C# Inspired Collection Library for C

Kristian Andersen and Ulrik Damm

February 14, 2013

0.1 Background

The purpose of this project is to specify and implement a collection library for the programming language C. The project is inspired by well-adopted collection libraries like the .NET and Java collection libraries.

0.1.1 Why?

Most people who learn programming today will start out with OO (Object Oriented) languages. An increase in the popularity [1] has put languages like C#, Java, Ruby, Python, etc. at the top of the charts. Languages like Java and C# come with standard libraries that contain well-documented and comprehensive collection libraries that are built right into the langugage and ready to use. They use consistent naming and are easy to use. This is however not the case for the procedural programming language C. This trait makes the language look almost antique in comparison with these newer OO languages and can scare off programmers who started out learning OO languages.

Despite the increased interest in the OO languages C still remains one of the most popular languages and has many good uses, hence the need for a collection library that is convenient to use for programmers mostly familiar with OO language concepts.

0.1.2 Existing solutions

There are already a number of existing collection libraries available for the C programming languages. Some are more structured and complete than others. In order to devise the requirements for the design of the project we must investigate how the existing solutions stack up against what an OO programmer might be familiar with. We will therefore compare the existing solutions against key features and characteristics of the .NET and Java collection libraries.

The single most important feature of the libraries is their documentation. It is very important that the libraries document their full API in a structured manner. It is also very important that the libraries use a naming and code structure that is consistent across the entire API and library. Most object oriented languages implements memory management in the form of garbage collection or reference counting. This features becomes even more significant with collection libraries that often contain large amounts of data. The last key feature is convenience (ease-of-use). Even though the term is very vague it is also important.

In addition to the key features the collection library should also include a decent variety of the most commonly used data structures like lists, tables, dictionaries, queues, stacks, etc.

[1] See http://langpop.com/ and http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html

The importance of these key features will be further discussed in section ?.

Below is a table showing comparison of the key features for each of the collection libraries for C that were found.

Library	**Documentation**	**Naming**	**Memory management**	**Avail. data structures**
GDSL	Good but no examples	Good	None	Decent
GLib	Good	Inconsistent	None	Few
APR	Bad	Inconsistent	Partly	Few
NPR	Chaotic and no examples	Good	No	Many
SGLIB	Good	Bad	No	Many
Gnulib	Good	Bad	?	Many
LibDS	Good but no examples	Bad	No	Decent

Table 1: Comparision of collection libraries.

The amount of features and data structures each collection library varies greatly. None of the libraries have all the qualities desired and none of them fully support memory management. Most of the libraries had good documentation but were lacking usage examples and had bad and/or inconsistent naming conventions.

One aspect that we did not include in the investigation of existing libraries were that of testing. Testing is naturally an important aspect of a well functioning library. None of the libraries disclosed test coverage or how to run tests. Some of the libraries didn't have tests.

- GDSL (Generic Data Structures Library) http://home.gna.org/gdsl/
- Glib (Gnome Library) http://developer.gnome.org/glib/2.34/
- APR (Apache Portable Runtime) http://apr.apache.org/
- NPR (Netscape Portable Runtime) http://www.mozilla.org/projects/nspr/
- SGLIB (A Simple Generic Library for C) http://sglib.sourceforge.net/
- Gnulib http://www.gnu.org/software/gnulib/MODULES.html#ansic%5Fext%5Fcontainer
- LibDS (A Portable Data Structures Library) http://libds.sourceforge.net/doc/index.html

0.1.3 Requirements

A collection library must:

- have full documentation of its full API with detailed examples.
- use consistent naming for types, functions, etc.
- provide an easy way to get started for the user.
- provide support for memory management.
- be overall easy to use.

0.2 Analysis

0.2.1 Code structure

To have a well structured source code is very important in regards to debugging, testing and maintainability of the project. That's why we are strictly following some basic rules regarding structure, information hiding and naming.

One of the most fundamental structural decisions is to enforce information hiding throuhgh the idea of the black box: a potentially decoupled interface and implementation, and only exposure of the interface to the user of the module. In Java-like languages, you would implement a black box like this with the public and private keywords, for selecting which parts of a class should be exposed, and which should not. In addition, things can be marked as not visible to a user, but visible to the rest of the package. In C, such levels of visibility is implemented with header files. Typically, you will have one header file and one implementation file per black box, with the header file being the public interface. Other projects have a single header file for all public interface code in a library[2]. The one-header-per-module approach is the most common in C, and also what we will use. In addition to this, to implement a package-private like behavior, modules may also have a header file for use internally in the project, which should not be exposed to the user. This allows use to "leak" some implementation details internally. This should be used for details relating to the internal structure of the project's data structures, not the implementation of the specific algorithm or data structure.

In addition to keeping a minimally exposed interface, it should also be documented with a description of each function, along with pre- and post conditions. See the documentation

[2]The one-header-per-library method is mostly used when the product of the project is an API, and not an implementation. An example of this is OpenGL, which is an API implemented differently across platforms, with the whole public interface in the gl.h header file.

chapter for more.

0.2.2 Memory Management

Manual memory management is often a big hurdle for people coming from garbage collected languages. With the right techniques and code structure, it can become less of a problem, but it's still something that can cause headaches. This is especially in collections, where programmers are often tempted to just create a static array instead of something that is dynamically expanding, because of the trouble of having to keep track of when you don't need it any more, leading to buffer overflow issues. And, apart from direct code problems, garbage collection is just a convenience that many high-level programmers expects to have. This is why we want our collections to be garbage collected.

Since there is no support for garbage collection in the C runtime, it has to run on top of the standard memory allocation. there are a number of libraries to do this, the most popular of these being the Boehm-Demers-Weiser conservative garbage collector (also known just as Boehm's garbage collector), which is used as the garbage collector in Mono and as a leak detector in the Mozilla project. It has replacement functions for malloc and realloc, which will instead allocate memory in heap blocks, allocated by the garbage collector, and associate marking bits. For collecting memory, it uses mark-sweep to find unused memory blocks and free them. Finalizers can also be registered for a block of memory, which could be used to replicate the functionality of deconstructors.

For use in a C program, the mark-sweep method has the advantage of being able to be made conservative. This ensures that there will be no problems in trying to differentiate pointers and integers, since there is no harm done by retaining a block of memory, that isn't in use, but has an integer —which can be interpreted as a pointer, but really is just a value —pointing to it. Non-conservative garbage collectors, such as stop-and-copy, would have to make a choise wether to move the block, and update the potentially false pointer, or garbage collect the memory.

In regards to performance, the mark-sweep collector can be slow for larger allocations, but is generally fast for smaller ones. Boehm's garbage collector claims to be on-par or faster than regular C memory management for smaller allocations, but slower for larger ones.

0.2.3 Data Objects

When using object oriented languages, we can use the composition pattern to make objects implement a specific method, giving a common interface to many different objects. An example

of where this is used is in object equality: all objects marked as comparable must implement an equals method, and we can then compare all kinds of objects with each other, and easily extend the amount of comparable objects.

In C, we do not have this pattern, and we will have to find a way around this. When adding data to a collection, we can't just specify the type as Object, like you would be able to in Java. One common replacement for a generic base object is to use a void pointer and a data length variable. The problem with this is that you still don't get the composition from object oriented languages. That's why we went a little further with our approach.

Our solution is to make a object struct, which can contain either a primitive type, a string, or any other data in form of a void pointer and a data length indicator. Along with just the data, the object also has a type, in the form of a string, which can act like a class name. This can be used to identify objects in a collection, or when comparing them.

For comparison, we start by saying that if two object are of two different types, they are automatically inequal. If they are of the same type, we need a specific comparator for that type. For this, you can register comparator functions for a specific type. This will be added to a global list of types and comparators, which will be used for lookup, when trying to compare two types. With this approach, you can define your own data type and comparator, and then add it to a collection and compare instances of this object type, mimicing what is in object oriented languages.

0.2.4 API

One of the problems with a lot of c libraries is the API design. If you don't put some though into making a consistent and readable interface, it will make the whole project seem messy, and confusing functions with a lot of parameters will be more prone to errors.

One of the goals in this project is to have consistent and well organised interfaces to our collections. We have made a style guide, that all code should follow, that is inspired both by classic C style naming and more Java-style method naming.

We have defined a style guide for both public interfaces and internal function implementations.

Style guide for public interfaces

- All names must be in lowercase, with underscores seperating each word. To avoid naming collisions, they should be prefixed first with `cc` for the library, and then the name of

the object or module it belongs in (e.g. `cc_set` and `cc_set_add()`). Struct names must end in `_struct`, and should be typedef'd to the name without `_struct` (e.g. `struct cc_set_struct` and `cc_set`).

- All objects must have a constructor function called `new` (e.g. `cc_set_new()`). All other "methods" to this object should take an instance pointer as the first parameter (e.g. `cc_set_add(cc_set *set, cc_object *obj)`). If the function returns anything through a parameter, it should be at the end of the parameter list. Furthermore, all objects must define a type string as a global constant, whose value should be the name of the object (any prefixes included).
- All global constants must be defined in the interface as an extern const pointer, and be implemented in the belonging implementation file (e.g. `extern const char * const cc_set_type`).
- No struct should be defined in the public interface. All interaction with the object should be done through functions. The definition of the struct should be in the private or package private interface.
- Everything defined in a public interface should have Doxygen documentation.

Style guide for internal implementation

- All object structs must start with a `cc_collection` variable.
- In the constructor, the object must register a comparator for its type and specify an implementation for `enumerator_move_next` in the `cc_collection` variable.
- Elements in an implementation file must have the following order: constant definitions, private function declarations, constructor implementation, public function implementations, private function implementations.

0.2.5 Documentation

We're using the documentation system Doxygen for generating documentation for all objects. Everything in the public interfaces for objects should be documented with description and pre- and post conditions.