# DESIGNING COPY CONSTRUCTION FOR THE D PROGRAMMING LANGUAGE

Răzvan Nițu[1], Eduard Stăniloiu[1], Răzvan Deaconescu[2], Răzvan Rughiniş[3]

*The D programming language was developed as a natural replacement for the C and C++ languages with emphasis on memory safety, fast compile time, fast run time and advanced meta-programming. However, D makes use of a garbage collector that severely impacts runtime performance for low level applications. The garbage collector may be disabled, but then the user is constrained to manually manage the applications' memory.*

*The alternative is to implement a lightweight reference counted object as a library solution that manages the memory internally. Then, the user simply has to declare the desired object as being reference counted and everything will be handled with no extra effort. Although simple, this solution cannot be implemented currently, because of the incompatibility between certain functional aspects of the language, such as transitive type qualifiers, and the existing copy construction semantics.*

*In this paper, we propose a novel copy construction mechanism that can be used to implement reference counting in functional style contexts. Moreover, we design a generation strategy of both copy constructors and assignment operators for situations where objects are composed. Our design has been implemented and integrated in the D programming language and the current release version contains the novel copy constructor.*

**Keywords:** copy construction, language design, D, reference counting, functional

## 1. Introduction

In the past 20 years incredible advances were made in almost all areas of computer science. Distributed systems, operating systems, artificial intelligence, trading algorithms, IoT, acceleration hardware, gaming, virtual reality are just a few examples of fields that have have evolved significantly in the past

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

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

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

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

decade. This progress has created new market shares in the world of programming languages leading to an increased pressure on the existing languages [2] or emerging ones to develop new programming concepts and techniques that would minimize the time to market and maximize the performance of a program. As a result, the last two decades have witnessed the birth of a myriad of programming languages, domain specific and general purpose alike.

One of the promising new languages is D, an imperative, general purpose, systems programming language. D aims to fulfill the requirements of a full stack programming language, under the mantra "One language to rule them all". As a consequence, D has support for mechanical safety checks[9], functional programming [10], meta-programming[3], parallel programming, object oriented programming etc. Heavily inspired from popular languages like C, C++, Java and Python, D builds upon the features that exist in other languages: C-style syntax and manual memory management, Java-like classes and garbage collection, similar C++ template system, Scheme-like functional programming etc. Although some of the features are mutually exclusive (for example: manual memory management and garbage collection), the user may choose from the different options via command line switches.

Given the fact that D fights for the same market share as languages that have existed for more than 40 years, D comes with C, C++ interoperability out of the box[1], offering the possibility to gradually transition from old codebases to D[11]. In addition, most C code can be compiled with the D compiler with minor modifications.

Although D is uniquely positioned as the successor of C due to its compatibility with it and the additional modern programming techniques that it enables, the garbage collector still represents an obstacle when it comes to low level programming where performance is of utmost importance[13] [14]. The additional overhead that the garbage collector imposes outweighs the expressiveness benefits in a typical performance critical application (such as a kernel module). As a consequence, in such situations the garbage collector is disabled and the user has to manually manage the application memory.

In this context, the reference counting memory management technique presented in the form of a library solution [12] is a viable alternative. Reference counting is lightweight, easy to understand and to implement, but has the disadvantage that it is intrusive in the sense that the user must be aware of it and needs to make use of it. However, the intrusion is minimal: the user simply needs to declare that an object is being reference counted. Altogether, this makes reference counting the best substitute for garbage collection.

Implementing a minimal intrusive reference counting[6] mechanism requires that the language offers support for copy construction, move construction and their assignment counter-parts. Although D offers support for these operations, it does it in a manner that is incompatible with the transitive nature of the functional qualifiers such as **const** and **immutable**.

This paper proposes the implementation of a copy constructor in the D programming language that may be used in conjunction with transitive qualifiers, thus enabling the implementation of a reference counting mechanism as a library solution.

The remainder of this paper is organized as follows: Section 2 provides a base understanding of D features that relate to copy construction, Section 3 highlights the problems with the existing copy construction mechanism and Section 4 discusses the design aspects. Section 5 presents the evaluation and we conclude with Section 6.

## 2. Background

For a clear understanding of the problem definition, this section defines the term "copy construction", provides a brief presentation of type qualifiers are and showcases how copy construction was implemented in D prior to this work.

### 2.1. Copy Construction

In imperative programming languages copying an object from one location of memory to another is done by a bit-wise copy, as shown in Listing 1 (the listing employs C++ code, but the concept applies to most imperative languages).

```
1  struct A
2  {
3      int k;
4      int l;
5      int p;
6  }
7
8  void main()
9  {
10     A a = {2, 3, 4};
11     A b = a;
12 }
```

Listing 1. Bit-wise copy

The expression '**A b = a;**' has the effect of bit-copying the contents of **a** into **b**. The modification of an element of **a** is not going to be reflected in **b**. However, there are situations when bit-copying may lead to surprising behavior: pointer aliasing. When an object contains a pointer to some heap allocated chunk of memory, whenever an instance of that particular object is going to be bit-wise copied to another memory location, the 2 resulting instances will contain a pointer that references the same memory location.

This has proven to be problematic because although the two instances might be viewed as independent logical pieces, they are actually sharing memory.

The solution that is employed in most languages is to implement a copy function that takes care of creating the deep copy that needs to be manually called by the user. This solution has two shortcomings: the user has to manually call the function every time a copy is needed (increased prolixity) and the user may forget or not know that the copy function needs to be called (bug prone).

In response to these shortcomings, the C++ community has created and implemented the notion of a copy constructor: a user defined constructor function, defined in the scope of an aggregated declaration (struct or class) that is used to initialize an object from another object [7]. The copy constructor design has the advantage that the compiler implicitly inserts calls to it whenever a copy is created. This makes the code more expressive as there is no need to explicitly call a function and also it increases correctness as the user does not need to bother about copies - all of this is managed internally by the compiler.

### 2.2. Type Qualifiers

D type qualifiers modify a type by applying a type constructor. Type constructors are **const, immutable, shared, inout**. The most important aspect of D type qualifiers is that they apply transitively to each subtype[4], unlike C++'s **const** system (called head-const system) that applies only to the head reference. For example, given a pointer to an object **const P***, in C++ this is a mutable pointer to a **const** object **P**, while in D this is a **const** pointer to a **const** object **P**. If **P** would contain other pointers to other objects, in D, those would also be considered **const**, transitively, until a leaf definition (basic type) is encountered.

In addition to types, qualifiers may also be applied to member function of aggregated declarations (**class**, **struct**) with the meaning that the function will be called on a qualified object. This feature is useful to specify behavior depending on the particularities of each qualified type.

**immutable** data cannot change. Once constructed, **immutable** values do not modify throughout the entire execution of the program. **const** data cannot change through the current reference, however, there may exist other mutable references to it. Therefore, data that is accessed through a **const** reference, as opposed to **immutable**, may change.

In D, by default, variables are considered to be thread-local. In order to specify that a variable may be accessed from different threads, the **shared** qualifier is used. In addition, **shared** data access is required to be synchronized by means of atomic read and atomic write, thus ensuring correctness of parallel algorithms.

### 2.3. Copy Construction in D

Although the C++ notion of a copy constructor offers many advantages, it has some drawbacks in specific situations. For example, if an object contains a lot of fields that are fine with normal bit-wise copies and only a few of them need to be properly handled for deep copying, the copy constructor will have to initialize all the fields. Consider **Listing 3** where a typical C++ copy constructor is showcased. In this situation, when an object of type **A** is copy constructed from from another object of the same type for the majority of fields shallow (bit-wise) copying is the correct action; the only field that needs special treatment is the **p** pointer that needs to do a deep copy. However, with the current semantics, the user still needs write all the boilerplate[5] field initialization code, resulting in an extra 102 lines of code. One solution to handle this boilerplate would be to not define a copy constructor and let the compiler handle the bit-wise copy, after which the user may manually perform the deep copy of the **p** pointer. This indeed solves the boilerplate code problem, however it requires un-encapsulated attention at each copy site, making it an invalid solution.

```
1  struct A
2  {
3      int *p;
4      int size;
5      int a1, a2, ..., a100;
6
7      // Copy Constructor
8      A(const A &a)
9      {
10         this->p = malloc(a.size * sizeof(int));
11         for (int i = 0; i < a.size; i++)
12             this->p[i] = a.p[i];
13
14         this->a1 = a.a1;
15         this->a2 = a.a2;
16         ...
17         this->a100 = a.a100;
18     }
19 }
```

Listing 2. C++ Copy Constructor

The D implementation of copy constructors leverages this observation and automatically takes care of the shallow copy. This is achieved by initializing the destination with a bit-copy of the source. After this operation control is passed to a user defined function, called **postblit**. The posblit simply adjusts the fields that need updating, in our situation, the pointer**p**. The postblit

is defined as **this(this)** and has access only to the destination, therefore the source cannot be modified.

## 3. Motivation

Although the postblit has a clear advantage over the normal copy construction scheme, it cannot cohabit with a powerful language feature: the transitivity of type qualifiers. This section will highlight the nature of the conflict between the two language features.

The postblit function cannot be meaningfully overloaded or qualified. However, the compiler does not reject certain applications of qualifiers, as illustrated below:

```
1 struct A { this(this) const {} }
2 struct B { this(this) immutable {} }
3 struct C { this(this) shared {} }
```

Listing 3. Qualified Postblit

The semantics of the postblit in the presence of qualifiers is not defined, and experimentation reveals that the behavior is a patchwork of happenstance semantics: (1) **const** postblits are not able to modify any fields in the destination, (2) **immutable** postblits never get called (resulting in compilation errors) and (3) **shared** postblits cannot guarantee atomicity while bit-wise copying the fields.

Defining and implementing some meaningful semantics will break code that has been tested and deemed correct under the current semantics.

### 3.1. const / immutable postblits

A solution for **const** and **immutable** postblits would be to type check them as normal constructors, where the first assignment of a member is considered an initialization and subsequent assignments count as modifications. This is problematic because after the blitting phase, the destination object is no longer in its initial state and subsequent assignments to its fields will be regarded as modifications, making it impossible to construct nontrivial **immutable/const** objects in the postblit. In addition, it is possible for multiple postblits to modify the same field. Consider Listing 4.

When **B c = b;** (line 18) is encountered, the following actions are taken: (1) **b**'s fields are blitted to **c**, (2) **A**'s postblit is called and (3) **B**'s postblit is called

After step 1, the object **c** has the exact contents as **b**, but it is neither initialized (the postblits still need to run) nor uninitialized (the field **B.a** does not have its initial value). From a type checking perspective this is a problem because the assignment inside **A**'s postblit is breaking immutability. This makes it impossible to postblit objects that have **immutable** / **const** fields. To alleviate this problem one could consider that after the blitting

phase the object is in a raw state, therefore uninitialized; this way, the first assignment of **B.a.a** is considered an initialization. However, after this step the field **B.a.a** is considered initialized, therefore how is the assignment inside **B**'s postblit supposed to be type checked? Is it a violation of immutability, or should it be legal? Indeed, it is breaking immutability because it is changing an **immutable** value. However as this is part of initialization (recall that **c** is initialized only after all the postblits are ran) it should be legal, thus weakening the immutability concept and creating a different strategy from that implemented by normal constructors.

```
1  struct A
2  {
3      immutable int a;
4      // modifying immutable, an error or OK?
5      this(this) { this.a += 2; }
6  }
7
8  struct B
9  {
10     A a;
11     // modifying immutable, an error or OK?
12     this(this) { this.a.a += 2; }
13 }
14
15 void main()
16 {
17     B b = B(A(7));
18     B c = b;
19 }
```

Listing 4. **const** and **immutable** postblits

### 3.2. shared postblits

**shared** postblits cannot guarantee atomicity while blitting the fields because that part is done automatically and it does not involve any synchronization techniques. Consider **A b = a;** is executed in a multithreaded environment: (1) **a**'s fields are copied to **b** bitwise ("blitted") and (2) **this(this)** is called. In the blitting phase, no synchronization mechanism is employed, which means that while copying is in progress, another thread may modify **a**'s data, resulting in the corruption of **b**.

## 4. Design

As discussed above, the postblit is difficult to type check without unreasonable restrictions and cannot be synchronized without undue costs.

This paper proposes the implementation of a copy constructor with the following benefits: (1) the feature is used to good effect in the C++ language, (2) the copy constructor can be type checked as a normal constructor - since no blitting is done, the fields are initialized the same as in a normal constructor. This offers the benefit that **const** / **immutable** / **shared** copy constructors will be type checked exactly as their analogous regular constructors and (3) offers encapsulation.

The downside of this solution is that the user must copy all fields by hand, and every time a field is added to a **struct**, the copy constructor must be modified. However, this issue can be easily worked around by using D's introspection mechanisms.

### 4.1. Syntax

Inside a struct definition, a declaration is a copy constructor declaration if it is a constructor declaration that takes the first parameter as a non-defaulted reference to the same type as the struct being defined. Additional parameters may follow if and only if all have default values. Declaring a copy constructor in this manner has the advantage that no parser modifications are required, thus leaving the language grammar unchanged. Listing 5 highlights the described syntax.

The argument to the copy constructor is passed by reference in order to avoid infinite recursion. Note that if the source is an rvalue, no call to the copy constructor is necessary because the value will be bit-wise moved into the destination.

Type qualifiers may be applied to the parameter of the copy constructor and also to the function itself in order to allow defining copies across objects of different mutability levels. The type system handles the call to the best matching copy constructor.

```
1  struct A
2  {
3      this(ref A rhs) { writeln("x"); }
4      this(ref A rhs, int b = 7) immutable
5      { writeln(b); }
6  }
7  void main()
8  {
9      A a;
10     A b = a;              // prints "x"
11     A c = A(b);           // prints "x"
12     immutable A d = a;    // prints 7
13 }
```

Listing 5. Copy Constructor Syntax

### 4.2. Semantics

**Copy constuctor and postblit cohabitation**. In order to ensure a smooth transition from postblit to copy constructor, this report proposes the following strategy: if a struct defines a postblit (either user-defined or generated), all copy constructor definitions will be ignored for that particular struct and the postblit will be preferred. Existing code bases that do not use the postblit may start using the copy constructor, whereas code bases that currently rely on the postblit may start writing new code using the copy constructor and remove the postblit from their code.

**Copy constructor implicit calls**. A call to the copy constructor is implicitly inserted by the compiler whenever a struct variable is initialized as a copy of another variable of the same unqualified type: (1) when a variable is explicitly initialized, (2) when a parameter is passed by value to a function and (3) when a variable is returned by value from a function. The parameter of the copy constructor is passed by reference, so initializations will be lowered to copy constructor calls only if the source is an lvalue. Although this can be worked around by declaring temporary lvalues which can be forwarded to the copy constructor, binding rvalues to lvalues is beyond the scope of this paper.

**Type checking**. The copy constructor type check is identical to that of the constructor. The major difference between a constructor type check and a normal function type check is the following: in constructors, the first assignment of a field is considered initialization while subsequent assignments are viewed as modifications; in normal functions, all assignments are viewed as modifications. This difference stems from the fact that constructors are supposed to initialize an object, including its **const** or **immutable** fields. Once the initialization of non-mutable fields is done, their values are locked down and attempts to modify them are signaled as compilation errors. This is the fundamental reason why the postblit function could not accommodate qualifiers - it had a normal function type-check and it couldn't update non-mutable fields.

**Overloading**. The copy constructor can be overloaded with different qualifiers applied to the parameter - copying from a qualified source - or to the copy constructor itself - copying to a qualified destination. The proposed model enables the user to define the copy from an object of any qualified type to an object of any qualified type: any combination of two among mutable, **const**, **immutable**, **shared**, **const shared**. The **inout** qualifier may be applied to the copy constructor parameter in order to specify that mutable, **const**, or **immutable** types are treated the same. In the case of partial matching, existing overloading and implicit conversion rules apply to the argument.

**Copy Constructor call vs. Blitting**. When a copy constructor is not defined for a struct, initializations are handled by copying the contents from the memory location of the right-hand side expression to the memory location of the left-hand side expression (i.e. "blitting"). When a copy constructor

is defined for a struct, all implicit blitting is disabled for that struct. This decision was taken to prevent the situations where the user forgets to define a copy constructor for a specific transformation.

```
1  struct M
2  {
3      this(ref immutable M) {}
4      this(ref immutable M) shared {}
5  }
6
7  struct S
8  {
9      M m;
10     this(ref shared S) {}
11     // Generated copy constructors:
12     // this(ref immutable S)
13     // this(ref immutable S) shared
14 }
```

Listing 6. Example of Generated Copy Constructors

For example, if a user wants to implement reference counting, but forgets to define a copy constructor for **const** to **const** copies when a **const** object is going to be initialized from another **const** object, a bit-wise copy is going to be made and the reference count is going to miss one increment. With the proposed solution, this situation is going to be flagged with a compilation error.

**Generation of copy constructors**. A copy constructor is generated for a **struct S** if any member of **S** defines a copy constructor that **S** does not define. A field of a **struct** may define multiple copy constructors. In this situation, a copy constructor is generated for each overload. Listing 6 exhibits the implemented behavior. The body of all the generated copy constructors performs member-wise initialization. Inside the body of the generated copy constructors, each field assignment will be rewritten as a call to corresponding copy constructor. Blitting is employed where possible for each field that does not define a copy constructor. If any field is non-copyable (e.g. the copy constructor is disabled), the generated copy constructor will be annotated with **@disable**, rendering it unusable. If the copy constructors of of some fields match in terms of qualifiers, a single copy constructor is generated.

## 5. Evaluation

To verify the soundness of our design, we have implemented our proposal in the reference compiler [8] and substituted all uses of the postblit in the D standard library with the novel copy constructor.

Our implementation uses a fork of the reference compiler (dmd) and its standard library. The changes that we had to make were to simply substitute the posblit declaration **this(this)** with a copy constructor one **this(ref inout(T) rhs) inout** and use the code snippet presented in Listing **??** to do the copy of fields. This works because the postblit behaves correctly only for situations where the type of the source exactly matches the type of the destination. Moreover, profiling the test suite of the D standard library has shown there is no significant performance difference between the postblit and copy constructor versions. The D standard library requires that any implemented feature has at least one unittest. We could not decide which unittests end up calling a postblit/copy constructor, therefore, we have ran the entire test suite and timed it with both the postblit and copy constructor implementations. We have timed 100 such runs and computed the average time for each of the implementations. The results were: 5 minutes 47 seconds for the postblit implementation and 5 minutes and 56 seconds for the copy constructor. Given that not all unittests result in the call of a copy constructor, we conclude that the difference is negligible. Our findings are on par with our expectations since the operations performed are largely the same.

In addition, we have tried to implement a reference counted object, however, we did not succeed. The problem stems from the fact that a reference counted object needs to update a counter payload that is stored **inside** the object. Once a reference counted object instance is declared as **immutable**, the payload field cannot be updated. As a consequence, further language changes are needed to enable the user to break the transitivity of qualifiers in certain scenarios.

## 6. Conclusions

In this work we have presented a new design for copy construction for the D programming language. Our approach is able to combine copy construction and type qualifiers so that a destination object can be copy constructed from a differently qualified source object.

We have implemented our approach in the reference D compiler and showed that it can successfully substitute the existing postblit function with no loss in expressiveness or performance. However, we are still not able to implement reference counting for qualifier objects. A mechanism to break the transitivity of qualifiers is an interesting topic for future work.

Our proposal has been analyzed by the D standard committee and has been subjected to the rigors of the "D improvement process". Both the community and the leadership have found that our design is a suitable substitute for the postblit. As a consequence, our proposed implementation has been integrated in the language and is being used commercially.

# REFERENCES


[1] 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.

[2] W. S. Curran. The future of programming languages. In *Proceedings of the 30th Annual Southeast Regional Conference*, ACM-SE 30, pages 196–202, New York, NY, USA, 1992. ACM.

[3] Krzysztof Czarnecki, Kasper Østerbye, and Markus Völter. Generative programming. In *European Conference on Object-Oriented Programming*, pages 15–29. Springer, 2002.

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

[5] Jaakko Järvi, Mat Marcus, and Jacob N Smith. Programming with c++ concepts. *Science of Computer Programming*, 75(7):596–614, 2010.

[6] Lutz Kettner. Reference counting in library design—optionally and with union-find optimization. *Library-Centric Software Design (LCSD'05)*, pages 1–10, 2005.

[7] Daniel Lincke and Sibylle Schupp. The function concept in c++: An empirical study. In *Proceedings of the 2009 ACM SIGPLAN Workshop on Generic Programming*, WGP '09, pages 25–36, New York, NY, USA, 2009. ACM.

[8] Razvan Nitu. Implementing copy construction. https://github.com/dlang/dmd/pull/8688.

[9] Razvan Nitu, Eduard Staniloiu, Cristian Done, and Razvan Rughinis. Security audit for the d programming language. In *2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pages 1–6. IEEE, 2021.

[10] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 71–84, New York, NY, USA, 1993. ACM.

[11] Eduard Staniloiu, Razvan Nitu, Cristian Becerescu, and Razvan Rughinis. Automatic integration of d code with the linux kernel. In *2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, pages 1–6. IEEE, 2021.

[12] Baltasar Trancón y Widemann. A reference-counting garbage collection algorithmfor cyclical functional programming. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 71–80, New York, NY, USA, 2008. ACM.

[13] Katsuhiro Ueno and Atsushi Ohori. A fully concurrent garbage collector for functional programs on multicore processors. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 421–433, New York, NY, USA, 2016. ACM.

[14] David Ungar, David Grove, and Hubertus Franke. Dynamic atomicity: Optimizing swift memory management. In *Proceedings of the 13th ACM SIGPLAN International Symposium on on Dynamic Languages*, DLS 2017, pages 15–26, New York, NY, USA, 2017. ACM.