Table of Contents

Preface

This report has been written in May 2013 in a 15 ECTS project at the IT University of Copenhagen under the supervision of Hans Henrik Brandenborg Sørensen and Peter Sestoft.

During the project we developed a collection library for the programming language C. The library is strongly inspired by standard libraries found on the .NET and Java platforms. The library mimics characteristics and concepts traditionally only found in collection libraries for Object Oriented programming languages. These traits include garbage collected data structures, enumerations, collection filters and more. The library also adheres to the same level of consistency and documentation which is customary for OO-language collection libraries but uncommon for C libraries.

Introduction

Data structures are a natural asset in the development of any application. Unlike most many languages, C does not come with a full stack of data structures, neatly packaged together into a collection libraries. In fact there is not a single data structure in the standard library. Developers find themselves faced with two options. Either write the implementation you need from scratch or go online and find an existing implementation that looks usable.

Developers find themselves scouring the web for good libraries and are left with a number of choices. None of the many options available are very much alike the typical OO-language collection library.

Languages like Java and C# are highly regarded in part because of their extensive, well-documented and loved collection libraries that come as part of their standard libraries. Developers who primarily develop with languages like these are left with no similar options in C.

The general purpose of this project is to investigate how a collection library written for C might be inspired by other libraries from object-oriented programming languages like C#.

Existing libraries

There is a large number of libraries with data structures available for C. In the table below is listed some of the most commonly known by C developers.

Existing solutions for ADT's

Library	Documentation	Naming	Memory	Avail. ADTS
GDSL	Good, no ex	Good	None	Decent
GLib	Good	Incons.	None	Few
APR	Bad	Incons.	Partly	Few
NPR	Chaotic, no ex	Good	No	Many
SGLIB	Good	Bad	No	Many
Gnulib	Good	Bad	?	Many
LibDS	Good, no ex	Bad	No	Decent

Most of the libraries listed here are not strictly collection libraries but larger libraries where a small part of it is data structures. This is the case for APR, NPR, Glib and Gnulib.

Finding libraries purely dedicated to data structures is not very easy. We were only able to locate GDSL, SGLIB and LibDS. All of these have one thing in common; no memory management. In fact most of the libraries have no memory management support, which means the developer has to manage the memory manually. All C developers are used to this fact it is however very uncommon for developers from OO-languages to manage memory themselves.

An intricate part of any library is testing. The libraries from our research have varying degrees of testing implemented. Some have a few tests per data structure. A few have full-blown test suites with all public functionality tested. A single claims to have high-coverage unit tests, but is closed source. A lot of have no tests or no tests checked in to the public source.

All the libraries have varying quality of documentation and naming conventions. These two qualities are crucial factors for the entire quality of a library.

Many leaps has happened in computer science since C had its peek, let alone was invented. However it seems that not many of the advancements made in collection libraries in newer programming languages has found their way in to C. There could be several explanations for this. Consider this: many of the newer features doesn't mix well with the non-object-oriented nature of C.

Vision

In examining the existing libraries containing data structures available for C, it is found that there is room for improvement. This project will aim to describe and implement a dedicated collection library for C that fills some of the gap between existing solutions for C and the OO-languages.

This library will consist of a small selection of commonly used data structures. The data structures that will be implemented are as follows:

- Linked list
- Array list (also known as vector)
- Queue
- Stack
- Set (unique value store)
- Dictionary (hashed key-value store)
- Sorted list
- Sorted set
- Sorted dictionary

- Binary tree

In additions to the features that would normally be associated with common data structures like the ones mention above, the library will aim to implement a number of concepts and features inspired by other OO libraries. Most notably the data structures will all have support for enumeration, something that none of the existing libraries for C has. The library will also aim to be very easy to extend with additional features, types and structures. This will be done with the means of solid code conventions, good documentation and unit testing.

Following code conventions is always good sense when writing library code but it becomes even more important with C because of the lack of modularity in C. There are no namespaces, packages or classes into which functions can be neatly organized.

Just as important as code conventions is documentation. A library must have a well documented API for developers to easily use it. A library must also be well documented internally for developers to easily extend it.

To further mimic the trades of libraries from other libraries the library of this project will support garbage collection. Memory is manually managed in C and removing that concern from the developers when they deal with collection libraries will make the library even more usable.

Architecture

There are generally two different ways of designing a system in procedural programming languages. The first is the “classic” way, where you would have a single function doing one task on all the supported data types. The other is the more modern object-oriented way of having one implementation of a task for each supported data type. An example of the difference could be a method, which will calculate the weight of a car: the classic method would have a single function with support for all known types of cars; the object-oriented would have one specialized function on each car type. Each method has pros and cons. If you want to add a car type, in the classic method, you would have to find every place in the code where car-type-specific calculations were made, while in the object oriented, you would only need to implement the functions for the new car type. On the other hand, if you wanted to have a function to calculate the height of a car, in the classic method, you would need to implement just one function, where in the object-oriented way, you would have to find all car types, and implement a function for each.

Newer languages with focus on object oriented programming makes the decision of which to use very easy. In pre-object languages, such as C, the choice is a bit harder. The easy answer is to use the classic way, since this is what the language is built for; however, there are ways to build functionality which replicates some object-oriented functions in a procedural way, which makes object composition available.

One other difference between the two methods is execution speed. The newer, more dynamic ways of programming uses a lot of dynamic lookup at runtime, which have a performance impact, compared to more static programming. If you compare the same code written in C and C++, the C++ code will most likely be a bit slower, since dynamic features like method calls uses runtime table lookups. A more extreme example is languages like JavaScript, which, even when compiled to bytecode, runs at half the speed of the same C program. [remember source, that article with asm.js and Unreal Engine]

There isn’t a single general answer to, which method is the best to use. It depends on the specific situation. What we can do, is that we can set a general style for the project as a guideline, not a rule. The general style should be prioritized over the other, but if one doesn’t make sense to use in a situation, the other should be used.

Since most C libraries seems to be using the classic, non-object way, we have chosen to attempt a more modern approach to C programming. The general style guideline for the project is to use object-like thinking when it makes sense: think about how to make things modular and extensible, instead of trying to make it perform.

In the following sections, we will go more in-depth with the different design choices in the library.

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 mark- ing 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 choice 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.

Data Objects

The most obvious problem we encounter is how to have an insert method, which can insert any kind of object. A collection should be able to contain every type of object, which isn't a problem in object oriented languages, where there is a system of classes and instances, but is a little more tricky in C, since there are multiple kinds of objects. We will want a function to insert both a char, which is a single-byte stack variable, an int, a multi-byte stack variable, a string, which is a pointer to a block of memory with an unknown size, along with whatever data structure the user has made.

There are multiple approaches to this. One way is to have the user specify an element size when creating the data structure, and any value can then be put into each element, as long as it doesn't exceed the element size. Another way is to make an insert function for each supported data type, so that you would have functions such as `insert_int`, `insert_float`, `insert_string`, ect.

There are some problems with both of these approaches. The first approach prevents the collection from containing different types of objects, if you don't want to get into typecasting. Also just inserting values will not be type-safe, since there will be only one insert function to handle any type. The second approach solves these problems, but this gets back to the classic/dynamic style discussion earlier. We would not be able to add custom data types this way. The only way to insert a custom data structure would be to constantly serialize it to data.

The approach we have chosen is to out-delegate the task of handling different types away from the collections. We will define a data type, which will act as a wrapper for whatever kind of data needed, called `cc_object`. Along with making the collections only need to worry about one type of data, the cc_object can also handle things like data comparison, value equality, data conversion, serialization and hashing, all in an extensible manner.

The cc_object works by having a data structure, which supports each data type, and has special insert functions for each. The difference from having a lot of insert functions on the collection is that when you add a new data type, you only have to add one new function to the cc_object, instead of adding a new function to each collection. This is the same idea as used in Objective-C, where the collection classes cannot contain primitive data types, so numbers gets wrapped in NSNumber instances, and various structures in NSValue instances. If you would save an integer in a collection, you would make an `NSNumber` with the `-numberWithInteger:` method, and the other way around call the `-integerValue` method to get the primitive value back. Likewise, with cc_objects, you would create a new object with the `cc_object_with_int` function, and to get the value back, use the `cc_object_int_value` function. When using custom data types, the user can just make a custom cc_object initializer method, to keep everything type-safe.

A very useful feature of object-oriented languages, which is difficult to replicate in C, is the concept of interfaces. Suppose that you want to be able to compare two cc_objects. If it was an object, you would just make a comparable-interface, and have each object-subclass implement a compare method. In C, we'll have to use a slightly different method.

To handle different types of cc_objects (which in object oriented languages would be to make subclasses of an abstract superclass), we use a string identifier on each object, to indicate its type. The CCollections library has a few built-in types, such as `cc_object_type_int` and `cc_object_type_string`, and it is possible to define custom types. Allowing us to identify the type of an object, allows us to get the same result as using an interface. With a function call, a function can be registered as the comparator function for a specific type. When the user calls `cc_object_compare`, the function will fire compare the types of the two objects. If they are not the same type, we can already say that they're not equal. If they are, we call that type's comparator function.

With the cc_object-approach, we have, in normal C code, replicated some of the functionality from more modern languages, making the code more dynamic and extensible.

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 through 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[1]. 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 chapter for more.

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 organized 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 separating 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.

Testing

As noted earlier only some of the libraries encountered in our research had testing implemented. Some had a few tests per data structure to make sure that the data structure sort-of worked. A few had full-blown test suites with all public functionality tested. A lot of had no tests or no tests checked in to the public source at least.

The library for this project will be using unit tests with coverage of all data structures and all public API. This will serve three main goals of the library.

1. The library is well-tested. Every time a change is made to a data structure or another part of the library, which may affect several data structure, all the tests can be run to check nothing breaks. This dramatically lovers the chance of small bugs creeping in.
2. Tests drive forward implementation. This is also known as Test-Driven Development; TDD for short. Once the general structure of the library has been determined features can be fleshed out by writing the tests first and then the implementation. This will both ensure all API is tested but also that there is no API features that has not been tested.
3. A well tested library will incentivize third-party developers, that wish to extend and contribute their code to the library, to write tests for their code. Looking at the existing tests will also make it easier for the developer to write matching tests for their code.

Error handling

Libraries like C# and Java has native support for Exceptions which makes it really easy for developers to determine if something went wrong and then take proper actions. Since C doesn't have any native support for handling errors we'll have to deal with them other ways. There are several approaches to the problem.

- Return values (–1, –2, etc. corresponding to a error)
- out parameters. functions have a last parameter that will contain an error message or code that must be checked.
- callbacks. functions have a callback parameter that takes a function pointer to a function that will be called with a error message or code.
- global callbacks. like callbacks but there is only a global function that will be called no matter where the error occurs.
- error state. each instance of a data structure holds a state which indicates whether an error has occurred along with an error code or an error message. Developers will have to check to check the error state for each call.

Implementation

---

1. 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. ↩