `Complex` class acts pretty much just like the built-in value type `float`.

However, no one ever uses a straight ADT approach in designing classes. For example, a "purist" approach to a `Stack` ADT would define `push` to return a *new* stack state value, different than the original. So, if `Stack s` denoted a stack with a million items on it, `s.push(23)` would return something representing a million and one. Nobody wants this, partially just because of efficiency. Even with a lot of underlying cleverness, too much data copying is required to represent new values resulting from `push`es. However, it also represents the point at which object-oriented thinking rightfully creeps into ADT-based design. When clients push `17` onto `Stack s`, they essentially always wants `s` itself to change, not to construct a new, distinct representation. For this reason, OO approaches to ADTs usually include mutative operations on objects that propel them into different states rather than, or in addition to, operations that construct new representations of new states and values. (Non-OO ADT approaches often do this too, but usually describe them differently. For example, they might talk about the "symbol `s` being rebound to a new value" after a push.)

The twist in `libg++` and other OO libraries is to support some mixture of value-oriented and object-oriented usage, almost always within the very same classes. This is a natural practice, especially in C++, since the C base of C++ already does this. For example, unlike most procedural languages, C contains both the constructive value-oriented `+` operation for adding built-in number types, as well as the (vaguely) object-oriented `+=` "method" for mutating number objects.

`Libg++` was originally a set of experiments in how to go about meeting the occasionally conflicting demands of the two approaches. The two approaches do indeed sometimes conflict, and lead to different trade-offs seen in different classes. Many of these, in turn, reflect trade-offs made in the language itself. Stroustrup has described C++ as a language supporting *both* data abstraction and OO design. Crosses between ADT and OO designs often find themselves at the very borders of both kinds of support, in ways that most other C++ designs do not.

Arguments, Results, and Copying

Generally, operations defined within a value-oriented approach rest on value-based arguments and results, while those from an object-oriented approach use pointers or references to new or existing objects. When applied to classes, value-passing relies on copy-construction, not reference propagation.

Designers of value-oriented classes often give in to the urge to minimize copying overhead while still conforming to value semantics. There are many techniques for doing this, for example, via internal pointers to underlying representations that are shared whenever the support procedures determine that this is possible. The original versions of many `libg++` classes in fact contained reference counting and other ploys to maintain this effect. They were later removed. C++ already contains simple ways for people to obtain copy versus reference semantics. Programmers themselves are in a much better position to know when to make copies and when to use references. Hiding these matters often leads to less efficient and predictable behavior, especially for classes like `Strings`. In many applications, tricks like *copy-on-write* add more overhead than they save. In many others, explicit use by programmers of pointers to shared `Strings` only when desirable and possible is more effective than any automated policy. Thus, except in a few cases where copy-prevention strategies are transparent and algorithmically superior, `libg++` classes maintain the convention that a copy-constructor actually makes a copy. Similar remarks hold for assignment and other operations.

Storage Management

Such decisions reflect the idea that a basic support library should provide *mechanism* not *policy*. Most `libg++` classes are designed so that users who want to implement their own policies are provided with all the tools to do so. `Libg++` contains several classes and utilities that facilitate development of specialized allocation and management. For example, an `Obstack` class supports "mark/release" allocation and deallocation. The `MPlex` class helps manage sparse tables. An optionally included version of `malloc` (underlying operator `new`) has been shown to provide superior performance than most other versions for typical C++ (and C) programs. Other classes provide mechanisms useful for very special allocation needs.

However, this stance is probably the least defensible overall design decision in all of `libg++`. Effective, correct, and efficient storage management in C++ is sufficiently difficult and fragile to demand a better alternative. The only general solution is to rely on automated storage management (garbage collection). If attempts to provide full, transparent garbage collection in C++ succeed, the library (or a version thereof) should be redesigned to exploit it.

Inheritance and Overloading

Value approaches traditionally lie in the world of overloading and parameterization, while OO approaches obtain generality via inheritance. The two do not always mix well in C++. The resulting "edge effects" lead to some pervasive design trade-offs.

For example, suppose you define a `String` class, along with a value-oriented `operator +` method or function that returns a new `String` representing the concatenation of its arguments. Now define subclass `RString` that adds an in-place `reverse` method to `String`. Without evasive action, this leads to a problem in user code such as:

```
RString t, u;   //...
RString s = t + u;
```

Depending on other class details, users may obtain either a type error or unexpected behavior, due to the fact that the expression `t + u` returns a new `String`, not an `RString`. There are workarounds to this, but none of them are very satisfying or general. For example, if `+` were a non-class operation, then overloading another special version for `RString`s would work here, but not when `s, t,` and `u` were passed to:

```
void app(String& a, String& b, String& c) { a = b + c; }
```

Here, the `String` version would be called instead, leading to undesired behavior. Similar snags result from other strategies.

The net result of these considerations is that value-oriented classes are *not* readily subclassable in C++. The best two solutions are the most extreme ones: Either give up on value semantics (at least for troublesome operations) or give up on subclassability for classes with extensive reliance on constructive functions.

Most "lightweight" classes (e.g., strings, multiple precision numbers, bit sets) take the latter option. In consequence:

- Class interfaces are very extensive. The base classes provide functionality that, although rarely needed, is otherwise difficult to support without subclassing.
- No operations are virtual, and many are inlinable.

These consequences are not *all* bad. However, in classes like `String`, these issues along with the additional need to interconvert with other representations (`char`, `char*`, `RegExp`), lead to an embarassing number of methods and related functions.

If I were redesigning them today, I would surely take the first option, and not provide an extensive value-based support interface. This would place the structural design of such classes closer to that of the kinds of OO applications classes people ordinarily write, enable better separation of interfaces and implementation, simplify some algorithmics, avoid nasty C++ issues such as redundant copy construction and the deletion of temporaries, and allow the use of inheritance to express commmonalities among classes. However, it is also very likely that users would complain about inconveniences stemming from the lack of simple value-based operations such as constructive string concatenation. The code would probably also be a bit slower because of virtuals and lack of inlinability. In classes like `String`, obtaining efficiency close to that of raw `char*`'s was an important design decision early on in C++ and C++ libraries. Making `String`s simultaneously better and (often enough) faster than raw C character manipulation made the transition from C to C++ far easier for many programmers.

Containers

The first option (not supporting many constructive operations) was indeed taken for `libg++` container classes including sets, maps, lists, and queues. These classes mainly support standard "object semantics". For example, there is a `Set::operator |=(Set& b)` to union all of the elements of `b` into the receiver, but not a `Set operator |(Set& a, Set& b)` to construct a new `Set` by unioning two others. There were a number of reasons for this decision. The simplest is that mutative operations are most typically needed in applications using such classes.

There is also a technical reason. It is a common (if not always defensible) policy to tie the interfaces of classes like `String` and `Complex` to particular representations. However, classes like `Set` really must be *abstract base classes*. They provide interfaces without providing implementations. There are countless ways of implementing `Sets` (lists, arrays, trees, tables, and so on). It is a terrible idea to settle on any one particular strategy. The many `libg++` classes that implement this functionality in particular ways are defined as subclasses. But one cannot construct (direct) instances of abstract base classes. Thus, it is impossible to declare a constructive version of `operator |()` that covers all cases in the first place, regardless of whether other overloading versus inheritance issues could be settled. (A workaround would be to define this operator to return a pointer or reference. But this interferes with value semantics and leads to storage management responsibility problems.)

Inheritance and Parameterization

Container classes may be used in two slightly different roles:

Collections. Classes that keep track of groups of objects that are all related in some way or are all to be manipulated in a certain fashion.

Repositories. Classes that "house" groups of objects while also providing structured access.

The basic implementation difference is that collections hold pointers to objects that "live" elsewhere, while repositories hold and internally manage the objects themselves. Luckily, the low-level mechanics do not so much that these two forms cannot be combined via the convention that any object used as an element in a repository must support a copy constructor, an assignment operator, and, in some classes, a default constructor, an equality function, and/or a magnitude comparison function.

There are two reasonable stances in designing and using pointer-based collections. For example, for `Stack`s, one may either define a single class that holds pointers to *Any* object, or design a special class for each different element type. In the latter case, parameterization mechanisms avoid the need to write so many special classes that differ only with respect to element pointer type information. `Libg++` does not provide policy about this issue, only mechanism. One may define a `Stack` that holds pointers to anything as:

```
typedef void* AnyPtr;
AnyPtrSLStack mystack;
```

(`Libg++` containers were designed long before C++ `template`s were defined and implemented. They still rely on the use of a manual expansion tool rather than `template` mechanisms. As support for paramterized types in C++ improves, the distributed versions are being modified accordingly. Dependence on simple manual tools resulted in other minor trade-offs as well. For example, even though desirable, different collection classes are not linked to, say, a `Collection` superclass and/or other intervening abstract classes. The two-level abstract/concrete organization was hard enough to use as it was. This, in turn led to unnecessary code duplication within the library.)

These declarations define a stack that may hold instances of any kind of class whatsoever. This is OK for putting things into a stack, but sometimes less so when they are pulled out. Unless it somehow happens to have additional information, a client looking at the `top` element does not know anything at all about its capabilities. As far as type information is concerned, it could be anything.

On the other hand, if a client has a `WindowPtrStack` (i.e., a stack holding pointers to objects of class `Window`), it knows that all elements are `Windows`. The objects might still be of any subclass of `Window`; perhaps `BorderedWindow`, `ScrollableWindow` or whatever. But they are surely at least `Windows`. This guarantees that clients can perform window-based operations on all of the objects without having to bother with type tests, downcasting, or error-handling details.

Parameterized collection classes are thus generally safer than unrestricted classes and lead to simpler use by clients. However, because this is a matter of *relative* safety, there is much room for judgment and disagreement about designs. For example, in a particular application operation, one may really require that all objects are in fact `BorderedWindows`, in which case type testing, etc., would still be warranted whether stacks of `Any` or `Window` were accepted as arguments. Given this, along with the fact that parameterization can generate wasteful multiple versions of the same code but with different type contraints, a library must provide both options.

Moreover, only the parameterization option is viable in the case of repositories. Even though the high-level source code is identicial, each version of a parameterized class maintains storage space, arranges construction, etc., specialized for a particular element type. Thus, both types and executable code are different for each kind of element. In fact, while very useful, the resulting code is slightly dangerous. For example, the type information in a repository set, `s`, of `Windows` does not indicate that if `s.add(b)` for a `BorderedWindow` `b`, then only a "chopped copy" of `b` is actually held in `s` (i.e., the internally held copy of `b` will act only as a `Window` on access).

Performance

It is not very hard to come up with a basic data structures library. Programming techniques for implementing most components are familiar to most programmers. It is another matter to design and implement a library containing among the *best* data structures and algorithms known for supporting common applications. `Libg++` includes both "elementary" structures such the obvious implementations of lists, complex numbers, etc., as well as "fancy" ones including balanced trees, self-adjusting arrays, and sophisticated random number generators. Nearly all of these are implemented in a completely interoperable fashion. For example, programmers may switch from a simple array-based implementation of `Sets` to one based on `SplayTrees` with very little effort, without caring at all about *why* splay trees happen to speed up their application. Of course, it might be that simple arrays are faster. One reason for including so many different implementations is that general-purpose library classes are used in a much broader range of contexts than are application-specific classes. The library writer has no idea of the expected execution profile and possible trade-offs. The best that can be done is to supply enough versions so that the likelihood of acceptable performance is high enough for the library to be considered useful.

Specification and Testing

In part because most of them are based on ADTs, `libg++` classes and utilities include some fairly effective specification and testing constructs.

Most classes themselves include an internal invariant check method `OK()`. Whenever invoked, this function checks a collection of run-time evaluatable constraints that must hold internally for the object to be in a consistent state (not necessarily the "right" state, just a legal one). For example, the `OK()` method in a binary search tree checks that the tree is actually ordered, and `OK()` in the multiple-precision integer class checks that the internal representation describes a legal integer. There are limits on these kinds of checks. For example, they cannot usually diagnose contamination or other errors surrounding dynamically allocated storage. However, they serve effectively as internal and external guides to correctness – every public method must respect listed invariants.

The `libg++` distribution also includes a number of *trace tests* that propel objects through states for which known properties should hold. This is where the wealth of knowledge about ADTs comes in handy. For a simple example, a stack should come back to its original state when a `push; pop` sequence is applied. The test suites place objects through a large number of such exercises. Of course, these measures cannot themselves guarantee total correctness. Implementation errors stemming from nonportable constructions, insufficient testing, and incompatibilities across different versions of classes do still occur.

Conclusions

`Libg++` and other libraries (e.g., NIHCL) designed relatively early in the evolution of C++ have served as examples and counter-examples for the development of many others of their general form. The language, compilers, users, and typical applications have all evolved since their original design. Besides pseudo-templates, `libg++` contains other designs and code that are almost anachronistic, and design assumptions that no longer hold. (For example, a few existing marginal efficiency measures may actually slightly degrade performance on some RISC platforms.) Still, many of the design issues and solutions remain useful guides in the development of resuable C++ components of any kind.