A NEW COLLECTIONS FRAMEWORK FOR THE D PROGRAMMING LANGUAGE STANDARD LIBRARY

Eduard Stăniloiu[1], Răzvan Niţu[2], Răzvan Deaconescu[3], Răzvan Rughiniş[4]

D is a general purpose, high level and high performance, programming language capable of interfacing with the operating system API and system's hardware. One of the core features of D is represented by ranges, a powerful and new approach of iterating through a set of elements. However, being a state of the art feature, ranges are not used in the D ecosystem at their full potential.

In this work we provide a new collections framework for the D Standard Library that is compatible with the existing algorithms and that provides an API similar to the one provided by ranges. Our implementation enables the collections to infer the safety of the operations from the contained, user provided, type.

We show that our implementation may provide performance benefits of up to 2x compared to the existing standard library implementations, when used in conjunction with a custom allocator.

Keywords: ranges, D, collections, data structures

1. Introduction

The D programming language [4] aims to provide a fast and effective way of writing correct, fast and maintainable programs. D implements modern and powerful features, such as memory safety, function purity and improved code readability, to satisfy the needs of the continuously growing software engineering industry.

Ranges, as we will elaborate in the next chapter, are the D way of iterating through a set of elements. A range is capable of the following operations: accessing elements, testing for emptiness and modifying elements. Ranges provide a great access adaptor for the algorithms in the standard library. This is due to the fact that having a common interface is simplifying the use of

[1]PhD candidate, Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: `eduard.staniloiu@upb.ro`

[2]PhD candidate, Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: `razvan.nitu1305@upb.ro`

[3]Lecturer, Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: `razvan.deaconescu@cs.pub.ro`

[4]Professor, Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: `razvan.rughinis@cs.pub.ro`

the data structures and allows the user to have reasonable expectations about collections.

Currently, the D standard library provides users with a set of collections to use, but the implementation of those predates some of the existing language features. The existing collections use the builtin Garbage Collector [12], assume that the underlying user type is unsafe and don't work well with the language's *const* and *immutable* type qualifiers.

In this work, propose a set of general purpose collections (vector, list, map, hashtables, heaps etc.) that are fast, reliable and compatible with the existing algorithms, while leveraging the powerful features of the language. The collections allow the user to choose the desired allocation scheme, by providing a custom allocator to be used by the data structure. They do not make assumptions about the underlying user type, instead they infer the safety and purity of it, all while being able to work in an immutable environment. The end result of our work is a collections library that has been integrated in the D standard library that is more efficient, more flexible and safer in terms of memory usage than the existing alternative solutions.

The remainder of this paper is organized as follows: Section 2 presents the background, Section 3 highlights the design of our collections and the problems encountered, Section 4 discusses the implementation and Section 5 presents the evaluation. We conclude with Section 6.

2. Background

In this section we will present the *de facto* approach of iterating over a container through iterators and its disadvantages. Next we will discuss the benefits that ranges bring over traditional iterators and how ranges have been implemented in D. Finally we present a few core D concepts: memory safety, purity and type qualifiers.

2.1. Iterators

An iterator represents a way of providing access to the elements of a container. Because pointers represent the fundamental abstraction model used in STL [7], there aren't many use cases for a single iterator; you need both the **begin**ning and the **end**ing iterators of the container in order to safely traverse it. This approach suffers from two shortcomings:

- When working with iterators, the user needs to be careful with the pairing of the **begin** and **end** iterators. Wrongfully pairing two iterators is both a frequent mistake and a hard bug to discover.
- When implementing algorithms, a user, essentially, needs to provide both iterators in order to define the **range** of his container. This leads to clunkier, error-prone and less maintainable code.

Ranges represent an alternative to iterators that offer all of the benefits, but do no suffer of any of the shortcomings. The next section will present in detail how Ranges are defined and how they improve the existing iterators.

2.2. Ranges in D

Ranges are an abstraction of element access and a core part of D. They provide a new approach to the problem of accessing the elements of a container. Ranges were introduced by Andrei Alexandrescu in the **On Iteration** [1] article, pointing out the weaknesses in the *iterator design*, used by the C++ Standard Template Library (STL), and demonstrating the advantages of the *range design*.

The range design that was presented by Alexandrescu has been implemented in the D programming language. For simplicity, we will present the concept of ranges following the D implementation. Note that other implementations may be slightly different, however, the core concept remains the same.

In D, any structure that provides access to the elements of a container, in order to be considered a range, needs to implement three methods:

- **empty()** - acknowledges if the range is empty or not.
- **front()** - provides access to the first element of the range.
- **popFront()** - removes the first element of the range, shrinking it's size by one.

Those three operations define the basic range type, the **InputRange**. The input range abstracts the sequential iteration of a container and it adapts well to streams of data, such as reading from the standard input.

Some containers need to be iterated more than once. Such a scenario makes the InputRange not suitable. The ForwardRange adds to the InputRange interface the **save()** operation, which returns a copy of the range.

For situations where a reversed traversal is required, the **BidirectionalRange** adds two more operations to the ForwardRange interface:

- **back()** - provides access to the last element of the range
- **popBack()** - removes the last element of the range

Lastly, the most powerful range there is, the **RandomAccessRange** is providing the **opIndex(size_t i)** operator, which provides access to the **i**'th element of the container.

2.3. Memory safety

In computer science, a program is being defined as memory safe if, for any possible input, it cannot produce memory corruption. For decades, memory safety has been a struggle [11][9][10] that was enforced by manual code auditing and third party code analyzers. However, in recent years, safe languages that automatically check for unsafe operations have been developed, such as D and Rust.

In D, all unsafe operations, such as taking the address of a local variable or doing pointer arithmetic, are forbidden. Safety checks are enabled by the user by annotating function declarations with the **@safe** keyword. For exceptional use cases where the programmer takes responsibility for any unsafe action, an escape hatch is provided: the **@trusted** keyword. When annotating functions with it, the body is exempted from safety checks.

2.4. Functional Purity

The concept of *pure* functions comes from functional programming [2]. A pure function has two main characteristics that are detailed below.

Firstly, a pure function will produce the same result for the same set of parameters. As a result, a pure function's result may be cached and used to elide subsequent calls of the same function with the same parameters. Secondly, a pure function is not allowed to access global mutable data. As a consequence, pure functions don't have side effects. Such functions may be formally proven to be correct or not.

Purity both helps the user to reason about code logic and represents a powerful optimization enabler for the compiler.

2.5. Type Qualifiers

In D, there are three type qualifiers: **immutable**, **const** and **shared**. An **immutable** object will not change after it was successfully constructed. **const** means that the object cannot be changed through this **const** reference, but the data may be referred from somewhere else through a mutable reference.

In D, type qualifiers are transitive. This means that for an object qualified as **immutable**, not only the reference but also the data that it refers to are **immutable**.

An object is not implicitly shared among threads. In order to share an object, it needs to be annotated with the **shared** qualifier. A **const** reference can be a reference to either a mutable or an immutable object. Figure 1 highlights the qualifier conversion rules in D.

3. Design

This section discusses the goals of our work and the means to achieve them.

3.1. Ease of use

Ranges represent a core feature of the D language. Any user will eventually become familiar with the range interface. As a consequence, we want to leverage this familiarity and benefit from a uniform interface for all of the data structures defined in the collections library.

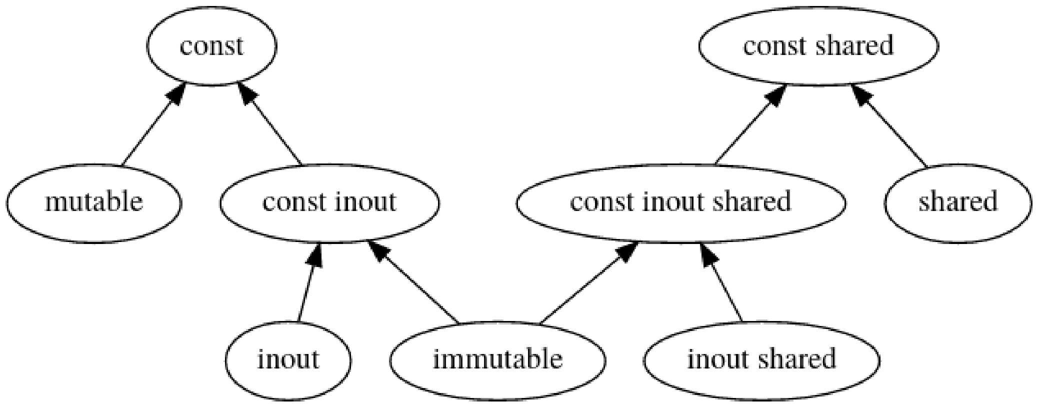

FIGURE 1. Qualifier conversions in D [6]

3.2. Dangling Ranges

As shown in Listing 1, the dangling range problem consists of returning a range to a collection that no longer exists.

This problem is elegantly solved by our design. A collection is defined as a **range + optional primitives**. Because we don't have two distinct pieces, the collection and the range over the collection, the dangling range problem vanishes.

```
int[] foo()
{
  int[5] bar = [1, 2, 3, 4, 5];
  int[] oops = bar[];
  return oops;
}
```

LISTING 1. Dangling Range Example

3.3. Speed

We want our collections framework to be as fast as possible, given the user constraints. To achieve this goal we track the lifetime of a collection using reference counting and enable the user to use custom allocators [3].

3.4. Memory Safety

The type systems guarantees that code annotated with **@safe** does not cause memory corruption. However, a collection will interact with user defined types that may call unsafe functions. As a consequence, no matter how safe the collections code is, the safety of the whole will be determined by the safety of the contained type.

In order to achieve safety, the following two observations need to be taken into account: (1) Memory allocation is a safe operation. There should not be anything unsafe here. We just ask the allocator for a chunk of memory, which he will provide if he has any more left. (2) Deallocation is an unsafe operation from the allocator's point of view because it can not know it there are any more references left to the memory buffer he is about to free. Deallocations made by the collections, on the other hand, are safe because the reference counting provides the guarantee that there are no more references to the buffer at the point of freeing it.

3.5. **Functional style**

D provides the *const* and *immutable* type qualifiers, immutability being an essential part of the functional programming paradigm. Prior to our work, the container module did not support the use of such qualifiers.

The new collections, on the other hand, supports such qualifiers. This enables the implementation of multithreading algorithms and the use of functional style idioms.

3.6. **Collection properties**

The collections defined in our library have the following properties:

- Provide at least the following methods: *empty()*, *front()*, *popFront()*, *save()*.
- Are usable in *safe*, *pure*, *nogc*, *nothrow* contexts.
- Are usable in conjunction with D's transitive type qualifiers: *const*, *immutable* and *shared*
- Are usable with existing algorithms that work on ranges
- Are optimal in terms of performance, provided that the user constraints are met.

The new collections framework enables constraint driven choice: the user can request a collection that satisfies certain constraints [5]. For example: if the user desires a collection that has a key - value mapping with constant insertion and retrieval time, he will be provided a hashtable. In order to achieve this, the internal implementation of the collections uses D's powerful compile-time introspection.

4. **Implementation**

Collections should provide both speed and flexibility. As a consequence, the user must be able to choose the memory allocation strategy. If the constraints permit it, the collection may use the garbage collector, but, if necessary, the user may plug in the desired memory allocator to be used. In D, code that does not use the garbage collector is annotated with the **@nogc** attribute. Also, in order to be classified as memory safe, the collection needs to infer the safety from the contained type. In order to support qualifiers, the collection needs to take into account race conditions and the special cases imposed by the *const* qualifier.

Because we are not necessarily using the garbage collector, the collections need to do their own memory management, which is achieved through reference counting. Because we need to support type qualifiers we were faced with the following problem: how can we reference count immutable objects? This problem arises because in D, when a type is qualified as immutable it's content also becomes qualified as immutable. To solve this issue, we make use of D's standard library `AffixAllocator`. The advantage of using this allocator lies in the fact that AffixAllocator fronts each allocation with an extra `Prefix` that is independently typed. We will use the Prefix to store our reference count, as shown in Figure 2.

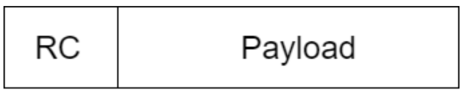

Figure 2. Reference Counting with AffixAllocator

4.1. Reference counting dynamic allocator interfaces

Building a reference counted struct around the dynamic allocator interface involves multiple steps.

The first step was to add the **incRef()** and **decRef()** methods to both the *IAllocator* and *ISharedAllocator* interfaces. The concrete class that implements the interface is going to keep the reference counter inside its implementation; this means that we implement *intrusive reference counting* [8], meaning that we are going to have good cache locality when accessing the object and updating the reference count, and so we will have a good runtime performance.

The actions that the concrete class needs to take, for each reference counting method, are:

- *incRef()* - Increase the internal reference count; For stateless allocators, such as Malloc and GC, it does nothing.
- *decRef()* - Decrease the internal reference count, and when the reference count reaches 0, the object self-destructs. As with *incRef()*, for stateless allocators it does nothing.

The next step of the implementation was to update the concrete classes from the module that implements the interfaces: **CAllocatorImpl** and **CSharedAllocatorImpl** respectively. To achieve this, a technique is required to enable an object to self-destruct when the reference count hits 0. To better explain our technique, we first provide information on how an object is created.

The module provides two helper functions, *allocatorObject()* and *sharedAllocatorObject()* that build a dynamic allocator from a static allocator, regardless if it's stateful or stateless. We previously mentioned that we do not reference count stateless allocators, so we will concentrate our attention on stateful allocators.

When *allocatorObject()* is called on a stateful allocator *A*, it uses it to provide the memory required to hold the dynamic allocator *B*. Then B is built in place (we call this operation **emplace**) and A is moved at the memory location where B was constructed. Listing 2 highlights a schematic version of our implementation.

```
// allocate memory for the dynamic allocator
auto state = a.allocate(stateSize!(CAllocatorImpl!A))

// emplace the dynamic allocator
auto dynAlloc = emplace!(CAllocatorImpl!A)(state);

// move the static allocator inside the dynamic one
move(a, dynAlloc.impl);
```

Listing 2. Creating a dynamic allocator

For such constructed objects, once the reference count reaches 0, the concrete implementation will have to do the above steps backwords:

- move the initial static allocator, held by the **impl** field on the stack.
- get a reference to the memory chunk that it has initialized.
- use the static allocator, from the stack, to free the memory, thus preventing memory leaks.
- when the static allocator goes out of the function stack scope, it will have his destructor called, thus finishing the self destruction phase.

The final step in the implementation of the reference counting of dynamic allocators was implementing the structs that call the *incRef()* and *decRef()* accordingly. Those structs, which now represent the API with which users work, are **RCIAllocator** and **RCISharedAllocator**, the latter is being used for *shared* allocators.

From this point on, the only entities that hold a reference to a raw **IAllocator**, respectively **ISharedAllocator**, object are **RCIAllocator** and **RCISharedAllocator**. These implement all the operations that require reference counting, such as constructor, destructor, copy-construction and assignment operations, and call *incRef()* and *decRef()* on the private *IAllocator/ISharedAllocator* object. Listing 3 provides a simplified high-level overview of the implementation of **RCIAllocator**.

```
struct RCIAllocator
{
  private IAllocator _alloc;

  private this(this _)(IAllocator alloc) {
      assert(alloc);
      _alloc = alloc;
  }

  @nogc @safe this(this) {
      if (_alloc !is null) {
          _alloc.incRef();
      }
  }

  @nogc ~this() {
      if (_alloc !is null) {
          bool isLast = !_alloc.decRef();
          if (isLast) _alloc = null;
      }
  }
  /* ... */
}
```

Listing 3. RCIAllocator snippet

4.2. Pure allocators

From the type systems' point of view, allocators are not pure as they access and modify global state in order to acquire and release memory. From a logical point of view, they must be pure, as they are just a handle to acquiring system resources.

The Garbage Collector represents a special case with regards to purity. The type system acknowledges its existence and therefore considers any call to the GC as a pure function call. In contrast, when implementing a library solution for allocators, calls to functions that allocate memory are not pure since they rely on functions that modify the systems resources (which are inherently impure). However, as we mentioned above, from a logical standpoint, these functions are pure. Therefore, we must treat such functions as pure even though they are, technically, not. This can be achieved by faking purity. Still, in this scenario, immutable collections that are also pure may have some operations wrongfully optimized away, thus leading to incorrect behavior.

To solve this issue we have created a technique that avoids such optimizations. The general idea is highlighted in Listing 4.

```
{
  private union {
      void *_;
```

```
        size_t _pMeta;
    }
    /* ... */
}
```

Listing 4. Break immutable transitivity

The compiler knows that we store either an address or a size_t, both of which are immutable. The compiler can't assume anything regarding the object that resides at that address, therefore it will not perform any optimizations.

5. **Evaluation**

To evaluate our work we have created a synthetic benchmark that is designed to assess the performance implications of our implementation.

```
auto getSList(size_t steps, RCIAllocator alloc)
{
    SList!size_t a = SList!size_t(alloc);
    for (size_t i = 0; i < steps; ++i) {
        a.insert(i);
    }
    return a;
}

void benchmark(size_t steps, RCIAllocator alloc)
{
    for (size_t i = 0; i < 10; ++i) {
        auto a = getSList(steps, alloc);
        auto b = getSList(steps, alloc);
        auto c = a ~ b;
    }
}
```

Listing 5. Benchmark sample

Listing 5 highlights a high-level overview of the benchmarking code. We create two singly-linked lists using a custom allocator. We insert a number of elements (ranging from 10K to 40M) to analyze the implications of using different allocators.

Figure 3 exhibits the results of running our benchmark in 3 separate scenarios: (1) using the standard library implementation of a singly-linked list that uses the garbage collector, our singly-linked list implementation (tagged as exp) using both (2) the garbage collector and (3) the malloc-based allocator. Our observations show that as the number of allocated elements increase, our implementation that uses malloc is the fastest (with a maximum speed-up

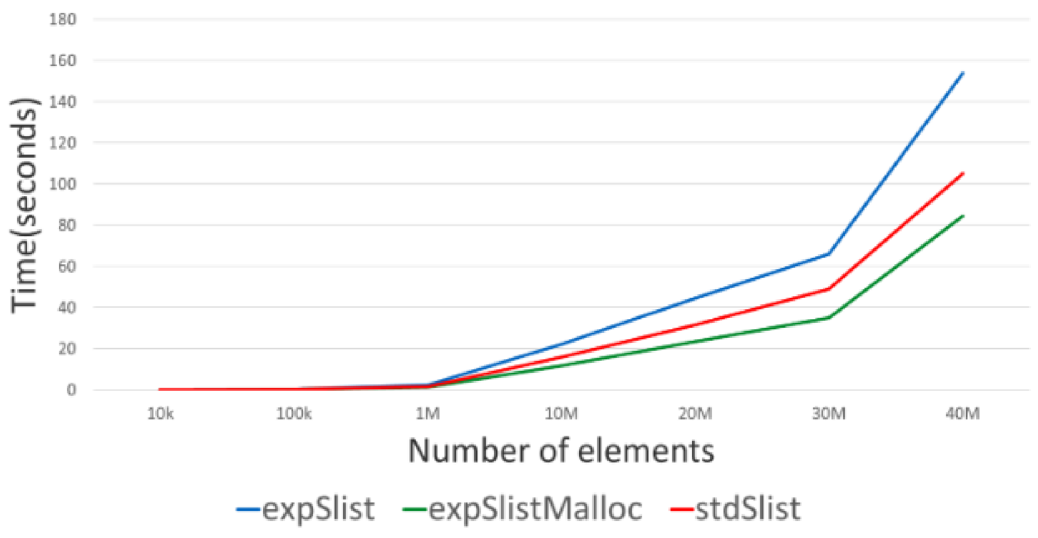

FIGURE 3. **Performance evaluation of singly-linked list using different allocators**: *stdSlist* is the Standard Library implementation that uses the GC; *expSlist* is the experimental framework implementation that uses the GC; *expSlistMalloc* is the experimental framework implementation that uses the MAllocator (malloc based allocator). *expSlistMalloc* shows the benefits of using a custom allocator

factor of 2, given our experimental setup). The difference is explained by the fact that malloc is a lighter alternative to the garbage collector. What is surprising, however, is that the standard library implementation is faster than the experimental singly-linked list implementation that uses the garbage collector. The explanation lies in the fact that in the first case, the garbage collector has the freedom to decide when memory is allocated and freed and therefore can optimally apply its memory allocation strategy, whereas, in the second scenario, calls to the garbage collector are inserted by the collection framework. This forces the garbage collector to act upon command, without taking into account its formal metrics.

Our results demonstrate that our singly-linked list implementation can be used in conjunction with a custom allocator to obtain better performance than the standard garbage collected implementation.

6. Conclusions

D is a growing language that needs to provide a strong suite of collections. Using the benefits provided by adhering to the ranges interface, the collections will fit right in with the algorithms in the standard library without any extra effort.

We have developed a new collections framework that brings together all the important features of the D Language, while being easy to use and intuitive for the user.

This novel approach of designing collections as ranges with optional primitives has proven to deliver better performance than the standard garbage collected alternative. Moreover, it offers flexibility to the user to choose the most suitable allocation strategy.

Additionally, we have successfully updated the allocators module to a safer API without impacting performance and improved the function attributes of the allocators API. This has improved the compile time type inference of the system. We have released the collections framework as a dub package and at the time of this writing the library has been downloaded 1293915 times.

REFERENCES

[1] Andrei Alexandrescu. On Iteration. http://www.informit.com/articles/printerfriendly/1407357, 2009.

[2] Mike Barnett, David A Naumann, Wolfram Schulte, and Qi Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on formal techniques for Java-like programs (FTfJP)*. Citeseer, 2004.

[3] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. Composing high-performance memory allocators. *ACM SIGPLAN Notices*, 36(5):114–124, 2001.

[4] Walter Bright, Andrei Alexandrescu, and Michael Parker. Origins of the d programming language. *Proceedings of the ACM on Programming Languages*, 4(HOPL):1–38, 2020.

[5] Ulrich W Eisenecker. Generative programming (gp) with c++. In *Joint Modular Languages Conference*, pages 351–365. Springer, 1997.

[6] D Language Foundation. D Language Specification - Implicit Qualifier Conversions. https://dlang.org/spec/const3.html#implicit_qualifier_conversions.

[7] Douglas Gregor and Sibylle Schupp. Making the usage of stl safe. In *Generic Programming*, pages 127–140. Springer, 2003.

[8] Paul Hudak. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 351–363, 1986.

[9] Oscar Llorente-Vazquez, Igor Santos, Iker Pastor-Lopez, and Pablo Garcia Bringas. The neverending story: Memory corruption 30 years later. In *Computational Intelligence in Security for Information Systems Conference*, pages 136–145. Springer, 2021.

[10] Takamichi Saito, Ryohei Watanabe, Shuta Kondo, Shota Sugawara, and Masahiro Yokoyama. A survey of prevention/mitigation against memory corruption attacks. In *2016 19th International Conference on Network-Based Information Systems (NBiS)*, pages 500–505. IEEE, 2016.

[11] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[12] Paul R Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, pages 1–42. Springer, 1992.